Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

HASDF: A Generalized LR-parser Generator for Haskell

Merijn de Jonge, Tobias Kuipers, and Joost Visser

Software Engineering (SEN)

SEN-R9902 January 1999

Report SEN-R9902
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

HASDF: A Generalized LR-parser Generator for Haskell

Merijn de Jonge
Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
mdejonge@wins.uva.nl

Tobias Kuipers
CWI
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands
Tobias.Kuipers@cwi.nl

Joost Visser
Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
jvisser@wins.uva.nl

ABSTRACT

Language-centered software engineering requires language technology that (i) handles the full class of context-free grammars, and (ii) accepts grammars that contain syntactic information only. The syntax definition formalism SDF combined with GLR-parser generation offers such technology. We propose to make SDF and GLR-parsing available for use with various programming languages.

We have done so for the functional programming language Haskell. By combining Haskell data type definitions with the syntax definition formalism SDF we have designed HASDF. HASDF is a domain-specific language in which the *concrete* syntax of an arbitrary context-free language can be defined in combination with the Haskell data types that represent its *abstract* syntax. We have implemented a tool that generates a GLR-parser and an unparser from a HASDF definition.

1991 Computing Reviews Classification System: D.1.1, D.1.2, D.2.2, D.2.6, D.2.7, D.3.4

Keywords and Phrases: Functional programming, parser generation, Haskell, ASF+SDF, program generators, language design

Note: Work carried out under project SEN-1.1, *Software Renovation*.

1 Introduction

Motivation

Real-world software systems are multi-lingual in several respects. Their components are usually programmed in a number of different languages. These components can communicate with each other and with their environment using a variety of formal languages, such as data formats, command sets, query languages, and programming languages.

These languages can be of varying degrees of complexity and domain-specificity, and evolve continually into new, often co-existing dialects. The design of these languages has created a strong need for formal grammar definitions. For the cost-effective development and maintenance of systems that use these languages it is essential that tools are *generated* from these grammars [23]. An example of such a tool is a parser.

Unfortunately, common tool generation technology, e.g. Yacc [11], does not allow generation of tools from arbitrary plain grammar definitions. These parser generators put restrictions on the class of (context-free) grammars they accept, and they need grammar descriptions to be augmented with semantic actions. These properties can be useful in application areas that involve a single, stable language. However, in application areas where languages are not fixed but subject to change,

these properties are harmful [25].

An example of such an application area is the renovation and analysis of legacy software systems. A typical legacy language, such as Cobol, exists in many dialects, some of which are not documented. In order to parse programs in these languages, the grammar needs to be modified for each dialect. We will call these application areas "language-centered" in the remainder of this paper.

For such areas, more advanced language technology is needed. In the ASF+SDF Meta-Environment [12] such technology is available in the form of the syntax definition formalism SDF [8] combined with GLR parser generation [19]. GLR parsers can be generated from *any* context-free grammar. SDF allows *any* context-free grammar to be defined.

Plan

Currently SDF is tightly coupled with the algebraic specification formalism ASF. We propose to make SDF and GLR-parser generation available for use with other programming languages besides ASF. This can be done by defining a syntax definition formalism for such a programming language that combines SDF with the data structure declarations of the language at hand. In such a combined formalism, the concrete syntax of an arbitrary context-free language can be defined in conjunction with its abstract syntax. From this simultaneous definition of concrete and abstract syntax, a GLR parser can be generated, as well as an unparser, that can interoperate with programs written in the programming language.

For the functional programming language Haskell, we have defined such a formalism for simultaneous definition of concrete and abstract syntax, called HASDF. This formalism allows Haskell data type declarations to be annotated with the concrete syntax they are to represent. Also, a tool has been implemented that generates parsers and unparsers from HASDF definitions.

Outline

The paper is organized as follows. In the first part we asses common language technology. Section 2 explains why common parser generation tools can not meet the demands of application areas involving multiple, rapidly evolving languages, and why SDF with GLR parser generation can. Section 3 discusses existing parser technology for functional programming languages in particular.

In the second part of the paper, alternative language technology is presented. Section 4 contains a concise presentation of the syntax definition formalism SDF. In section 5 we explain how we combined SDF with Haskell data type definitions into the syntax definition formalism HASDF. We present the HASDF-tool, and make some remarks about its implementation.

Section 6 summarizes our work and lists possible directions of future work.

2 Generating parsers

Chomsky [2] classified languages into a hierarchy of four types. The largest class of languages for which efficient parsing methods are known is the class of context-free grammars. In this section we will discuss why common technology for generating parsers from context-free grammars does not satisfy the needs of language-centered software engineering areas. A more elaborate account can be found in van den Brand *et. al.* [25, 23]

Grammar restrictions and impurity

The application of common language technology to language-centered software engineering is problematic. These problems mainly derive from the following two sources:

Restrictions Input is restricted to some subclass of the context-free grammars.

Impurity Input does not consist of plain grammars, but of grammars that are augmented with information that serves other purposes than defining syntax.

Most parser-generators do not accept arbitrary context-free grammars, but restrict their input to some subclass of context-free grammars, such as LL(1), LR (1), LL(k), or LALR-grammars [7]. The most widely known and used parser generator, Yacc [11], accepts LALR-grammars only. This is also the case for a large number of Yacc-derivatives, such as Bison. Parser generators that are restricted to LL(k) grammars include ANTLR [16].

The input to most parser-generators does not consist of a plain grammar. Each production needs to be annotated with a semantic action – a block of C code in

the case of Yacc – that performs parse tree building operations or directly emits output per language construct. Semantic actions give the grammar writer ample control over the behavior of the parser that is generated. This allows him to circumvent some grammar restrictions, and to obtain improved space and time efficiency. Since semantic actions serve different purposes than syntax definition, the grammars that contain them can be said to be *impure*. Another source of impurity besides semantic actions are compiler directives.

When dealing with a single, relatively stable language, grammar restrictions and impurity do not present serious problems. Most programming languages can also be described by a LALR grammar, although this description in general is not the most natural. Defining grammars with semantic actions is more difficult than defining plain grammars, but offers a high level of control over the behavior of the generated parsers. For instance, if a language contains user-definable syntax, the parser can modify itself during the parse of a program in such a language. Another example is the parsing of the so-called offside rule in Haskell. Using semantic actions this rule can be parsed in one pass. Using a parser without semantic actions requires a preprocessing phase.

If, however, we are dealing with languages under development or under renovation, grammar restrictions and impurity can be harmful.

Why restrictions are not harmless

Language-centered software engineering areas, such as language prototyping, program transformation, and software renovation, create demands that are not met by common parser generators. To make the design and maintenance of multiple, evolving languages feasible, it is essential that language technology is used that can handle *arbitrary* context-free grammars and that allows these grammars to be described in a *pure* syntax definition formalism.

The grammar restrictions imposed by common language technology have two important consequences. Firstly, they create the need for grammar transformations. Secondly, they break the property of compositionality of grammars.

Many languages are described most naturally by a context-free grammar. Moreover, the class of context-free grammars does not have a proper subclass that (i) includes all unambiguous context-free grammars, and (ii) is closed under composition [19, page 12]. So, restriction to a proper subclass of the context-free grammars forces the grammar writer to construct an artificial encoding of his intentions. For many languages the context-free grammar that constitutes their natural expression will need to be transformed into a grammar in the restrictive subclass, before parsers or other tools can be generated. This transformation of grammars often takes substantial effort, and can not in general be done automatically [7, page 251]. Furthermore, grammar transformation leads to parse trees that are different from the ones that would correspond to the original grammar.

Without the property of compositionality, languages can not be developed incrementally, or constructed by composition of smaller languages. For instance, when two LALR-grammars are merged, the resulting grammar may need *substantial* modification (resolution of parse-table conflicts) before a new LALR-grammar is obtained again. Also, local changes to a language may require global changes in its grammar. Consequently, when confined to a subclass, context-free grammars and tools based on these grammars will need substantial re-engineering before they can be reused.

In areas that involve a single stable language, the need for grammar transformations and the loss of the property of compositionality does not present unsurmountable problems. Grammar transformation needs to be done only once, and there is no need for incremental or compositional grammars. However, in language-centered software engineering, the loss of these properties will lead to unacceptably high development and maintenance costs. The effort put into grammar transformation needs to be reinvested after each language change, dialects of the same language must be developed and maintained separately, and features of existing languages can be reused in new languages only through substantial re-engineering. For these reasons, nothing less than the full class of context-free grammars will do for language-centered software engineering.

Why impurity is not harmless

Related problems are created by impurity. Allowing information in grammars that does not serve to define syntax, leads to various development and maintenance problems in language-centered areas.

The maintenance and development of impure grammars also requires maintenance and development of the code that constitutes the semantic actions. When the grammars are not stable, this additional effort needs to

be invested repeatedly. Semantic actions can also break the compositionality of grammars, making their reuse problematic or impossible.

Impure grammars are less versatile. Semantic actions and compiler directives serve to optimize the generation process and the generated parsers. As a result, they make it hard to employ the grammars used for parser-generation for other purposes, such as unparser-generation. The same impure grammar can not be used as input for different tool generators.

Parsers generated from impure grammars produce abstract syntax trees that do not correspond in a standard way to the concrete terms they represent. Semantic actions offer a programmer full control over the construction of trees during parsing. As a result, the relation between concrete and abstract syntax becomes obfuscated: it is (partly) defined through the semantic actions. To know what the abstract syntax is not only requires knowledge of the syntax of the language, but also of the operation of the generated parser. This knowledge is needed to develop programs that process abstract syntax trees. Thus, the development of tools that employ a parser generated from a grammar is substantially more complicated when the grammar is impure.

Finally, impurity in grammars requires a more complex syntax definition formalism. Allowing semantic actions in a syntax definition language has the effect of importing an entire general-purpose programming language into the formalism. This cancels the maintenance advantages to be gained by a syntax definition language as a domain-specific language [4].

In summary, language-centered software engineering requires language technology that handles the full class of context free grammars and does not allow grammars to be extended with information that does not serve to define syntax. Without such technology, the development and maintenance of multiple, evolving languages is not cost-effective, or even feasible.

SDF and GLR-parser generation

In the ASF+SDF Meta-Environment [12] more advanced language technology is available in the form of the syntax definition formalism SDF [8] combined with GLR-parser generation [19] from SDF specifications.

In SDF any context-free grammar can be defined, as well as disambiguation rules for such a grammar. Thus, an unambiguous grammar can be defined in SDF by an ambiguous grammar and separate disambiguation rules. This offers many advantages over restriction to a subclass of context-free grammars and encoding disambiguation into the grammar [1, page 247]. SDF is a pure syntax definition formalism. It does not allow any information that does not serve the purpose of defining syntax.

In ASF+SDF, SDF is combined with GLR parser generation. The theoretical framework for generalized LR parsing was introduced by Lang [14]. The GLR parsing algorithm [19] is an improvement of Tomita's universal parsing algorithm [21]. Tomita's algorithm is restricted in the context-free grammars that it can handle because it loops on cyclic grammars. In the GLR algorithm this problem has been fixed, which resulted in a time efficient algorithm, which is as strong as Earley's universal context-free parsing algorithm [5]. For LALR-grammars, GLR-parsers are comparable in performance to LALR-parsers. The generation time for GLR-parsers is in general better than for LALR-parsers [19].

Due to the availability of SDF and GLR parser generation, the ASF+SDF-system has proven to be highly suitable for, for instance, language prototyping and software re-engineering [3, 27, 26].

3 Parsing and functional programming

In this section we will assess available parsing technology for functional programming languages in general, and for Haskell in particular. In the literature, two basic approaches to parsing in combination with functional programming can be found:

Parser combinator libraries Manually construct a parser in a functional language from a set of parser combinators offered by a library.

Parser generators Generate a parser from a grammar.

This dichotomy is not strict, since a combined approach of generating a combinator parser from a grammar could also be imagined. (The parser generator Lucky [13] seems to be an example of this approach). Moreover, the combinator approach can be seen as an instance of the generator approach. According to this viewpoint, the parser combinator library defines a syntax definition formalism, that allows the grammar of a language to be expressed in terms of the combinators,

```
exports
  sorts
    Varname Type
  context-free syntax
    Type "→" Type      → Type {right}
    Varname            → Type
    "(" {Type ","}+ ")" → Type
    "[" Type "]"       → Type
    "(" Type ")"       → Type {bracket}
  priorities
    "(" Type ")" → Type >
      "(" {Type ","}+ ")" → Type
  lexical-syntax
    [␣\t\n]            → LAYOUT
    [a-z] [A-Za-z0-9]* → Varname
```

Figure 1: Definition in SDF of the syntax of (highly simplified) Haskell types.

and the semantic actions in terms of the underlying programming language.

All parser combinator libraries known to us enable the construction of top-down parsers. These are either deterministic [20] or non-deterministic [10, 6, 9]. Hence, they are restricted to LL(1) or LL(k) grammars. They are unable to handle left-recursive rules or cyclic grammars. As syntax definition formalisms, combinator libraries are highly impure.

Happy [15] and ML-Yacc [18] are parser generators for Haskell and ML, respectively. They are both modeled after Yacc. Their syntax is similar to the Yacc syntax, and they are restricted to LALR-grammars. As semantic actions they allow Haskell and ML code. Thus, Happy and ML-Yacc suffer from the same grammar restrictions and impurities as Yacc.

4 The syntax definition formalism SDF

Like BNF, SDF is a formalism for syntax definition. Among its improvements over BNF are support for modularity and a disambiguation mechanism. We will explain SDF by piecewise presentation of an example grammar. This grammar defines the (simplified) syntax of Haskell types. The complete example is listed[1] in Figure 1.

[1]The code examples in this paper are formatted using the generic pretty-printer described in [22]

Context-free syntax

The context-free syntax of Haskell types can be described by the following SDF productions:

```
Type "→" Type      → Type
Varname            → Type
"(" {Type ","}+ ")" → Type
"[" Type "]"       → Type
"(" Type ")"       → Type
```

Note that, with respect to BNF productions, the direction of these SDF productions is reversed. These productions define function types, type applications, type variables, type constants, tuple types, and list types, respectively. In SDF, non-terminals start with a capital letter (`Type`, `Varname` and `Constrname` in this example). Terminals start with a lower case letter, or are enclosed in double quotes if they contain symbols or start with an upper case letter. The last of these productions contains an *iterator*. The expression `{Type ","}+` stands for any non-empty sequence of the non-terminal `Type`, separated by commas.

These context-free syntax productions form a grammar that is incomplete in several ways. First of all, the lexical non-terminals `Varname` and `Constrname` stand in need of a definition. Secondly, the grammar is ambiguous. We will explain how SDF allows the grammar to be disambiguated and how its lexical syntax can be defined.

Attributes

Ambiguities that involve a single production can be disambiguated in SDF by the use of *attributes*. For instance, the first production causes expressions that contain repeated function types, such as `a->b->c` to be ambiguous between `a->(b->c)` and `(a->b)->c`. To remove the ambiguity, we add the attribute `right`:

```
Type "→" Type → Type {right}
```

This attribute specifies function types to be right-associative, which will favor the first alternative over the second. Likewise, type application can be specified to be left-associative using the attribute `left`.

There is a third attribute available in SDF called `bracket`. In the example, the production for parenthesized types can be annotated with `bracket`:

“(” Type “)” → Type {bracket}

The meaning of this annotation is that a parenthesized type will be considered to be syntactically equivalent to its non-parenthesized version. This is useful, because the reason to introduce type parentheses was not to introduce new syntax, but to allow type associations to be expressed that disambiguate or override the disambiguation defined by other attributes or priorities.

Priorities

Ambiguities that are due to the interaction of two different productions can be eliminated in SDF by specifying *priorities*. In a **priorities**-clause, chains of relative priorities between context-free productions can be defined. In the example, there is an ambiguity between parenthesized types and the singleton tuple type. It can be removed with:

“(” Type “)” → Type >
  “(” {Type “,”}+ “)” → Type

The priority declaration defines a decremental ordering of productions. It gives singleton tuple types priority over parenthesized types. Besides a decremental ordering, productions can be ordered incrementally as well ($p_1 < p_2 \ldots < p_n$).

Note that SDF offers a disambiguation mechanism based on *relative* priorities. Relative priorities offer better extensibility than absolute priorities.

Lexical syntax

The syntax of lexical non-terminals is specified in SDF with a **lexical syntax**-clause. The lexical productions listed in this clause make use of character-classes (regular expressions) to indicate which characters can make up each lexeme. The lexical syntax for variable names in our example is given by:

[a-z] [A-Za-z0-9]* → Varname

Hence, a variable name starts with a lower case letter and is followed optionally by a sequence of letters (both lower and upper case) and digits.

The lexical syntax section is also used to specify the special non-terminal LAYOUT. For instance, spaces, tabs, and new-lines are specified to qualify as layout characters as follows:

```
module Types
exports
  sorts
    Varname Type
  datatypes
    data Type =
      Arrow Type Type ⋈
        Type “→” Type → Type {right} |
      Var Varname ⋈ Varname → Type |
      Tuple [ Type ] ⋈ “(” {Type “,”}+ “)” → Type |
      List Type ⋈ “[” Type “]” → Type
  context-free syntax
    “(” Type “)” → Type {bracket}
  priorities
    “(” Type “)” → Type >
      “(” {Type “,”}+ “)” → Type
  lexical syntax
    [␣\t\n]              → LAYOUT
    [a-z] [A-Za-z0-9]* → Varname
```

Figure 2: Definition in HASDF of the concrete and abstract syntax of (highly simplified) Haskell types.

[␣\t\n] → LAYOUT

The effect of this definition is that any number of these characters can be inserted between context-free symbols.

Modularity

An important aspect of SDF that does not feature in our small example is its support for modularity. A SDF-specification can be distributed over several modules. The productions belonging to a single non-terminal are not required to be in the same module.

5 Combining SDF with Haskell data types

When concrete syntax is combined with a programming language, we can distinguish two programming techniques. The first technique will be called *abstract programming*. According to this technique, programs are written to operate on abstract syntax trees. Parsing and unparsing is performed explicitly in the program and is strictly separated from processing abstract syntax trees. The second technique, called *concrete pro-*

gramming, is a programming technique in which programs are written in concrete (rather than abstract) syntax. This programming technique allows programs to operate on concrete syntax directly. Parsing and unparsing is performed implicitly. ASF+SDF is an example of a system that fully supports concrete programming.

Abstract programming does not affect current Haskell programming and therefor we believe it is more appealing to Haskell programmers than concrete programming. Hence, we decided *not* to introduce concrete programming in Haskell yet, but consider it future work.

To create a parser generator to be used for abstract programming, we need a formalism for the simultaneous definition of abstract and concrete syntax. We defined HASDF as syntax definition formalism to combine SDF with Haskell. A HASDF specification defines an explicit mapping between concrete and abstract syntax. Since we do not support programs written in concrete syntax, our approach requires *explicit parsing*. Hence, a GLR parser has to be integrated explicitly as part of the program. We provide a tool that generates a GLR parser from a HASDF module. This tool could have been designed to produce a Haskell implementation of a GLR parser. However, because no GLR parser generator currently exists that produces Haskell code, we decided to generate non-Haskell code which enables us to reuse existing GLR parser generator implementations. The generated GLR parser will be connected to a Haskell program as a preprocessor or through external function calls.

In the remainder of this section we will first describe the HASDF formalism. Next we describe the HASDF tool that generates such a GLR parser from a HASDF specification. Finally, we will discuss several implementation details of this tool.

The syntax definition formalism HASDF

We have combined SDF with Haskell data types. We have done this by adding a clause to the SDF language. This clause is marked by the keyword **datatypes**. In such a clause, Haskell data types can be written. Each constructor declaration in such a data type definition can be annotated with an SDF style context-free grammar rule. For instance, Figure 2 shows a HASDF definition of the abstract and concrete syntax for types. This HASDF definition combines the concrete syntax of Haskell types from Figure 1 with abstract syntax defined as Haskell data type.

```
module Types
  where
  data Type =
    Arrow Type Type |
    Var Varname |
    Tuple [ Type ] |
    List Type
      deriving
        ( Read , Show )
  type LAYOUT = String
  type Varname = String
```

Figure 3: Haskell module generated by the HASDF tool from the HASDF module of Figure 2.

In this way, abstract syntax and concrete syntax can be specified in conjunction. Note that no abstract syntax is given for lexical sorts. All lexical sorts will be represented by the Haskell type `String` using type synonyms.

We do not allow just any Haskell data type definition. There are a number of restrictions because only data types that can represent abstract syntax trees make sense in the context of HASDF. For instance, parameterized data type constructors are not allowed. Furthermore, list types occurring nested in list or tuple types are not supported, because there is no immediate translation for them in SDF. This does not restrict the expressiveness however, because using an intermediate sort, the same thing can be effected. See the appendix of this report for the syntax of data type declarations that are allowed.

Built-in Haskell types are supported in HASDF by means of the implicit imported module 'Prelude', which contains their concrete syntax. Types like `Integer` and `String` can therefor be used without having to define their lexical syntax. The generated parser automatically maps these types to corresponding Haskell types.

The reader should note here that the language HASDF has been designed for readability in the first place. The syntax of the language allows for constructs that do not generate a valid parser. Furthermore, there is a lot of redundancy in the language. See Section 6 for further remarks on future improvements of the language.

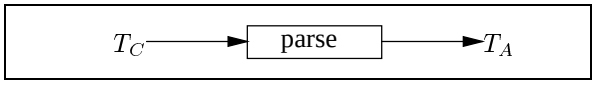

Figure 4: From the viewpoint of a Haskell programmer, the generated parse functions form a parser.

The HASDF-tool

To use the HASDF tool, a user should first compose a HASDF input file. Secondly, this file can be processed by the HASDF tool to produce the following three items: (i) a Haskell module containing the data type that was present in the input file; (ii) a parser; and (iii) an unparser. Thirdly, the generated Haskell module should be imported in the Haskell program. Finally, the parser and unparser can be used by a Haskell program as external functions, or as pre- and postprocessors. Abstract terms coming from the parser and going to the unparser can be constructed and deconstructed with the `show` and `read` functions that are generated by Haskell via the `deriving` construct. For instance, a concrete term `a -> [b]` is parsed to the abstract term `Arrow (Var "a") (List (Var "b"))`. This latter term can be read into Haskell by the `read` function for the type `Type`. The classes `Show` and `Read` may, but do not need to be given explicitly in deriving clauses in the HASDF source; the tool will add them where necessary.

Figure 3 depicts the Haskell module that is generated by applying the HASDF tool to the example of Figure 2. This module should be imported by programs that operate on the data type `Type`.

Some remarks on the implementation

We have used the ASF+SDF Meta-Environment [12] to implement the HASDF tool. The ASF+SDF Meta-Environment is a system for generating programming environments, using SDF for defining the syntax of the language, and ASF for defining the semantics. In principle, any programming language could have been used instead, but the parsing and unparsing facilities of the ASF+SDF system make it an especially powerful tool for this job. We have made use of these facilities in two distinct ways. Firstly, we have defined the HASDF language by a context-free grammar in SDF. From this grammar definition, a parser for HASDF is generated automatically by the ASF+SDF system. Secondly, the parsers and unparsers that we generate are ordinary ASF+SDF programs which implicitly use the GLR parsers generated by the ASF+SDF system. Hence, we implemented generators without explicitly constructing parsers and unparsers ourselves. This double use of the parse and unparse facilities of ASF+SDF has considerably reduced development time.

To explain the setup of our implementation, we first need to explain a few more things about ASF+SDF. ASF+SDF combines concrete programming with term rewriting. So, the user first defines the concrete syntax of his data types and functions in SDF. Secondly he defines rewrite rules in ASF+SDF as conditional equations over this concrete syntax. The term rewrite system thus specified is actually executed by (i) parsing concrete input terms as well as the concrete terms in the equations to some abstract syntax representation, (ii) rewriting the abstract input term to normal form, and (iii) unparsing the abstract normal form to a concrete term.

The implementation of the HASDF tool basically consists of a context-free grammar of HASDF written in SDF, and five generators programmed in ASF+SDF as rewrite equations over terms of this grammar. These five generators generate the following components:

- A Haskell module containing the data type definitions of the abstract grammar.
- An SDF definition of the concrete grammar (by removing the Haskell data types).
- An SDF definition of the abstract grammar (by inspecting the data types).
- Rewrite equations of the following form:
  `parse(concrete-term) = abstract-term`
- Rewrite equations of the following form:
  `unparse(abstract-term) = concrete-term`

The last four together, form an ASF+SDF program which defines a language specific `parse` and `unparse` function for every non-terminal.

From the point of view of a Haskell programmer, the parse functions that are generated by the HASDF tool, form a parser for terms over the concrete syntax specified in a HASDF module (see Figure 4). This parser takes as input a concrete term over the language specified in HASDF (T_C) and constructs an abstract representation (T_A) using Haskell data types. A similar model applies to unparsing.

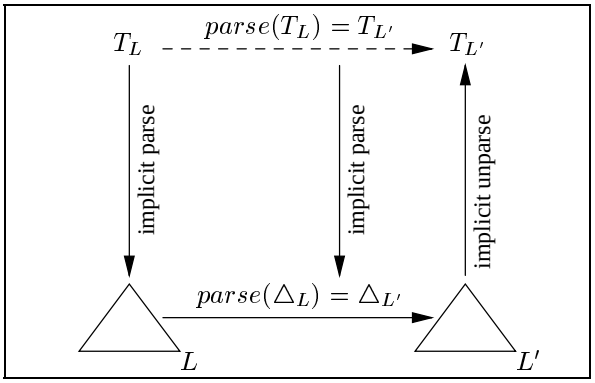

Figure 5: From the viewpoint of ASF+SDF, the parse functions just form a rewrite system.

A HASDF module specifies a mapping from concrete to abstract terms. A HASDF module therefor defines two languages L and L'. The generated ASF+SDF program translates concrete terms over language L to concrete terms over language L' and vice versa. Hence, from the viewpoint of ASF+SDF, the generated program is just a rewrite system. Although input terms and rewrite rules are in ASF+SDF defined over concrete syntax, implicit parsing is performed by the ASF+SDF system before the rewrite system is executed. Likewise, implicit unparsing is performed after rewriting. As a consequence, we were able to implement the parse and unparse functions as ordinary rewrite systems over concrete syntax without parsing and unparsing at all. All parsing and unparsing is handled implicitly by the ASF+SDF system.

Figure 5 shows the parse function from the viewpoint of ASF+SDF. The figure shows that implicit parsing of the concrete input term of language L and of the rewrite system is performed by ASF+SDF. This concrete input term corresponds to the term T_C in Figure 4. The abstract representation of the rewrite system is used to rewrite the abstract representation of the input term. After reduction, the abstract representation of the normal form is unparsed implicitly to obtain its concrete representation. This concrete representation of the normal form is a term over language L', which corresponds to an abstract term T_A from the viewpoint of Haskell (see Figure 4).

6 Conclusions and future work

We will briefly summarize the work presented in this paper and its potential significance for functional programming. Also, we will mention some possible directions for future work.

Contributions

Two concrete contributions were presented:

- HASDF: a formalism for simultaneous definition of the concrete syntax of a language and the Haskell data types that represent its abstract syntax.
- A tool that generates a GLR-parser and an unparser from a HASDF definition.

Significance to functional programming

At the moment, functional programming does not enjoy the popularity its adherents aspire to. Wadler [28] diagnosed this fact to be caused by, among other factors, (i) isolationism (ii) lack of tools (iii) lack of real-world applications. We feel that HASDF might contribute to solving these problems. On the one hand it could open up a vast range of language-centered application areas, and on the other hand it might help conquer functional programming isolationism by allowing Haskell programs to speak many languages.

Future work

- Improve the formalism HASDF. The current version of HASDF was designed to explicitly contain both Haskell data types declarations and SDF productions. This was done to make the formalism accessible to users with Haskell and SDF backgrounds alike. The drawback is that HASDF definitions are needlessly verbose. In a future version of HASDF we plan to offer a more concise syntax.
- Implement support for concrete programming in Haskell. On the basis of the work presented in this paper, support for concrete programming in Haskell could be realized. This would involve creating a preprocessor for Haskell, which transforms Haskell programs that contain concrete syntax into programs that contain abstract syntax only. This would pave the way for integration of Haskell into

the new implementation of the ASF+SDF Meta-Environment [24].

- Improve HASDF's support for modularity. SDF is a modular formalism. Haskell is a modular programming language. To properly exploit these modular features, some enhancements to Haskell and to the HASDF-tool need to be made. SDF has some modularity possibilities that do not map easily onto Haskell. Different grammar productions for the same non-terminal can be distributed over different modules. Extensible data types could be added to Haskell in order to make a seamless mapping of this form of grammar modularity onto Haskell data types possible. These enhancements would improve HASDF's support for aspect-oriented language design.

- Make SDF with GLR-parser generation available for various other programming languages. This can be done by introducing an intermediate language in which two related syntaxes can be defined simultaneously by SDF-productions. This intermediate language would be a generalization of the HASDF-language, and could be called SDF-SDF. Then, the HASDF-tool implementation can without much effort be restructured into a front-end that translates HASDF to SDF-sdf, and a number of back-ends that generate parsers and unparsers from this translation. These front- and back-ends would constitute an instantiation of a general framework for a family of tools. By providing new front-ends and back-ends, this framework can be instantiated to create generators for other programming languages besides Haskell. Also, by providing additional back-ends, the framework can be used to generate other traversal-based components besides parsers and unparsers. For instance: default pretty-printers, default program transformation and program analysis functions, and renamers.

- Investigate the use of SDF as interface definition language. When a family of syntax definition formalisms is developed for a number of programming languages, as we propose, SDF-SDF can be used to connect them. The generated tools can be used to port data between programs written in different languages. This may provide a key to the interoperability of these languages.

Acknowledgments

We would like to thank Arie van Deursen (CWI), Paul Klint (CWI), Leon Moonen (University of Amsterdam), Eric Saaman (University of Groningen), and Mark van den Brand (CWI) for reading earlier drafts of this paper.

A Syntax Definition of SDF

This appendix contains a specification of the grammar of SDF used in HASDF. It is based on the definition described in [8]. It has been adapted to correspond to the syntax that is accepted by the current implementation of the ASF+SDF Meta-Environment. The SDF syntax that is accepted by the ASF+SDF Meta-Environment does not correspond completely to the original SDF definition defined in [8]. The syntax accepted by the ASF+SDF Meta-Environment differs in that it does not support the **module** keyword and top-level syntax sections (i.e., syntax sections that are not embedded in an exports or hiddens section).

A.1 Module Sdf

imports
Literals

exports
sorts
Sdf-Id Iterator CharClass Module Section SyntaxSection LexicalFunction SpecialLexId BasicLexElem LexElem CfFunction BareFunction CfElem Attributes PriorChain FunctionList Sdf-Variable VarSort FunOpName ListOpName ModuleKeyword ModuleToken

lexical-syntax

[A-Z]	→ Sdf-Id
[A-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]	→ Sdf-Id
"\\" ~[]	→ EscChar
"\\" [01] [0-7] [0-7]	→ EscChar
~[\000-\037\-\[\]\\]	→ C-Char
EscChar	→ C-Char
C-Char	→ CharRange
C-Char "-" C-Char	→ CharRange
"[" CharRange* "]"	→ CharClass
[a-z]	→ Literal
[a-z] [A-Za-z0-9\-_]* [A-Za-z0-9]	→ Literal
"%%"	→ ModuleToken

context-free syntax

ModuleKeyword Sdf-Id Section*	→ Module
ModuleToken	→ ModuleKeyword
"imports" Sdf-Id+	→ Section
"exports" SyntaxSection+	→ Section
"hiddens" SyntaxSection+	→ Section
"sorts" Sdf-Id+	→ SyntaxSection
"lexical" "syntax" LexicalFunction+	→ SyntaxSection
"context-free" "syntax" CfFunction+	→ SyntaxSection
"priorities" {PriorChain ","}+	→ SyntaxSection
"variables" Sdf-Variable+	→ SyntaxSection
LexElem+ "→" Sdf-Id	→ LexicalFunction
LexElem+ "→" SpecialLexId	→ LexicalFunction
"LAYOUT"	→ SpecialLexId
Sdf-Id	→ BasicLexElem

Literal → BasicLexElem
CharClass → BasicLexElem
“~” CharClass → BasicLexElem
BasicLexElem Iterator → LexElem
BasicLexElem → LexElem
“+” → Iterator
“*” → Iterator
FunOpName CfElem* “→” Sdf-Id Attributes → CfFunction
Literal “(” {CfElem “,”}* “)” “→” Sdf-Id Attributes → CfFunction
Sdf-Id → CfElem
Literal → CfElem
Sdf-Id Iterator ListOpName → CfElem
“{” Sdf-Id Literal “}” Iterator ListOpName → CfElem
→ Attributes
“{” {Literal “,”}+ “}” → Attributes
Literal “:” → FunOpName
→ FunOpName
“:” Literal → ListOpName
→ ListOpName
FunctionList “>” {FunctionList “>”}+ → PriorChain
FunctionList “<” {FunctionList “<”}+ → PriorChain
“{” Literal “:” {BareFunction “,”}+ “}” → PriorChain
BareFunction → FunctionList
“{” {BareFunction “,”}+ “}” → FunctionList
“{” Literal “:” {BareFunction “,”}+ “}” → FunctionList
CfElem* “→” Sdf-Id → BareFunction
Literal+ → BareFunction
Literal “(” {CfElem “,”}* “)” “→” Sdf-Id → BareFunction
LexElem+ “→” VarSort → Sdf-Variable
Sdf-Id → VarSort
Sdf-Id Iterator → VarSort
“{” Sdf-Id Literal “}” Iterator → VarSort

A.2 Module Layout

exports
lexical-syntax
[␣\t\n] → LAYOUT
“%%” ~[\n]* “\n” → LAYOUT
“%” ~[%\n]+ “%” → LAYOUT

A.3 Module Literals

imports
Layout

exports
sorts
Literal
lexical-syntax
"\\" ~[] → EscChar
"\\" [01] [0-7] [0-7] → EscChar
~[\000-\037"\\] → L-Char
EscChar → L-Char
"\"" L-Char* "\"" → Literal

B Syntax Definition of Haskell

In this appendix we give the definition in SDF of the part of the Haskell grammar that is used in HASDF. This grammar definition is based on the definition in [17]. It has been simplified in several respects. HASDF makes use of two kinds of top declarations only: data type definitions and type synonyms. Therefore, the grammar is restricted to productions that are related to these. The syntax of data type definitions has been simplified, because only data types that can represent abstract syntax trees make sense in the context of HASDF (see Section 5). For instance, type variables and function types are not allowed in the arguments of data type constructors. Furthermore, list types occurring nested in list or tuple types are not supported.

B.1 Module Haskell-Syntax

imports
Haskell-Lexical-Syntax

exports
sorts
Varid Conid Varsym Consym Tyvar Tycon Tycls Modid Qtycon Qtycls HaskellModule Body Topdecl Topdecls Vars Type Btype Atype Gtycon ParenOptGtycon Simpletype Constrs Constr Fielddecl Deriving Dclass Var Con Conop SmallOrLargeOrPrimeOrUnderscore Prime Underscore Colon SymbolOrColon ModidPeriodOpt ModidPeriod DerivingOpt BtypeOrStrictAtype StrictOptAtype StrictOpt StrictAtype TypeOrStrictAtype DclassOrDclassParenCommaStar
lexical-syntax
Small SmallOrLargeOrPrimeOrUnderscore* → Varid
Large SmallOrLargeOrPrimeOrUnderscore* → Conid
Small → SmallOrLargeOrPrimeOrUnderscore
Large → SmallOrLargeOrPrimeOrUnderscore
Prime → SmallOrLargeOrPrimeOrUnderscore
Underscore → SmallOrLargeOrPrimeOrUnderscore
['] → Prime
[_] → Underscore

Symbol SymbolOrColon* → Varsym
[:] SymbolOrColon* → Consym
Symbol → SymbolOrColon
Colon → SymbolOrColon
[:] → Colon
Varid → Tyvar
Conid → Tycon
Conid → Tycls
Conid → Modid
ModidPeriodOpt Tycon → Qtycon
ModidPeriodOpt Tycls → Qtycls
ModidPeriod → ModidPeriodOpt
[]* → ModidPeriodOpt

Modid [.] → ModidPeriod

context-free syntax

“module” Modid “where” Body → HaskellModule
Body → HaskellModule
Topdecls → Body
Topdecl* → Topdecls
type Simpletype “=” Type → Topdecl
data Simpletype “=” Constrs DerivingOpt → Topdecl
Deriving → DerivingOpt
→ DerivingOpt
{Var “,”}+ → Vars
Btype → Type
Atype → Btype
Gtycon → Atype
“(” Type “,” {Type “,”}+ “)” → Atype
“(” Type “)” → Atype
“[” ParenOptGtycon “]” → Atype
Qtycon → Gtycon
“(” “)” → Gtycon
“(” ParenOptGtycon “)” → ParenOptGtycon
Gtycon → ParenOptGtycon
Tycon → Simpletype
{Constr “|”}+ → Constrs
Con StrictOptAtype* → Constr
BtypeOrStrictAtype Conop BtypeOrStrictAtype → Constr
Con “{” {Fielddecl “,”}+ “}” → Constr
StrictOpt Atype → StrictOptAtype
“!” → StrictOpt
→ StrictOpt
Btype → BtypeOrStrictAtype
StrictAtype → BtypeOrStrictAtype

“!” Atype	→ StrictAtype
Vars “::” TypeOrStrictAtype	→ Fielddecl
Type	→ TypeOrStrictAtype
StrictAtype	→ TypeOrStrictAtype
deriving DclassOrDclassParenCommaStar	→ Deriving
Dclass	→ DclassOrDclassParenCommaStar
“(” {Dclass “,”}* “)”	→ DclassOrDclassParenCommaStar
Qtycls	→ Dclass
Varid	→ Var
“(” Varsym “)”	→ Var
Conid	→ Con
“(” Consym “)”	→ Con
Consym	→ Conop
“‘” Conid “‘”	→ Conop

B.2 Module Haskell-Lexical-Syntax

imports
Haskell-Layout

exports
sorts
Small ASCsmall Large ASClarge Symbol ASCsymbol Digit
lexical-syntax

ASCsmall	→ Small
[a-z]	→ ASCsmall
ASClarge	→ Large
[A-Z]	→ ASClarge
ASCsymbol	→ Symbol
[!#$%&*+./<=>?@\\^|\-~]	→ ASCsymbol
[0-9]	→ Digit

B.3 Module Haskell-Layout

imports
Layout

exports
lexical-syntax

“--” ~[>\n] ~[\n]* [\n]	→ LAYOUT
“--” [\n]	→ LAYOUT
~[\-]	→ H-CChar
“-” ~[}]	→ H-CChar
“{-” H-CChar* “-}”	→ LAYOUT

C Syntax Definition of HASDF

This appendix contains the syntax definition of HASDF. This syntax definition combines Haskell with SDF. The HASDF syntax definition introduces a modified construct for Haskell data type definitions and a new syntax section. The data type definition of HASDF allows its constructors to be ordinary Haskell constructors or constructors annotated with concrete syntax. The new syntax section, recognized by the keyword **datatypes**, enables the definition of Haskell data types in HASDF.

imports
Sdf Haskell-Syntax

exports
sorts
H-SDF-Sep HASDF-Topdecl HASDF-Constr HASDF-Constrs HASDF-Module HASDF-Section HASDF-SyntaxSection
context-free syntax

Constr	→ HASDF-Constr
Constr H-SDF-Sep CfFunction	→ HASDF-Constr
"⋈"	→ H-SDF-Sep
{HASDF-Constr "|"}+	→ HASDF-Constrs
"datatypes" HASDF-Topdecl+	→ HASDF-SyntaxSection
data Simpletype "=" HASDF-Constrs DerivingOpt	→ HASDF-Topdecl
"module" Sdf-Id HASDF-Section*	→ HASDF-Module
"module" Sdf-Id HASDF-SyntaxSection+	→ HASDF-Module
"imports" Sdf-Id+	→ HASDF-Section
"exports" HASDF-SyntaxSection+	→ HASDF-Section
"hiddens" HASDF-SyntaxSection+	→ HASDF-Section
SyntaxSection	→ HASDF-SyntaxSection

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[3] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing, 1996.

[4] Arie van Deursen and Paul Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] Jeroen Fokker. Functional parsers. In *First International Summer School on Advanced Functional Programming Techniques*, 1995. LNCS 925.

[7] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques. A Practical Guide*. Ellis Horwood, 1990.

[8] 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. Latest version: ftp://ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z.

[9] Steve Hill. Combinators for parsing expressions. *Journal of Functional Programming*, 6(3):445–463, 1996.

[10] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[11] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[12] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[13] Norbert Klose. Lucky manual, version 0.3, 1997. Available from http://www.ki.informatik.uni-frankfurt.de/~klose/lucky.

[14] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[15] Simon Marlow. Happy user guide, 1997. Available from http://www.dcs.gla.ac.uk/fp/software/happy/happy.ps.

[16] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, July 1995.

[17] John Peterson and Kevin Hammond (editors). Report on the programming language Haskell. a non-strict, purely functional language. version 1.4, April 1997. Available from http://www.haskell.org/definition.

[18] David R. Rarditi and Andrew W. Appel. ML-Yacc user's manual, version 2.3. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Yacc/manual.html, 1994.

[19] Jan Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[20] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Second International Summer School on Advanced Functional Programming Techniques*, 1996. LNCS 1126.

[21] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[22] M. G. J. van den Brand and M. de Jonge. Pretty-Printing within the ASF+SDF Meta-Environment: a Generic Approach. In preparation, 1998.

[23] M. G. J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[24] M. G. J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new ASF+SDF Meta-Environment. In A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer Verlag, 1997. Available from ftp://ftp.cwi.nl/pub/kuipers/papers/asfsdf97.pdf.

[25] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[26] M. G. J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[27] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[28] Philip Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 33(8):23–27, 1998.