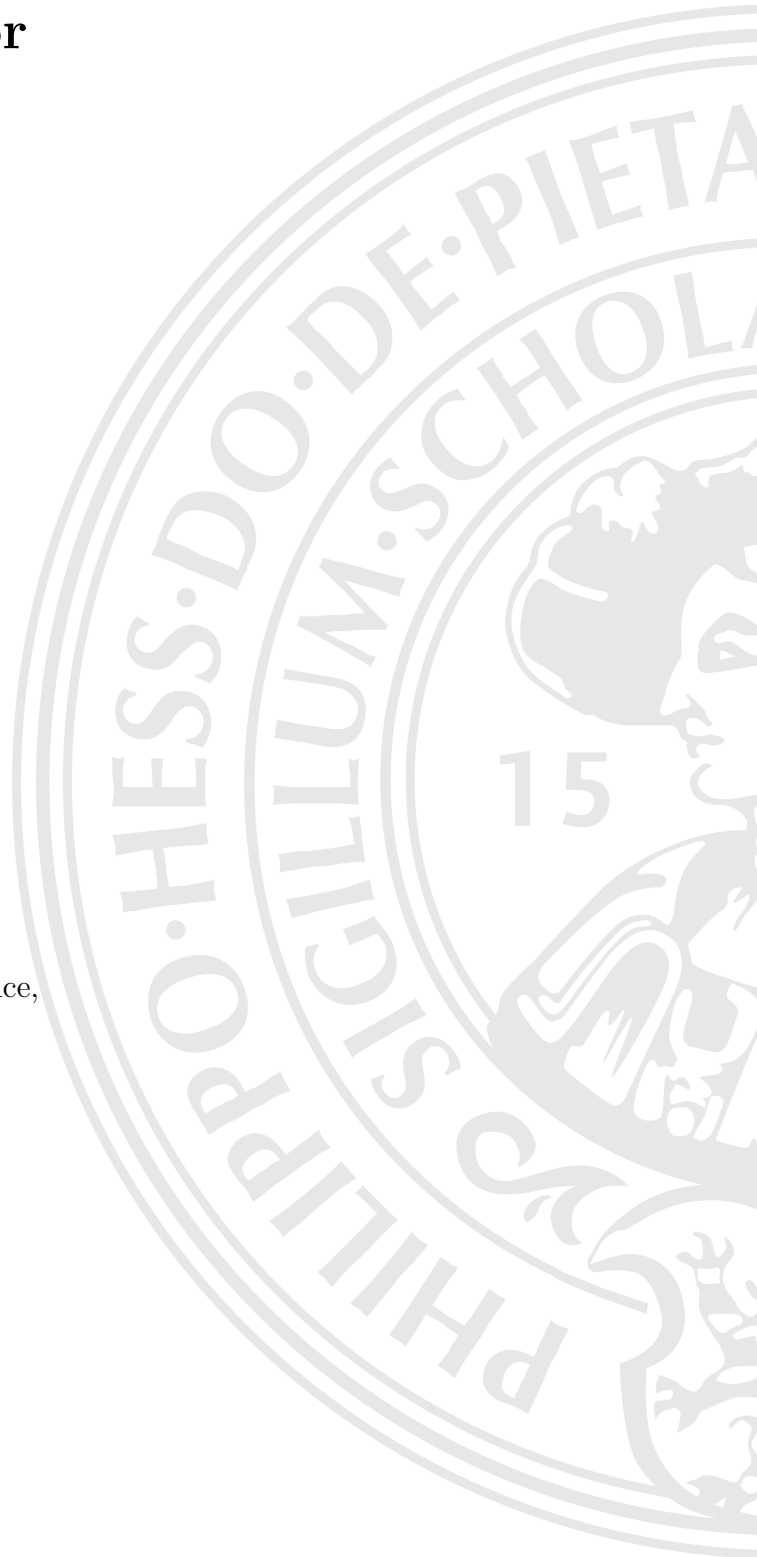
Extensible Languages for Flexible and Principled Domain Abstraction



Dissertation
for the degree of
Doctor of Natural Sciences

Submitted by
Sebastian Thore Erdweg, M.Sc.,
born March 14, 1985 in Frankfurt/Main

Department of Mathematics and Computer Science,
Philipps-Universität Marburg

Referees:
Prof. Dr. Klaus Ostermann
Dr. Eelco Visser
Prof. Dr. Ralf Lämmel

Submitted November 28, 2012.
Defended March 06, 2013.

Marburg, 2013.

Abstract

Most programming languages are designed for general-purpose software development in a one-size-fits-all fashion: They provide the same set of language features and constructs for all possible applications programmers ever may want to develop. As with shoes, the one-size-fits-all solution grants a good fit to few applications only.

The trend toward domain-specific languages, model-driven development, and language-oriented programming counters general-purpose languages by promoting the use of *domain abstractions* that facilitate domain-specific language features and constructs tailored to certain application domains. In particular, domain abstraction avoids the need for encoding domain concepts with general-purpose language features and thus allows programmers to program at the same abstraction level as they think.

Unfortunately, current approaches to domain abstraction cannot deliver on the promises of domain abstraction. On the one hand, approaches that target internal domain-specific languages lack *flexibility* regarding the syntax, static checking, and tool support of domain abstractions, which limits the level of actually achieved domain abstraction. On the other hand, approaches that target external domain-specific languages lack important *principles*, such as modular reasoning and composition of domain abstractions, which inhibits the applicability of these approaches in the development of larger software systems. In this thesis, we pursue a novel approach that unifies the advantages of internal and external domain-specific languages to support flexible and principled domain abstraction.

We propose *library-based extensible programming languages* as a basis for domain abstraction. In an extensible language, domain abstraction can be realized by extending the language with domain-specific syntax, static analysis, and tool support. This enables domain abstractions as flexible as external domain-specific languages. To ensure the compliance with important software-development principles, we organize language extensions as libraries and use simple import statements to activate extensions. This facilitates modular reasoning (by inspecting import statements), supports the composition of domain abstractions (by importing multiple extensions), and allows uniform self-application of language extensions in the development of further extensions (by importing extensions in an extension definition). A library-based organization of extensions enables domain abstractions as principled as internal domain-specific languages.

We designed and implemented *SugarJ*, a library-based extensible programming language on top of Java. SugarJ libraries can declare and export extensions of SugarJ's syntax, static analysis, and editor support. Thereby, a syntactic extension consists of an extended syntax and a desugaring transformation from the extended syntax into SugarJ base syntax, an analysis extension matches on part of the current file's abstract syntax tree

and produces a list of errors, and an editor extension declares editor services such as coloring or code completion for certain language constructs. SugarJ extensions are fully self-applicable: An extended syntax can desugar into the declaration of another extensions, an extended analysis can check the declaration of an extension, and an extended editor can assist developers in writing extensions. To process a source file with extensions, the SugarJ compiler and IDE inspect the imported libraries to determine active extensions. The compiler and IDE adapt the parser, code generator, analyzer, and editor of the source file according to the active extensions.

In this thesis, we do not only describe the design and implementation of SugarJ, but also report on extensions of the original design. In particular, we designed and implemented a generalization of the SugarJ compiler that supports alternative base languages besides Java. Using this generalization, we developed the library-based extensible programming languages SugarHaskell, SugarProlog, and SugarFomega. Furthermore, we developed an extension of SugarJ that supports polymorphic domain abstraction and ensures communication integrity. Polymorphic domain abstraction enables programmers to provide multiple desugarings for the same domain-specific syntax. This increases the flexibility of SugarJ and supports scenarios known from model-driven development. Communication integrity specifies that components of a software system may communicate over explicit channels only. This is interesting in the context of code generation where it effectively prohibits the generation of implicit module dependencies. We augmented SugarJ's principles by enforcing communication integrity.

On the basis of SugarJ and numerous case studies, we argue that flexible and principled domain abstraction constitutes a scalable programming model for the development of complex software systems.

Zusammenfassung

Die meisten Programmiersprachen werden als Universalsprachen entworfen. Unabhängig von der zu entwickelnden Anwendung, stellen sie die gleichen Sprachfeatures und Sprachkonstrukte zur Verfügung. Solch universelle Sprachfeatures ignorieren jedoch die spezifischen Anforderungen, die viele Softwareprojekte mit sich bringen.

Als Gegenkraft zu Universalsprachen fördern domänenspezifische Programmiersprachen, modellgetriebene Softwareentwicklung und sprachorientierte Programmierung die Verwendung von *Domänenabstraktion*, welche den Einsatz von domänenspezifischen Sprachfeatures und Sprachkonstrukten ermöglicht. Insbesondere erlaubt Domänenabstraktion Programmieren auf dem selben Abstraktionsniveau zu programmieren wie zu denken und vermeidet dadurch die Notwendigkeit Domänenkonzepte mit universalsprachlichen Features zu kodieren.

Leider ermöglichen aktuelle Ansätze zur Domänenabstraktion nicht die Entfaltung ihres ganzen Potentials. Einerseits mangelt es den Ansätzen für interne domänenspezifische Sprachen an *Flexibilität* bezüglich der Syntax, statischer Analysen, und Werkzeugunterstützung, was das tatsächlich erreichte Abstraktionsniveau beschränkt. Andererseits mangelt es den Ansätzen für externe domänenspezifische Sprachen an wichtigen *Prinzipien*, wie beispielsweise modularem Schließen oder Komposition von Domänenabstraktionen, was die Anwendbarkeit dieser Ansätze in der Entwicklung größerer Softwaresysteme einschränkt. Wir verfolgen in der vorliegenden Doktorarbeit einen neuartigen Ansatz, welcher die Vorteile von internen und externen domänenspezifischen Sprachen vereint um flexible und prinzipientreue Domänenabstraktion zu unterstützen.

Wir schlagen *bibliotheksbasierte erweiterbare Programmiersprachen* als Grundlage für Domänenabstraktion vor. In einer erweiterbaren Sprache kann Domänenabstraktion durch die Erweiterung der Sprache mit domänenspezifischer Syntax, statischer Analyse, und Werkzeugunterstützung erreicht werden . Dies ermöglicht Domänenabstraktionen die selbe Flexibilität wie externe domänenspezifische Sprachen. Um die Einhaltung üblicher Prinzipien zu gewährleisten, organisieren wir Spracherweiterungen als Bibliotheken und verwenden einfache Import-Anweisungen zur Aktivierung von Erweiterungen. Dies erlaubt modulares Schließen (durch die Inspektion der Import-Anweisungen), unterstützt die Komposition von Domänenabstraktionen (durch das Importieren mehrerer Erweiterungen), und ermöglicht die uniforme Selbstanwendbarkeit von Spracherweiterungen in der Entwicklung zukünftiger Erweiterungen (durch das Importieren von Erweiterungen in einer Erweiterungsdefinition). Die Organisation von Erweiterungen in Form von Bibliotheken ermöglicht Domänenabstraktionen die selbe Prinzipientreue wie interne domänenspezifische Sprachen.

Wir haben die bibliotheksbasierte erweiterbare Programmiersprache SugarJ entworfen und implementiert. SugarJ Bibliotheken können Erweiterungen der Syntax, der statischen Analyse, und der Werkzeugunterstützung von SugarJ deklarieren. Eine syntaktische Erweiterung besteht dabei aus einer erweiterten Syntax und einer Transformation der erweiterten Syntax in die Basissyntax von SugarJ. Eine Erweiterung der Analyse testet Teile des abstrakten Syntaxbaums der aktuellen Datei und produziert eine Liste von Fehlern. Eine Erweiterung der Werkzeugunterstützung deklariert Dienste wie Syntaxfärbung oder Codevervollständigung für bestimmte Sprachkonstrukte. SugarJ Erweiterungen sind vollkommen selbstanwendbar: Eine erweiterte Syntax kann in eine Erweiterungsdefinition transformiert werden, eine erweiterte Analyse kann Erweiterungsdefinitionen testen, und eine erweiterte Werkzeugunterstützung kann Entwicklern beim Definieren von Erweiterungen assistieren. Um eine Quelldatei mit Erweiterungen zu verarbeiten, inspizieren der SugarJ Compiler und die SugarJ IDE die importierten Bibliotheken um die aktiven Erweiterungen zu bestimmen. Der Compiler und die IDE adaptieren den Parser, den Codegenerator, die Analyseroutine und die Werkzeugunterstützung der Quelldatei entsprechend der aktiven Erweiterungen.

Wir beschreiben in der vorliegenden Doktorarbeit nicht nur das Design und die Implementierung von SugarJ, sondern berichten darüber hinaus über Erweiterungen unseres ursprünglich Designs. Insbesondere haben wir eine Generalisierung des SugarJ Compilers entworfen und implementiert, die neben Java alternative Basissprachen unterstützt. Wir haben diese Generalisierung verwendet um die bibliotheksbasierten erweiterbaren Programmiersprachen SugarHaskell, SugarProlog, und SugarFomega zu entwickeln. Weiterhin haben wir SugarJ ergänzt um polymorphe Domänenabstraktion und Kommunikationsintegrität zu unterstützen. Polymorphe Domänenabstraktion ermöglicht Programmierern mehrere Transformationen für die selbe domänenspezifische Syntax bereitzustellen. Dies erhöht die Flexibilität von SugarJ und unterstützt bekannte Szenarien aus der modellgetriebenen Entwicklung. Kommunikationsintegrität spezifiziert, dass die Komponenten eines Softwaresystems nur über explizite Kanäle kommunizieren dürfen. Im Kontext von Codegenerierung stellt dies eine interessante Eigenschaft dar, welche die Generierung von impliziten Modulabhängigkeiten untersagt. Wir haben Kommunikationsintegrität als weiteres Prinzip zu SugarJ hinzugefügt.

Basierend auf SugarJ und zahlreicher Fallstudien argumentieren wir, dass flexible und prinzipientreue Domänenabstraktion ein skalierbares Programmiermodell für die Entwicklung komplexer Softwaresysteme darstellt.

Acknowledgements

This thesis would not have been possible without the shoulders of many giants that I was allowed to stand on.

First of all, I would like to thank my advisor Klaus Ostermann for his persistence in convincing me to start a PhD in the first place. Since I joined Klaus at Aarhus University four years ago, he taught me many things about research, from reading and discussing scientific articles, to writing and reviewing scientific articles myself. I am most grateful, though, for Klaus's unconditional support in following my own ideas, which eventually led to SugarJ and this thesis.

Tillmann Rendel was a constant source of inspiration and his unconventional thinking was the basis of innumerable interesting and insightful discussions about programming languages, the universe, and everything. Tillmann took an active role in the design of SugarJ, challenged my ideas early on, and contributed many ideas himself to the project described in this thesis. I would like to thank Tillmann for his support and productive collaboration.

Christian Kästner joined our research group in 2010 and has supported me ever since. Christian offered invaluable feedback on ideas, paper drafts, and, even after leaving our group three months ago, on this thesis. Christian was a reliable source of advice that provided guidance during my PhD. Christian's dedication to support others is inspiring.

I am very grateful to all colleagues and students I was allowed to discuss with, collaborate with, and learn from in the past years: Michael Achenbach, Yufei Cai, Yi Dai, Olivier Danvy, Stefan Fehrenbach, Paolo Giarrusso, Katharina Haselhorst, Christian Hofer, Lennart Kats, Karl Klose, Jonas Pusch, Felix Rieger, Thomas Thüm, and Eelco Visser. Finally, I would like to thank my family and friends, but especially my partner Katharina, for their continuous support far beyond my work.

Contents

1 Introduction

The complexity of modern software systems calls for new forms of abstraction. Modern software systems have to address concerns from different domains and technical spaces. However, conventional abstraction mechanism mostly focus on the run-time behavior of programs and cannot sufficiently support multiple domains, which come with their own notation, invariants, and tool support. Therefore, new forms of abstraction are required that support user-defined syntax, invariant validation, and tool support.

In general, an abstraction hides low-level implementation details and introduces new high-level concepts for programmers. Common abstractions include

- symbolic variables to abstract from memory addresses,
- control structures such as loops to abstract from *goto* statements,
- object-oriented programming to abstract from individual code blocks by managing classes of blocks and their instances,
- garbage collection to abstract from manual memory management, and
- multithreading to abstract from sequential and finitely parallel computation.

These and other forms of abstraction are part of many high-level programming languages, such as Java, C#, Scala, OCaml, and Haskell.

Programmers demand new forms of abstraction due to a perceived lack of high-level language constructs or due to perceived trouble with existing language constructs. Both scenarios frequently occurred in the history of programming languages.

For example, Dijkstra argues that *goto* statements complicate program understanding, because the dynamic control flow does not align well with the lexical structure of the program text [Dij68]. Dijkstra concludes that to resolve this problem, more restrictive control structures such as procedures or *while* loops should be used, because they entail a unique and simple relation between dynamic control flow and code. Thus, Dijkstra argues for new abstractions on the basis of troublesome existing language constructs. Conversely, Dahl, Myhrhaug, and Nygaard motivate the design of SIMULA with the lack of domain-specific language features for the domain of large discrete-event simulations [DMN67]. Driven by this demand, they propose classes, objects, and inheritance for decomposing large applications into interacting classes of code blocks. As we know now, these features turned out to be useful in a wider area of application than originally anticipated.

More generally, it is not possible to anticipate all scenarios in which programmers may want to apply a programming language [LZ74]. In some applications the included

language features will impose a laborious programming style, in other applications more high-level language features will be desired to address the problem at hand more directly. Therefore, it is not enough for a programming language to include built-in abstractions. Instead, to promote the expressiveness of programmers, a programming language should enable programmers to introduce new forms of application-specific abstractions.

Throughout the history of programming languages, it has been a research goal to discover programming-language concepts that enable user-defined abstractions. For example, procedures and higher-order functions enable abstraction from repeating patterns in a program [BBG+63, FFFK01], abstract data types support user-defined data representations with encapsulation [LZ74], and object-oriented programming facilitates the definition of stateful, interacting components by the programmer [DMN67]. However, most existing abstraction mechanisms only support semantic abstraction, but neglect the need for integrating user-defined abstractions into the syntax, static analysis, and editor of a programming language. This limits the usability of user-defined abstractions because users are bound to the language's original syntax, static analysis, and editor support, and, conversely, they are oblivious to the user-defined abstractions.

In particular, today's abstraction mechanisms provide insufficient support for the development of software systems that simultaneously have to deal with a multitude of domains and technical spaces, such as network communication, persistency management, visualization, and data analysis. For example, the Eclipse platform provides an update mechanism (network), stores source and configurations files (persistency), provides an interactive editor (visualization), and supports source-code queries (analysis). Existing abstraction mechanisms impose the same syntax, invariants, and tool support on all code of the project, irrespective of the domain that the code addresses. This precludes abstraction potential. In particular, a better domain-specific syntactic integration can circumvent syntactic boilerplate, domain-specific static analyses can enforce application-specific invariants to reduce the number of potential runtime errors and provide more rapid feedback to developers, and domain-specific editor support can improve the understandability and modifiability of source code. For these reasons, abstraction mechanisms should support user-defined syntax, static analyses, and editor support.

This problem can also be motivated from the perspective of domain-specific languages. A domain-specific language (DSL) consists of a collection of user-defined abstractions that are specifically useful for a particular domain [Ben86, Fow10, MHS05]. Often a DSL is useful in multiple applications. For example, regular expressions, SQL, statemachines, and XML are widely adopted DSLs. However, the language-oriented-programming paradigm suggests that the definition of a DSL can be beneficial even if it is used in a single application only [Dmi04, Fow05b, War95]. DSLs are typically classified as either *external* or *internal* [Fow05b], which largely influences their applicability and provides a good starting point for our discussion of DSLs.

An external DSL is an independent programming language. Due to their independence, external DSLs are very flexible regarding their syntax, static analysis, semantics, and

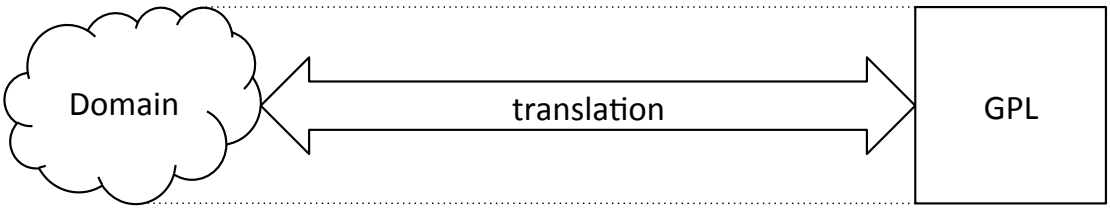
editor support. However, this flexibility inhibits interoperability between programs written in different external DSLs: There is no common ground for composing external DSLs because each DSL has its own parser, analyzer, code generator or interpreter, and editor. However, the composition of DSLs is essential, since DSLs focus on a single domain and thus are incomplete by design; in realistic software projects, the application of a single DSL is insufficient [PRBA10, WHG+09]. Moreover, general-purpose functionality such as a collections API needs to be reimplemented for each external DSL, which raises the development cost of external DSLs [Hud98]. These drawbacks are significant and justify the investigation of abstraction mechanisms that enable the integration of DSLs into existing programming languages.

Internal DSLs employ the existing abstraction mechanisms of a programming language (called the host language) to encode domain abstractions. For example, in an object-oriented host language, domain abstractions can be represented as classes and methods. An internal DSL merely provides a domain-specific view and decomposition principle on an otherwise regular host-language program. The reuse of host-language abstractions has three central advantages that result from the fact that a program written in an internal DSL also is a host-language program. First, programs of an internal DSLs adhere to the principles of the host language, such as modular reasoning, well-defined variable scoping, abstraction mechanisms for code reuse, and type-system guarantees. Second, programs written in different internal DSLs can interoperate with each other using the standard schemes of interaction from the host language. Third, programs of an internal DSL can directly reuse any general-purpose functionality present in the host language, such as the collections API. Unfortunately, as consequence of the reuse of the host language's abstraction mechanisms, internal DSLs inherit the deficiencies of these abstraction mechanisms as well. In particular, existing abstraction mechanisms fail to provide good support for the integration of domain-specific syntax, domain-specific analyses, and domain-specific editor support.

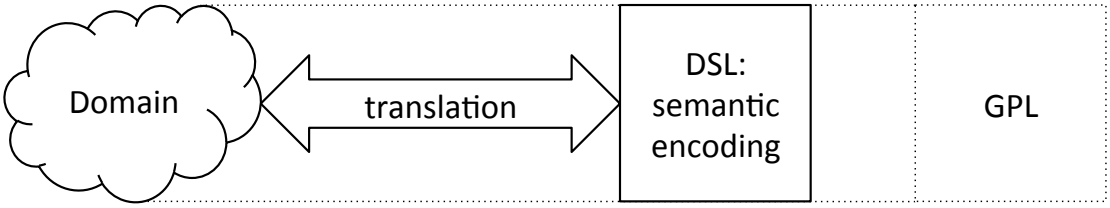
It is our goal to investigate abstraction mechanisms for domain abstraction as flexible as external DSLs and as principled as internal DSLs. In the remainder of this chapter, we present our design goals in detail and outline our solution, which is based on extensible languages. We dedicate the rest of this thesis to demonstrating that *extensible languages enable flexible and principled domain abstraction.*

1.1 Flexible domain abstraction

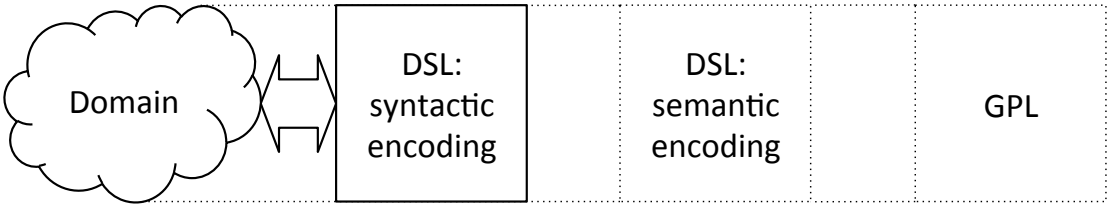
The goal of domain abstraction is to bridge the representational gap, that is, "the gap between our mental model of the domain and its representation in software" [Lar02]. A better representation of domain concepts enables programmers to map domain knowledge into source code and vice versa, which simplifies the creation, comprehension, and maintenance of domain-specific programs. We illustrate this idea in Figure 1.1.



(a) Representational gap between domain concepts and code written in general-purpose language (GPL).



(b) A semantic encoding of domain concepts as a DSL narrows the gap.



(c) Domain-specific syntax narrows the representational gap even more.

Figure 1.1: Domain abstraction narrows the representational gap.

Domain-specific semantics. Without domain abstraction (Figure 1.1(a)), programmers need to translate their understanding of domain concepts into a general-purpose programming language. For example, suppose a software developer needs to implement a parser in Java. The developer has already designed the grammar that the parser should accept, using parser-specific concepts such as terminal, nonterminal, and production. In Java, there is no corresponding representation of these domain concepts. Therefore, the developer needs to encode the grammar with concepts that already exist in the Java language, such as input streams and *switch-case* statements. Since the resulting code does not resemble the grammar, it is difficult to develop the initial parser or to maintain the parser when the grammar evolves.

With domain abstraction, programmers can express domain concepts in the corresponding DSL, instead of translating domain concepts into a general-purpose programming

language. This narrows the representational gap between domain concepts and their realization as illustrated in Figure 1.1(b). For example, to implement a parser, a programmer can use an internal DSL such as parsec [LM01]. Parsec represents nonterminals as variables of the host language and productions as assignments to these variables. The syntax-definition part of a production is represented with parser combinators that describe sequences, alternatives, and repetition of terminals and nonterminals. Since parsec provides a semantic encoding for each domain concept, it is easy to translate a grammar into a parsec program. Accordingly, we define our first design goal for flexible domain abstraction.

> *Domain-specific semantics*: A domain abstraction should provide a semantic encoding of each domain concept.

Polymorphic domain abstraction. A semantic encoding does not only provide a representation of domain concepts, but also defines how a domain-specific program can be executed. However, often multiple execution strategies are possible for a single language construct. For example, if the domain-specific semantics is given by a code generator, it can generate code of different languages, produce documentation or a pretty print, apply different optimizations, or simply impose different meanings on a domain concept. Flexible domain abstraction should not preclude different semantics. Instead, we postulate that domain abstraction is polymorphic, as is typically the case in model-driven development frameworks. Polymorphic domain abstraction represents our second design goal for flexible domain abstraction.

> *Polymorphic domain abstraction*: Domain abstractions should allow multiple coexisting semantics for domain concepts.

Domain-specific syntax. A semantic encoding is not sufficient. While it provides a way of representing domain concepts in a program, the representation is often inflated or unnatural. As illustrated in Figure 1.1(c), a better syntactic representation can further narrow the gap between domain concepts and there realization. For example, EBNF is a standard notation for representing grammars. Domain experts can easily understand and define EBNF grammars. The following code shows an EBNF production for parsing a lambda expression:

```
exp ::= "lambda" var "." exp  {Lambda}
```

The identifier in curly braces denotes the name of the production. For comparison, here is the same production using parsec in Haskell:

```
exp = do
  string "lambda"
  v <- var
```

```
string "."
e <- exp
return (Lambda v e)
```

Even though the production and nonterminal domain concepts are semantically represented, their textual representation is not natural for domain experts. Moreover, from the perspective of a domain expert, the parsec representation includes complicated boilerplate code such as Haskell's *do* notation, the string combinator, and the manual denotation of the standard abstract syntax tree. Therefore, we state as third design goal for flexible domain abstraction:

> *Domain-specific syntax*: A domain abstraction should provide a natural and concise syntactic encoding of domain concepts.

We should emphasize that domain-specific syntax is an important issue for the usability of domain abstractions. If a domain has a well-known notation (such as EBNF or XML), supporting this notation can shorten the familiarization phase for domain experts. Furthermore, the avoidance of syntactic boilerplate can have a significant impact on the productivity of programmers. After all, it is the syntax of a programming language that programmers have to cope with in their everyday work. In fact, empirical studies confirm that external DSLs can be beneficial in the creation, comprehension, and maintenance of software [HPvD09, KMC12, KOM+10, vDK98].

This indicates that, by narrowing the representational gap, domain abstractions can reduce the artificial complexity of writing programs. However, domain abstraction cannot eliminate the essential complexity of the problem at hand—domain abstraction is no silver bullet [Bro87]. While domain-specific semantics and syntax enable programmers to focus on the essentials of a program, the intrinsic complexity of the domain is present nonetheless. Therefore, it is desirable for a domain abstraction to assist programmers beyond syntax and semantics in tackling the intrinsic complexity of the domain.

Domain-specific static analysis. Mainstream programming languages often provide assistance in the form of static analyses or type checking. A static analysis rejects a program based on a violation of some domain-specific invariant. For example, for the parsing domain, a static analysis could inform the programmer about the presence of a left-recursive production in the grammar. In a parser framework like parsec, which does not support left-recursive grammars, such a domain-specific analysis can prevent run-time errors that otherwise might occur after deployment. In case a static analysis detects a violation of a domain-specific invariant, it can provide valuable domain-specific feedback to the programmer. Therefore, static analysis forms our fourth design goal for flexible domain abstraction.

> *Domain-specific static analysis*: A domain abstraction should be accompanied by static analyses that validate the invariants of the domain.

Domain-specific editor services. Integrated development environments (IDEs) can nicely present the result of a static analysis to the programmer by decorating part of the source code. Furthermore, IDEs offer editor services such as syntax coloring, content completion, or reference resolving to assist the programmer in reading, navigating, writing, and adapting code. For example, editor services for EBNF can apply a different coloring for terminals and nonterminals, propose existing nonterminal names as code completion, and resolve nonterminal references to their definition site. Such editor services can significantly improve the productivity of programmers [RCM04, HW09]. Therefore, for domain abstraction, we require the same level of tool support that mainstream programming languages enjoy. This constitutes our fifth design goal for flexible domain abstraction.

> *Domain-specific editor services*: A domain abstraction should be supplemented by editor services to support programmers.

Summary. We have defined five design goals for flexible domain abstraction: domain-specific semantics, polymorphism, domain-specific syntax, domain-specific static analyses, and domain-specific editor support. However, to enable programmers to make efficient use of such flexible domain abstractions, they should also follow important programming principles, as discussed in the subsequent section.

1.2 Principled domain abstraction

Flexible domain abstraction can be achieved using unprincipled approaches such as preprocessors and build scripts. In this section, we discuss principles that are important for the efficient application of flexible domain abstraction in complex software systems.

Modular reasoning. First of all, a domain abstraction should not inhibit a programmer's ability to modularly reason about a program. It should be possible for a programmer to understand a given source file by only looking at the source file and its dependencies. This entails that all dependencies of the source file must be explicit and no global reasoning is used. For example, build scripts often inhibit modular reasoning because they describe the global architecture of a software project by linking source artifacts and injecting dependencies between them. Since these dependencies are not visible in the source code, programmers must first understand the global build script to reason about a single source artifact and its dependencies. Such lack of modular reasoning significantly constrains the applicability of domain abstraction for larger software systems. Therefore, we formulate the first design goal for principled domain abstraction.

> *Modular reasoning*: Domain abstractions should permit modular program understanding.

Referential transparency. Modular reasoning is an important precondition for program understanding of large applications. However, in the context of domain abstractions, another important criterion for program understanding is referential transparency, which postulates that all variable references are resolved in the lexical context in which they occur [CR91]. For programmers this is crucial because it allows them reason about the identity and meaning of variable names they defined. Regular programming languages (without domain abstraction) ensure referential transparency through lexical scoping. Languages with domain abstraction require additional checks, because domain abstractions are typically implemented by interpreters or code generators that have full control over variable resolution. Referential transparency has been thoroughly studied in the context of syntactic macros [CR91, DHB92, KFFD86], but it is relevant for all forms of abstraction. Thus, we define our second design goal for principled domain abstraction.

Referential transparency: Domain abstractions should be referentially transparent.

Declarativity. Program understanding is not only important for users of domain abstractions. but for implementors of domain abstractions as well. Since domain abstractions are specific to an application or domain, the design and implementation of a domain abstraction must be conducted by some of the potential users. However, the implementation of a new domain-specific language or domain-specific language feature can be complicated, requiring the definition of syntax, semantics, static analysis, and tool support. Moreover, domain abstractions may evolve when the application domain shifts or broadens. To simplify the introduction and maintenance of domain abstractions, their implementation should be *declarative*. For example, EBNF-like languages provide declarative means for the definition of syntax, which avoids the technical details of lexical analysis. Declarative means for the definition of domain abstractions are important to lower the cost of their development and maintenance, and thus make domain abstractions an attractive alternative to traditional software development [Hud98]. Accordingly, we define as our third design goal for principled domain abstraction:

Declarativity: The implementation of domain abstractions should be declarative.

Implementation reuse. To further reduce the cost of developing and maintaining domain abstractions, it should be possible to reuse their implementation. For example, many DSLs contain an expression language for arithmetic and Boolean operations. Requiring developers of domain abstractions to reimplement such expression language for each DSL places an unnecessary burden on them. Instead, implementations of domain abstractions should be organized in a module system that enables the reuse of (part of) the syntax, semantics, static analysis, or tool support of a domain abstraction. This constitutes our fourth design goal for principled domain abstraction.

Implementation reuse: The implementation of domain abstractions should be reusable.

Composability. The previous design goal demands reusability of the implementation. But a domain abstraction itself should also be reusable in different contexts, even if other domain abstractions are needed as well. This requires support for the composition of domain abstractions, which has been the subject of research on *language-oriented programming* for some time [Dmi04, Fow05b, War95]. Language-oriented programming suggests that each component of software project should be implemented in the DSL that matches the component's domain. Since many components interact with multiple domains, the corresponding DSLs must be composable. For example, consider a component that uses HTTP with SSL to transmit a request encoded as an XML document. Even if we have domain abstractions for HTTP, SSL, and XML in separation, our example component requires support for composing these domain abstractions. Accordingly, we define our fifth design goal for principled domain abstraction.

> *Composability*: Domain abstractions should be composable such that clients can use concepts from multiple domains simultaneously.

Uniformity. So far, our discussion focused on domain abstractions for writing application code; only declarativity addresses the implementation of domain abstractions. However, our declarativity design goal is generic and does not address the specific needs of building domain abstractions for certain domains. For example, when building different DSL that are dialects of XML, a domain abstraction for implementing these domain abstractions could introduce XML Schema, which provides a domain-specific mechanism for declaring XML dialects. This requires a *uniform* language design where domain abstractions are self-applicable. As macro systems like Scheme [SDF+09] and Racket [Fla12] demonstrate, such uniform language design enables "growing a language" [Ste99] from a small core language into a full-fledged general-purpose language that can extend itself. We adopt uniformity as our final design goal for principled domain abstraction.

> *Uniformity*: Domain abstractions should be applicable in the implementation of other domain abstractions.

Summary. Domain abstraction should adhere to established programming principles. In particular, domain abstraction should permit modular reasoning and referential transparency, support composability and uniformity, and their implementation should be declarative and reusable. We believe that flexible and principled domain abstraction as defined here constitutes a useful programming model for complex software systems. Following these design goals, we developed a novel approach to domain abstraction, which we outline in the following section.

1.3 Extensible languages for domain abstraction

Existing approaches for domain abstraction fulfill many of the design goals discussed above. In particular, we observe that existing mechanisms for the definition of external DSLs provide flexibility, whereas existing mechanisms for the definition of internal DSLs are principled.

However, existing approaches provide insufficient support for the development of complex software systems because external DSLs lack important principles such as modular reasoning or composability, whereas internal DSLs are greatly restricted by the flexibility of the host language, which prevents true domain-specific syntax, domain-specific static analyses, and domain-specific tool support. We are looking for new forms of abstraction that combine the strengths of external and internal DSLs.

We propose the use of *extensible host languages* for domain abstraction, where domain concepts are integrated through language extensions. A language extension defines an embedding of the domain concepts into the host language. Since the host language is extensible, a language extension can, for example, introduce domain-specific syntax or domain-specific static analyses. Thus, extensible host languages break with the traditional inflexibility of internal DSLs. Simultaneously, extensible host languages can retain the benefits of internal DSLs.

Extensible programming languages have been an active research topic since the development of Lisp in the late 1950s [McC60]. Since then, and in particular in recent years, many extensible programming languages have been proposed, for example, ECL [Weg70], AEPL [KM71], Scheme [SDF+09, DHB92], Racket [Fla12, THSAC+11], Nemerle [SMO04], Katahdin [Sea07], Fortress [ACN+09], Helvetia [RGN10], or Honu [RF12]. The domain abstraction supported by these languages varies from fully flexible but unprincipled to rather restricted yet principled. For example, Scheme provides restricted flexibility in its macro system, which allows macros to define domain-specific syntax [Kri06] only as long as this syntax follows the s-expression format and starts with a unique macro identifier. On the other hand, Scheme macros support the important principle of referential transparency [CR91] through hygienic macro expansion [CR91, DHB92]. Other languages such as Helvetia provide a more flexible extensible syntax, but cannot guarantee referential transparency. We present a detailed comparison of existing approaches to domain abstraction in Chapter 8.

In this thesis, we explore a novel design for extensible programming languages. The central idea of our design is to organize *language extensions as libraries*. That is, programmers can define language extensions as libraries of the host language, and libraries can extend the semantics, syntax, static analysis, and tool support of the host language.

A library that contains a language extension behaves like a regular programming library. In particular, a library encapsulates and scopes any language extension that it contains. Consequently, language extensions are never activated implicitly. Instead, to

use a language extension, a programmer brings the extension into the current scope by importing the corresponding library.

In this thesis, we explore the design of language extensions as libraries through the development and refinement of an extensible programming language called *SugarJ*. The design of SugarJ targets flexible and principled domain abstraction. Based on library-based language extensibility, SugarJ provides the following features:

- domain-specific semantics, syntax, static analysis, and editor support through language extensions defined in libraries,
- polymorphic interpretations of domain-specific programs,
- modular reasoning on active language extensions and separate compilation of libraries,
- limited referential transparency based on communication integrity,
- declarative and reusable extension definitions based on SDF, Stratego, and Spoofax,
- declarative support for layout-sensitive syntax,
- composition of independent extensions,
- self-applicable extensions that target the extension mechanism itself,
- and independence of the base language.

In the design of SugarJ, we focused on library-based syntactic extensibility for Java. The goal was to provide programmers with a customizable surface syntax that allows them to write domain-specific programs more conveniently. To this end, we developed a methodology for incremental, import-dependent parsing of a source file, where each imported library can change the parser for the remainder of the file. This incremental parsing methodology is one of the core technical enablers of SugarJ.

Since we selected libraries as the main organizational unit for language extensions, the initial design of SugarJ supports modular reasoning and separate compilation. Essentially, to reason about a SugarJ source file, it suffices to inspect the imported libraries and the code of the current source file itself. The imported libraries fully determine the active language extensions, so that the remaining source code can be understood by the programmer and our compiler. While this may be unsurprising for users of macro systems such as Scheme, many domain-abstraction approaches apply external, global build scripts to activate language extensions. Since such build scripts are outside the source file, a programmer cannot locally reason about the active extensions. In contrast, SugarJ programmers use import statements to activate language extensions in the current module.

Another benefit of our design is that libraries provide a good means for code reuse. With respect to language extensions, two forms of code reuse are relevant. First, extensions can share and reuse part of their implementation by importing libraries that contain

auxiliary definitions for building extensions. Second, users can share and reuse language extensions by importing the same language extensions. Therefore, our design encourages the decomposition of language extensions into small, reusable units. However, this makes support for the composition of language extensions even more important. For using multiple extensions, our design aligns with the use of libraries in regular programming languages: A programmer simply imports all needed language extensions into a single source file. The SugarJ compiler composes all language extensions that are in scope of a source file before processing the body of the file. Technically, this requires a composable metalanguage for the definition of language extensions. For this reason, we chose SDF [Vis97b] and Stratego [VBT98] as metalanguages for the description of language extensions in SugarJ.

It is important to note, though, that SDF and Stratego are fully integrated into SugarJ. That is, SugarJ comprises Java, SDF, and Stratego. As consequence of this integration, the extension mechanism of SugarJ is self-applicable: Like a programmer can use a regular library in the implementation of another library by importing it, a programmer can also use a language extension in the definition of another language extension by importing it. Pragmatically, this means that a SugarJ programmer can define language extensions for the metalanguages SDF and Stratego. This way, SugarJ programmers can enjoy the benefits of domain abstraction while writing language extensions.

In a setting like SugarJ, where the language is subject to customization, conventional tool support fails, because it is oblivious to language extensions. For example, the syntax-coloring services of conventional Java IDEs such as Eclipse [The12] fail to color embedded XML syntax correctly. To address this issue, we designed an extensible IDE based on the language workbench Spoofax [KV10]. Spoofax provides a set of DSLs for the declaration of editor services. We integrated these DSLs into SugarJ such that programmers can declare editor services in a SugarJ library. In contrast to Spoofax, our extensible IDE does not activate editor services globally. Instead, our extensible IDE activates editor services based on the imported libraries on a file-by-file basis. Thus, SugarJ programmers can accompany a language extensions with corresponding editor services, which are imported together with the syntactic extension. This way, SugarJ provides an editing experience similar to what programmers know from mainstream languages such as Java.

In addition to SugarJ, we developed three dialects of the language: SugarHaskell, SugarProlog, and JProMo. SugarHaskell is an extensible programming language that uses Haskell as base language for application code. In the context of Haskell, layout-sensitive syntax is a major issue, which we addressed by developing a declarative and composable formalism for the specification of layout-sensitive languages. Furthermore, we reengineered our implementation of SugarJ to enable extensibility for other base languages than Java. In particular, we defined an interface that abstracts over the base-language dependencies of the SugarJ compiler. To demonstrate the host-language independence of the reengineered SugarJ compiler, we instantiated this interface for three

base languages: SugarJ, SugarHaskell, and SugarProlog.

We developed JProMo to explore polymorphic domain abstractions and to improve on SugarJ's lack of referential transparency. For polymorphic domain abstraction, we found inspiration in works on model-driven software development, where a domain abstraction (represented as a metamodel) can have many semantics (represented as model transformations). This enables the reuse of a single domain-specific program (a model) in different contexts with different semantics. To study such polymorphic interpretations of domain-specific programs, we designed and implemented an extension of SugarJ called JProMo (Java Programming with Models). JProMo retains SugarJ's central design choice of organizing domain abstractions in libraries, but it enables the transformation of libraries when importing them. That is, an import statement can declare not only the imported module but also a transformation that is applied to the imported module first. This way, different users can apply different transformations to the same domain-specific program. Moreover, we extended SugarJ with communication integrity [MQR95, LV95], which guarantees that a transformation does not inject module dependencies. This improves modular reasoning and represents an important first step toward referential transparency.

1.4 Contributions and outline

The main contribution of this thesis is a novel design for extensible programming languages based on libraries that provide flexible and principled domain abstraction. We have studied this design in-depth by designing SugarJ, developing a compiler and an IDE for it, and exploring the language in numerous case studies.

Alongside our main contribution, this thesis makes further contributions in the areas of language design and language engineering. Many of these contributions have been previously published by the author in collaboration with others in the proceedings of international conferences, symposia, and workshops. In the presentation of this thesis, we roughly follow the historical development of SugarJ.

In Chapter 2, we introduce library-based syntactic language extensibility and present the design of SugarJ. In particular, we describe how a programmer can define and use syntactic extension in SugarJ, and how SugarJ scopes language extensions to enable modular reasoning. Technically, we present the SugarJ compiler, which features separate compilation and applies an innovative incremental parser for import-dependent processing of a source file. We explore the design and demonstrate the applicability of our approach through five case studies: tuple syntax and anonymous first-class functions for Java, an embedding of XML with literal XML syntax, an extension of the metalanguage Stratego for concrete syntax in transformations, and an embedding of the domain-specific metalanguage XML Schema that can be used to define domain-specific dialects of XML. The latter two case studies demonstrate the utility of SugarJ's self-applicable extension

mechanism.

In Chapter 3, we focus on IDE support for extensible programming languages. To this end, we present an extensible IDE based on editor extensions, which are organized in libraries. For each file, our IDE inspects the editor extensions brought into scope with import statements, and presents the corresponding editor services to the user. We discuss the composability of user-defined editor services and demonstrate our extensible IDE by developing editor extensions for XML and Latex that give the look-and-feel of standalone XML and Latex IDEs.

In Chapter 4, we present groundwork for a variant of SugarJ based on Haskell, which employs a layout-sensitive syntax. To support Haskell, we develop an extension of SDF that features a declarative mechanism for the specification of layout-sensitive languages: We annotate regular productions of the grammar with layout constraints that restrict the applicability of a production to layout that satisfies the constraint. This mechanism is simple, declarative, and retains the composability of SDF grammars. We develop a generalized parser for grammars with layout constraints, develop layout-sensitive grammars for Python and Haskell, and perform an extensive evaluation by parsing 33 290 files.

In Chapter 5, we introduce the syntactically extensible programming language SugarHaskell, which uses our layout-sensitive parser and the Haskell grammar. In particular, SugarHaskell not only employs a layout-sensitive base language but also allows programmers to declare layout-sensitive syntax extensions. We present language extensions for applicative functors, arrows, and EBNF-based declarations of concrete and abstract syntax. Technically, we describe our implementation of a framework for building extensible languages with which support for new base languages can be realized relatively easy.

In Chapter 6, we introduce the model-oriented-programming paradigm. Model-oriented programming is a programming-language approach to model-driven development, where models, metamodels, and transformations are represented as libraries, and the application of a transformation to a model is explicitly declared with import statements. We realized model-oriented programming in the programming language JProMo, which is built on top of SugarJ. JProMo extends SugarJ both with respect to flexibility and principles. In particular, JProMo adds flexible polymorphic domain abstraction by separating models from transformations, and guarantees communication integrity as a first step toward referential transparency. We demonstrate the applicability of these new features with case studies on statemachines and #ifdef-based software product lines.

In Chapter 7, we focus on language composability, one of the most important principles applied in SugarJ, because our library-based design facilitates the decomposition of domains into multiple libraries and the composition of multiple libraries in a single file. In Chapter 7, we take a step back to investigate the meaning of language composition, to classify different forms of language composition, and to survey the support for language composition in existing systems. In particular, we introduce a precise terminology and an algebraic notation for describing language composition.

In Chapter 8, we discuss SugarJ in a wider context of related work and compare it with other approaches to domain abstraction. We provide a tabular overview of existing approaches using the design goals on flexible and principled domain abstraction that we introduced in the present chapter. As it turns out, the design goals provide a characterization of existing systems where no two systems satisfy the same goals. Furthermore, each of our design goals is addressed by some systems but not all of them—except for domain-specific semantics which is a necessity for domain abstraction.

In Chapter 9, we summarize our contributions and provide suggestions for future work on extensible languages.

We have realized all work described in this thesis in concrete implementations to guide and evaluate our design. All our implementations are open source and the source code of the following artifacts is available via `http://sugarj.org`:

- SugarJ compiler,
- SugarJ IDE,
- layout-sensitive generalized LR parser,
- plug-in-based compiler framework for extensible languages,
- SugarJ, SugarHaskell, SugarProlog, and SugarFomega compiler plugins,
- compiler for the model-oriented programming language JProMo,
- case studies for SugarJ, SugarHaskell, and JProMo (see overview in Appendix A).

The development of these tools represents another major contribution of this thesis. Our tools can be used by other researchers as the basis for further work. In particular, the extensible languages SugarJ, SugarProlog, and SugarHaskell can serve as research platforms for exploring language design in general, and future extensions of Java, Prolog, and Haskell in particular.

2 Syntactic Language Extensibility

This chapter shares material with the OOPSLA'11 paper "SugarJ: Library-based Syntactic Language Extensibility" [ERKO11].

We start our exploration of flexible yet principled extensible languages by focusing on extensible syntax. To this end, we present *sugar libraries*, a novel approach for syntactically extending a programming language within the language. A sugar library is like an ordinary library, but can, in addition, export syntactic sugar for using the library. The syntactic extensibility supported by sugar libraries comprises the full class of context-free languages. In particular, sugar libraries do not require keywords or macro names to mark the code belonging to some extension. Instead, syntactic extensions can be freely integrated into the host language syntax.

On the other hand, sugar libraries maintain the composability and scoping properties of ordinary libraries. Sugar libraries are never active by default. Instead, programmers *import* the sugar libraries they want to use. To apply multiple language extensions, a programmer simply imports all corresponding sugar libraries and thereby composes them. Since sugar libraries must be imported explicitly, programmers can modularly reason about their programs despite the use of language extensions. Furthermore, sugar libraries inherit self-applicability from regular libraries, which means that sugar libraries can provide syntactic extensions for the definition of other sugar libraries.

We realized sugar libraries in the syntactically extensible programming language *SugarJ*. SugarJ employs a novel incremental parsing technique, which allows changing the syntax within a source file. We demonstrate SugarJ by five language extensions, including embeddings of XML and closures in Java, all available as sugar libraries. We illustrate the utility of self-applicability by embedding XML Schema, a metalanguage to define XML languages.

2.1 Introduction

DSLs can bridge the representational gap between domain concepts and the implementation of these concepts in a programming language (see Figure 1.1). Accordingly, DSLs, such as regular expressions for the domain of text recognition or Java Server Pages for the domain of dynamic web pages, have often been argued to simplify software development [MHS05]. However, to use DSLs in large software systems that touch multiple domains, developers have to be able to compose multiple DSLs and embed them into a

```
import pair.Sugar;

public class Test {
  private (Integer, String) p = (17, "seventeen");
}
```

Figure 2.1: The import statement activates literal pair syntax in the current file.

common host language [Hud98]. In this context, we consider the long-standing problem of domain-specific *syntax* [Lea66, WC93, BLS98, BS02, BV04, RGN10].

Our novel contribution is the notion of *sugar libraries*, a technique to syntactically extend a programming language in the form of libraries. In addition to the semantic artifacts conventionally exported by a library, such as classes and methods, sugar libraries export also syntactic sugar that provides a user-defined syntax for using the semantic artifacts exported by the library. Each piece of syntactic sugar defines some extended syntax and a transformation—called *desugaring*—of the extended syntax into the syntax of the host language. Sugar libraries enjoy the same benefits as conventional libraries: (i) They can be used where needed by importing the syntactic sugar as exemplified in Figure 2.1. (ii) The syntax of multiple DSLs can be composed by importing all corresponding sugar libraries; their composition may form a new higher-level DSL that can again be packaged as a sugar library. (iii) Sugar libraries are self-applicable: They can import other sugar libraries and the syntax for specifying syntactic sugar can be extended as well.

In other words, sugar libraries treat language extensions in a unified and regular fashion at all metalevels. Here, we apply a conceptual understanding of "metalevel", which distinguishes the definition of a language from its usage: A language definition is at a higher metalevel than the programs written in that language. In this sense, sugar libraries (defining language extensions) are on a higher metalevel than the programs that use the sugar library, and the import of a sugar library acts across metalevels.

Sugar libraries are not limited to DSL embeddings; they can be used for arbitrary extensions of the surface syntax of a host language (for instance, an alternative syntax for method calls). However, due to their composability and their alignment with the import and export mechanism of libraries, they qualify especially for embedding DSLs.

To explore sugar libraries, we have designed and implemented sugar libraries in *SugarJ*. SugarJ is a programming language based on Java that supports sugar libraries by building on the grammar formalism SDF [Vis97b] and the transformation system Stratego [VBT98]. As an example of SugarJ's syntactic extensibility, in Figure 2.1, we import a sugar library for pairs that enables the use of pair expressions and types with pair-specific syntax. We show the corresponding sugar library pair.Sugar in Figure 2.2. It provides convenient syntax for the semantic encoding of pairs as a generic class Pair<A,B>.

```
package pair;
public class Pair<A,B> { ... }
```

(a) A generic Java class that implements the semantics of pairs.

```
package pair;

import org.sugarj.languages.Java;
import concretesyntax.Java;

public sugar Sugar {
  context-free syntax
    "(" JavaType "," JavaType ")" -> JavaType {cons("PairType")}
    "(" JavaExpr "," JavaExpr ")" -> JavaExpr {cons("PairExpr")}

  desugarings
    desugar-pair-type
    desugar-pair-expr

  rules
    desugar-pair-type :
      PairType(t1, t2) -> |[ pair.Pair<~t1, ~t2> ]|
    desugar-pair-expr :
      PairExpr(e1, e2) -> |[ pair.Pair.create(~e1, ~e2) ]|
}
```

(b) A sugar library that defines literal pair syntax and desugarings for expressions and types.

Figure 2.2: Sugar libraries provide convenient syntax for semantic encodings.

The pair.Sugar declaration extends the Java syntax with syntax for pair types and expressions by adding productions for the existing nonterminals JavaType and JavaExpr. To associate meaning to the new pair syntax, pair.Sugar also stipulates how pair types and expressions are desugared into Java. In Figure 2.1, for example, the desugaring transforms the pair type (String, Integer) into the Java type Pair<String, Integer> and the pair expression (17,"seventeen") into a static method call pair.Pair.create(17,"seventeen"). Since SugarJ supports arbitrary compile-time computation, sugar libraries can implement even intricate source transformations, perform domain-specific compile-time analyses, and program optimizations.

To set the context for SugarJ, in the following section we briefly review the syntactic extensibility of existing DSL embedding approaches. Subsequently, in this chapter, we

present the following contributions:

- We introduce the novel concept of sugar libraries, a *library-centric* approach for syntactic extensibility of host languages (Section 2.3). Sugar libraries enable the uniform embedding of DSLs at syntactic and semantic level, and retain the composability properties of conventional libraries.
- Sugar libraries combine the benefits of existing approaches: Sugar libraries support flexible domain-specific syntax (based on arbitrary context-free grammars and compile-time checks), scope language extensions, can be imported across metalevels, and act on all metalevels uniformly to enable syntactic extensions in metaprograms (self-applicability).
- The *simplicity* of activating syntactic extensions by import statements and the language-integrated support to develop new syntactic extension, even for small language extensions, encourages development in a language-oriented [Dmi04, Fow05b, War95] fashion.
- We present our implementation of SugarJ on top of existing languages, namely Java, SDF and Stratego, and explain the mechanics of compiling our syntactically extensible programming language (Section 2.4).
- Technically, we present an innovative incremental way of parsing files, in which different regions of a file adhere to different grammars from different syntactic extensions.
- We demonstrate the expressiveness and applicability of SugarJ on the basis of five case studies—pairs, closures, XML, concrete syntax in transformations, and XML Schema. The latter is an advanced example of self-applicability, since each XML Schema defines a new XML language (Section 2.5).

2.2 Syntactic embedding of DSLs

Many approaches for embedding a DSL into a host language focus on the integration of domain concepts at semantic level (e.g., [Oli09, HORM08, HO10]), but neglect the need for expressing domain concepts using domain-specific syntax. To set the context for sugar libraries, we survey the syntactic amenability of existing DSL embedding approaches here, and present a more thorough treatment of related work in Chapter 8.

String encoding. The simplest form of representing a DSL program in a host language is as unprocessed source code encoded as a host-language string. Since most characters may occur in strings freely, such encoding is syntactically flexible. Consider, for instance, the following Java program, which writes an XML document to some output stream `out`.

```
String title = "Sweetness and Power";
out.write("<book title=\"" + title + "\">\n");
out.write("  <author name=\"Sidney W. Mintz\" />\n");
out.write("</book>");
```

The string encoding allows writing XML code with element tags and attributes naturally. Nevertheless, in XML documents nested quotes and special whitespace symbols such as newline have to be escaped, leading to less legible code. Moreover, the syntax of string-encoded DSL programs is not statically checked but parsed at run time. Hence, syntactic errors are not detected during compilation and can occur after deploying the software. Furthermore, string encoded programs have no syntactic model and, therefore, can only be composed at a lexical level by concatenating strings. This form of composition resembles lexical macro expansion in a way that is not amenable to parsing [EO10] and opens the door to security problems such as SQL injection or cross-site scripting attacks [BDV10].

Library embedding. To avoid lexical string composition and syntax errors at run time, we can alternatively embed a DSL as a library, that is, a reusable collection of functionality accessible through an API. In Hudak's pure-embedding approach [Hud98], for instance, one builds a library whose functions implement DSL concepts and are used to describe DSL programs. For example, we can embed XML purely as follows:

```
String title = "Sweetness and Power";
Element book =
  element("book",
    attributes(attribute("title", title)),
    elements(
      element("author",
        attributes(attribute("name", "Sidney W. Mintz")),
        elements()))));
```

The syntax of the DSL can be encoded in the type system of the host language, so that, in a statically typed host language, the DSL program is syntax checked at compile time. In our example, such checks can prevent confusion of XML attributes and XML elements. But even in an untyped host language, purely embedded XML documents are properly nested by design, that is, it is not possible to describe ill-formed documents such as <a>.

An apparent drawback of purely embedded DSLs is the syntactic inflexibility of the approach: Programmers must adopt the syntax of function calls in the host language to describe DSL programs. Consequently, when solving a domain-specific problem, the programmer needs to "translate" any conceived domain-specific solution into the host language's syntax manually. Some host languages partially address this problem by overloading built-in or user-defined infix operators (e.g., Smalltalk), integer or string

literals (e.g. Haskell), or function calls (e.g., Scala). However, even in these languages a DSL implementer can only extend the host language's syntax in a limited, preplanned way. For example, while Scala supports quite flexible syntax for method calls, the syntax for class declarations is fixed.

To circumvent the need for manual translation of domain concepts, researchers have proposed the use of syntactically extensible host languages that support the syntactic embedding of DSLs [BP01, BS02, Tra08, WC93]. In particular, languages with macro facilities (or similar metaprogramming facilities) can be used to develop library-based syntactic embeddings of DSLs [Kri06]. Unfortunately, most macro languages only support user-defined syntax for macro arguments [BS02]. This obstructive requirement for explicit macro invocations prevents the usage of macro systems to syntactically embed DSLs like XML into a host language freely [BV04].

Independent of their syntactic inflexibility, one essential advantage of library embeddings is the composability of DSLs. By importing multiple libraries, a programmer can easily compose those libraries to build a new one. Since embedded DSLs are implemented as libraries of the host language, library composition entails the composition of DSL implementations. Therefore, library embedding supports modular definitions of DSLs on top of previously existing ones [HORM08]. These benefits of library embedding are the starting point and main motivation for our sugar-library approach.

Language extension. To support fully flexible domain-specific syntax, one possibility is to extend the host language such that it comprises the DSL. In this approach, syntactic and semantic language extensions are incorporated into the host language by directly modifying the host language's implementation or using an extensible compiler. Usually, language extensions are not restricted in the syntax they introduce: DSL implementors can integrate arbitrary DSL syntax and semantics into the host language. For example, Scala provides built-in support for XML documents:

```
val title = "Sweetness and Power"
val book =
  <book title="{title}">
    <author name="Sidney W. Mintz" />
  </book>
```

Scala's support for XML syntax has been directly integrated into the Scala compiler, which translates XML syntax trees into calls to the scala.xml library [Ode10]. Since the Scala compiler parses embedded XML documents at compile time, run-time syntax errors cannot occur and ill-formed documents cannot be generated. Moreover, compared to the nested library calls of a pure XML embedding, users of an XML-extended host language can write programs more naturally using literal XML syntax.

In general, modifying a (nonextensible) compiler to incorporate a DSL into the host language is impracticable and makes it hard to develop or compose independent DSLs.

More generic approaches for extending a language therefore support modular definition and integration of DSLs and are not specific to the used host language. In these approaches, which include extensible compilers [EH07a, NCM03] and program transformation systems [BV04, VKBS07], the active language extensions are determined by compiler configurations or by generating and selecting the right compiler variant. This becomes impractical if programmers use different combinations of DSLs in different source files: Compiler variants or configurations have to be generated for each combination of DSLs, and a significant part of the program's semantics and dependency structure is moved from the program sources to build scripts or configuration files.

Summary. String embedding is syntactically very flexible but lacks static safety and composability. Library embeddings excel in composability but lack syntactic flexibility. Language extensions are powerful but hard to implement and compose, and introduce an undesirable stratification into base code and metalevel code. Obviously, it would be beneficial to combine the respective strengths of these approaches.

2.3 SugarJ: Sugar libraries for Java

We propose to organize syntactic language extensions into sugar libraries. A sugar library encapsulates the specification of a syntax extension and a accompanying desugaring from the extended syntax into host-language syntax. To use a syntactic extension, a developer simply imports the corresponding sugar library. A sugar-library import activates the syntactic extension in the current module and allows the developer to use the new syntactic constructs in subsequent segments of the same file. Programmers and metaprogrammers can uniformly import sugar libraries to implement applications or other sugar libraries.

To demonstrate the concept, we have designed and implemented SugarJ, a variant of Java with support for sugar libraries. SugarJ is a host language for language embedding that comprises three existing languages: Java is used as base language for application code, the syntax definition formalism (SDF) [Vis97b] is used to describe concrete syntax of extensions, and the Stratego transformation language [VBT98] is used to describe desugarings of extension-specific code into SugarJ code. Importantly, extension-specific code can not only desugar into Java but into the full host language SugarJ. In particular, extension-specific code can desugar into SDF and Stratego fragments that define yet another syntactic extension.

We introduce sugar libraries by walking through an example. We extend Java with closures by introducing syntactic sugar and corresponding desugarings of the introduced closure syntax into plain Java code. (Closures, or lambda expressions, or anonymous functions are an frequently-requested feature for Java and plans exist to integrate closures into Java 8, which is expected for 2013.)

```
package javaclosure;
public interface Closure<Result, Argument> {
  public Result invoke(Argument argument);
}
```

(a) An interface for function objects.

```
final int factor = ...;
Closure<Integer, Integer> scale =
  new Closure<Integer, Integer>() {
    public Integer invoke(Integer x) {
      return x * factor;
    }
  };
List<Integer> scaled = original.map(scale);
```

(b) A closure that scales its input by a constant factor.

Figure 2.3: Closures can be implemented as function objects, but Java does not offer convenient syntax for closure types and expressions.

2.3.1 Using a sugar library

To use a sugar library, a programmer only has to import the library with an ordinary import statement. In a source file that imports a sugar library, the programmer may use syntax introduced by the library anywhere after its import. All syntax constructs from the library are desugared into plain Java code (more precisely into SugarJ code, because desugarings can produce new syntax extensions) automatically at compile time.

Our closure example illustrates the benefits of sugar libraries for programmers and how easy such libraries are to use. In plain Java code, a programmer would typically implement closures as anonymous inner classes as illustrated in Figure 2.3. However, the syntax is rather verbose, especially for the frequent use case of an anonymous inner class with exactly one method. With SugarJ, a programmer can import a sugar library that introduces a more concise notation for closures, following roughly the proposal of Gafter and von der Ahé [GvdA09] (one of several syntax suggestions). With this syntactic extension, we can rewrite our example as illustrated in Figure 2.4: Instead of verbose Java code, we write #R(T) to denote a closure type Closure<R, T> and #R(T t) { stmts...; **return** exp; } to denote a closure. The verbose code of Figure 2.3 and the concise of Figure 2.4 are equivalent. SugarJ automatically desugars the concise version into plain Java code at compile time.

```
import javaclosure.Syntax;
import javaclosure.Desugar;
```

(a) Import statements activate the sugar library for closures.

```
final int factor = ...;
#Integer(Integer) scale =
  #Integer(Integer x) { return x * factor; };
List<Integer> scaled = original.map(scale);
```

(b) Using specialized syntax for creating a closure.

Figure 2.4: The closure sugar library provides concise syntax for the declaration of closure types and expressions.

2.3.2 Writing a sugar library

To write a sugar library, one has to define how to extend the language and how to desugar the extension. Hence, a sugar library consists of two parts: An extension of the host language's grammar with new syntax rules and a desugaring of the new language constructs into the original language.

In SugarJ, programmers define both parts through top-level sugar declarations of the form **public sugar** Name { ... }, which contain SDF and Stratego code organized into sections. While we support all features of SDF and Stratego, here we concentrate on the features most essential for writing sugar libraries: syntax rules and program-transformation rules.

In an SDF section **context-free syntax**, a library developer can extend the host language's grammar with new syntax rules. Figure 2.5(a) shows the syntax rules for our closure example. A syntax rule specifies the nonterminal to be extended (to the right of the arrow ->), a pattern for the newly introduced concrete syntax (to the left of the arrow), and a name for the syntax tree node created by this production (in the cons annotation). Importantly, a syntax rule can refer to and extend existing nonterminals even if these are declared within other modules. In our example, we import the Java grammar org.sugarj.languages.Java to bring Java's nonterminals into scope. This way, sugar libraries can introduce new syntax for any syntactic category (e.g., class declarations, expressions or import statements) by extending SugarJ nonterminals or nonterminals introduced by other sugar libraries.

In a Stratego section **rules**, a library developer can define program transformations, called desugaring rules. We illustrate Stratego rules for closures in Figure 2.5(b). A desugaring rule consists of a name (before the colon), a matching pattern (to the left of the arrow) and a generation template (to the right of the arrow). Both pattern and template

```
package javaclosure;

import org.sugarj.languages.Java;
import concretesyntax.Java;

public sugar Syntax {
  context-free syntax
    "#" JavaType "(" JavaType ")"
      -> JavaType {cons("ClosureType")}

    "#" JavaType "(" JavaFormalParam ")" JavaBlock
      -> JavaExpr {cons("ClosureExpr")}
}
```

(a) We extend the Java grammar with syntax for closure types and expressions.

```
public sugar Desugar {
  rules
    desugar-closure-type :
      |[ #~result(~argument) ]|
      -> |[ javaclosure.Closure<? extends ~result, ? super ~argument> ]|

    desugar-closure-expr :
      |[ #~result(~argument ~id:x) ~block:body ]|
      -> |[ new javaclosure.Closure<~result, ~argument>() {
              public ~result invoke(~argument ~id:x)
                  ~block:body
            }
          ]|

  desugarings
    desugar-closure-type
    desugar-closure-expr
}
```

(b) We desugar closure types into reference types of the Closure interface and closure expressions into anonymous classes implementing the Closure interface.

Figure 2.5: A sugar library for closures that we split over two sugar declarations so that the syntax rules for closures are in scope of the desugaring declaration.

are specified using concrete syntax in brackets |[...]|, where metavariables are written with an initial tilde ~ [Vis02]. A desugaring rule denotes a program transformation from the extended syntax to the host language (possibly with some other extension).

Desugaring rules are specified using concrete syntax, so that a programmer does not need to read or write abstract syntax trees. In our example, the rule desugar-closure-type in Figure 2.5(b) matches on closure types using the # ... (...) concrete syntax just introduced in Figure 2.5(a). For technical reasons, a syntax rule is only activated *after* the sugar declaration it is defined in.[1] Therefore, one typically splits a sugar library into two parts, introducing syntax rules and desugaring rules separately, so the syntax rules for closures are in scope when we define the desugaring rules for closures. Accordingly, desugar-closure-type transforms a closure type into a reference type of the javaclosure.Closure interface. As a general coding convention, in desugarings, we write fully-qualified Java references to maintain referential transparency [CR91] (more on that later).

The transformation desugar-closure-expr matches on closure expressions and transforms them into declarations of anonymous classes that implement the javaclosure.Closure interface. Again we make extensive use of concrete syntax in the transformation. In fact, the definition of the desugaring rule exactly reveals the syntactic boilerplate we avoid with our closure abstraction. Instead of writing a verbose declaration of an anonymous class every time we need a closure, we use syntactic abstraction to provide a closure-specific shorthand notation.

In a final section **desugarings** of the sugar library, the library developer declares the main desugaring rules. After parsing, the SugarJ compiler exhaustively applies these desugaring rules in a bottom-up fashion, starting at the syntax tree's leaves and progressing towards its root. Compilation fails if an input program cannot be unambiguously parsed with the combination of all syntax rules in scope, if any of the triggered desugaring rules signals an error, or if the desugared program still contains fragments of user extensions.

2.3.3 Composing sugar libraries

Sugar libraries are composed by importing more than one sugar library into the same file. For example, in Figure 2.6, we import the sugar library for closures together with a sugar library for pairs to implement partial application of a function that expects a pair as input. Note that instead of importing javaclosure.Syntax and javaclosure.Desugar separately, we could have defined a compound module javaclosure.Sugar and import this one.[2] The scope of each sugar library is annotated in the figure. The syntax for closures

[1] Our implementation supports syntax changes only between top-level declarations, but not in the middle of, for example, a sugar declaration. See Sections 2.3.3 and 2.4 for details.

[2] Java supports wildcard imports like **import** javaclosure.*, but their semantics is ill-suited for our purpose: A wildcard import only affects unqualified class names, but the name of a sugar library never occurs

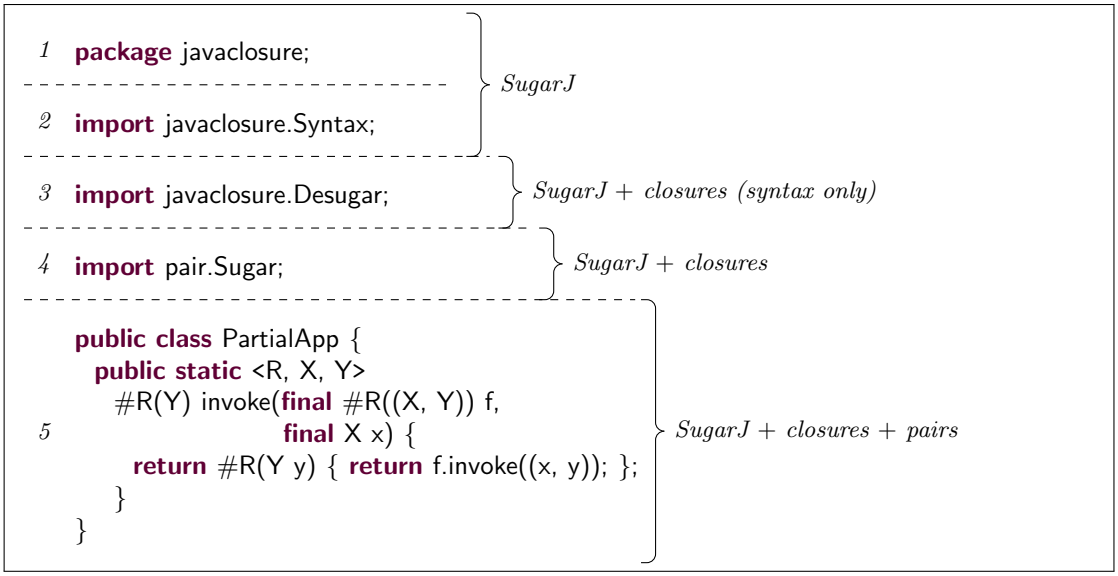


Figure 2.6: Imports of multiple sugar libraries composes the syntactic extensions.

and the syntax for pairs can be freely mixed in the class declaration, where both sugar libraries are in scope.

To merge several syntactic sugar, SugarJ composes the grammar extension and desugaring declarations of sugar libraries. The composability of the underlying grammar formalism and transformation language was the main criteria for deciding to build SugarJ on top of SDF and Stratego. Composing two sugar libraries is not always possible entirely without conflicts or ambiguities if the syntactic extensions overlap. Our experience, however, suggests that in most practical cases libraries can be freely composed or conflicts can be easily detected and fixed, see our discussion in Section 2.6.1.

2.4 SugarJ: Technical realization

A compiler for SugarJ parses and desugars a SugarJ source file and produces a Java file together with grammar and desugaring rules as output. Subsequently, we can compile the Java file into byte code, whereas the grammar and desugaring rules are stored separately as a form of library interface for further imports from other SugarJ files. In this section, we assume that desugaring rules are program transformations between syntax trees. Later, in Section 2.5.1, we show how an ordinary sugar library can extend SugarJ to support desugarings rules in terms of concrete syntax, as used in the examples so far.

in a source file. Instead, the SugarJ compiler needs to immediately import the sugar library to parse the next top-level declaration with an updated grammar.

2.4.1 The scope of sugar libraries

To parse and desugar a SugarJ source file, the compiler keeps track of which grammar and desugaring rules apply to which parts of the source file. Through importing or defining a sugar library, the grammar and desugaring rules may change within a single source file. Moreover, definitions and import statements of sugar libraries may themselves be written using an extended syntax. Thus, the compiler has to desugar such definitions before continuing to parse the remainder of the file.

In SugarJ, imports and declarations of sugar libraries can only occur at the top-most level of a file, but not nested inside other declarations. Therefore, the scope of grammar and desugaring rules always aligns with the top-level structure of a file. For example, in Figure 2.6, the grammar and desugaring rules change between the the second and the third top-level entry for the first time, hence the third top-level entry is parsed and desugared in a different context. Subsequently, it changes again after the third and after the fourth top-level entry, which influences parsing and desugaring of the remaining file. This alignment allows the SugarJ compiler to interleave parsing and desugaring at the granularity of top-level entries.

2.4.2 Incremental processing of SugarJ files

Our SugarJ compiler parses and desugars a SugarJ source file one top-level entry at a time, keeping track of changes to the grammar and desugaring rules, which affect the processing of subsequent top-level entries. A top-level entry in SugarJ is either a package declaration, an import statement, a Java type declaration, a declaration of syntactic sugar, or a user-defined top-level entry introduced with a sugar library. As illustrated in Figure 2.7, the compiler processes each top-level declaration in four steps: parsing, desugaring, generation, and adaption.

Parsing. Each top-level entry is parsed using the *current grammar*, that is, the grammar which reflects all sugar libraries currently in scope. For the first top-level entry, the current grammar is the initial SugarJ grammar, which comprises Java, SDF, and Stratego syntax definitions. For subsequent top-level entries, the current grammar may differ due to declared or imported syntactic sugar. The result of parsing is a heterogeneous abstract syntax tree, which can contain both predefined SugarJ nodes and user-defined nodes.

Desugaring. Next, the compiler desugars user-defined extension nodes of each top-level entry into predefined SugarJ nodes using the *current desugaring*. For each top-level entry, the current desugaring consists of the desugaring rules currently in scope, that is, the desugaring rules from the previously declared or imported sugar libraries. Desugarings are transformations of the abstract syntax tree, which the compiler applies in a bottom-up order to all abstract-syntax-tree nodes until a fixed point is reached. The result of

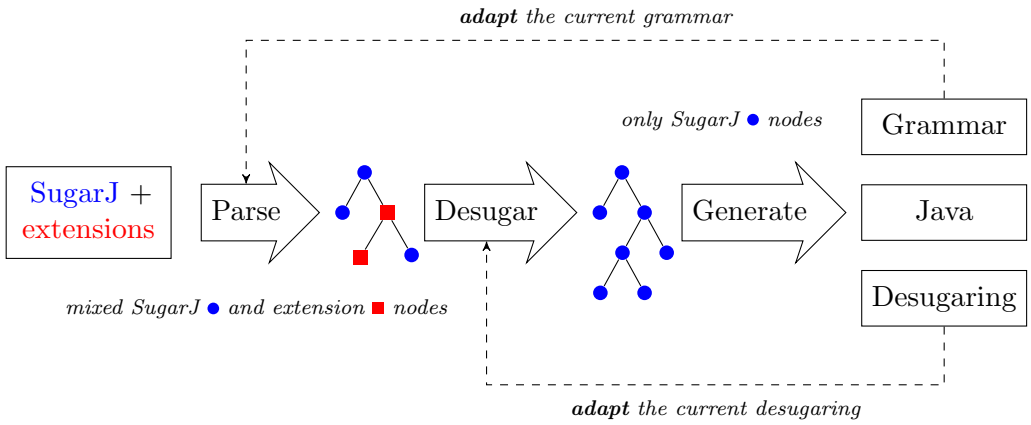


Figure 2.7: Processing of a SugarJ top-level declaration.

this desugaring step is a homogeneous abstract syntax tree, which contains only nodes declared in the initial SugarJ grammar (if some user-specific syntax was not desugared, the compiler issues an error message). Thus, this tree represents one of the predefined top-level entries in SugarJ and is therefore composed only of nodes describing Java code, grammar rules, or desugaring transformations. From these constituents, the compiler generates three separate artifacts.

Generation. We split each top-level SugarJ declaration into fragments of Java, SDF, and Stratego and reuse their respective implementations. Java top-level forms are written into the Java output, whereas a sugar declaration affects the grammar and desugaring output. Package declarations and import statements, on the other hand, are forwarded to all output artifacts to align the module systems of Java, SDF, and Stratego.

After processing the last top-level declaration, the Java file contains pure Java code and the grammar specification and desugaring rules are written in a form that can be imported by other SugarJ files. In case any produced artifact does not compile, the SugarJ compiler issues a corresponding error message. So far, however, the compiler can only report errors in terms of desugared programs.

Adaption. As introduced above, sugar declarations and imports affect the parsing and desugaring of all subsequent code in the same file. Therefore, after each top-level entry, we reflect possible syntactic extensions by adapting the *current* grammar and the *current* desugaring.

After desugaring, if the top-level declaration is a new sugar declaration, we (a) *compose* the current grammar with the grammar of the new declaration and (b) *compose* the current desugaring rules with the desugaring rules of the new declaration. If the top-level

declaration is an import declaration of a sugar library, we load the generated grammar and desugaring artifacts from the class path and compose them with the current grammar and desugaring. On pure Java declarations, we do not need to update the current grammar or desugaring.

When composed, productions of two grammars (e.g., from the initial SugarJ grammar and from a grammar in a sugar library) can interact through the use of shared nonterminal names. Hence, a sugar library can add productions to any nonterminal originally defined either in the initial grammar or in some other sugar library. In that way, nonterminals defined in the initial grammar represent initial extensions points for grammar rules defined in sugar libraries. Similarly, when composed, two sets of desugaring rules can interact through the use of shared names and by producing abstract-syntax-tree nodes that are subsequently desugared by rules from the other set.

Adaptation and composition of grammars and desugarings can take place after each top-level declaration and affects the processing of all subsequent top-level declarations.

2.4.3 The implementation of grammars and desugaring

As mentioned earlier, SugarJ uses the syntax definition formalism SDF [Vis97b] to represent and implement grammars, and the transformation language Stratego [VBT98] to represent and implement desugarings.

Our initial grammar (with regard to the process described in Section 2.4.2) is a standard Java 1.5 grammar augmented by top-level sugar declarations. To enable incremental parsing with different grammars, we have further augmented the Java grammar by a nonterminal which parses a single top-level entry together with the rest of the file as a single string. An alternative approach to this incremental parsing are adaptive grammars, which support changing the grammar at parse time [Shu93]. However, adaptive grammars are inherently context-sensitive, which makes their efficiency questionable. On the other hand, SDF employs a scannerless generalized LR parser [Vis97a] that yields a parse forest at cubic worst-case complexity.

Before using SDF grammars and Stratego transformations, SugarJ has to compile them. Our implementation caches the results of SDF and Stratego compilation to speed up the usual case of using the same combination of sugar libraries multiple times, either processing different files using the same set of sugar libraries, or reprocessing the same file after changes which do not affect the imports. In such a case, our compiler takes only a couple seconds to compile a SugarJ file. However, when changing the language of a SugarJ file, all syntax rules and desugaring rules in scope are recompiled, thus compilation takes considerably longer. Separate compilation [Car97] of grammars and desugarings would help to speed up compilation, but SDF and Stratego traditionally focus on the flexible combination of modules, not on compiling them separately.

2.5 Case studies

Our primary goal in designing SugarJ is to support the integration and composition of DSLs at semantic and syntactic level. To this end, we provide SugarJ with an extensible surface syntax that sugar libraries can freely extend to embed arbitrary domain syntax.

We have embedded a number of language extensions and DSLs into SugarJ, including syntax for pair expression and pair types (Section 2.1), closures for Java (Section 2.3) and regular expressions. All of these case studies are implemented in similar style: One defines an extended syntax and its desugaring into an existing Java implementation for the domain. In this fashion, we could have easily embedded many more DSLs such as Java Server Pages or SQL. Many such case studies have been performed for MetaBorg [BV04]; since we use the same underlying languages for describing grammars and desugarings, namely SDF and Stratego, these embeddings could easily be encoded as sugar libraries by lifting the implementations into SugarJ's syntax and module system. In contrast to the case studies in MetaBorg, the resulting SugarJ libraries can be activated across metalevels and composed by issuing import instructions and need neither complicated compiler configurations nor explicit compound modules. Due to the simplicity of activating sugar libraries, they are not only well-suited for large-scale embeddings of DSLs but also for using several small language extensions such as pairs and closures.

Since the embedding of further ordinary DSLs does not yield more insight, we defer from discussing them here and summarize them in Appendix A instead. Here, we focus our attention on more sophisticated scenarios that demonstrate the flexibility of sugar libraries compared to other technologies. In the pair and closure case studies, we already used a sugar library that provides concrete syntax for implementing program transformations. We will explain this sugar library for concrete syntax in the following subsection. Subsequently, we focus on the composability of sugar libraries by discussing an embedding of XML syntax into SugarJ, which reuses existing sugar libraries in nontrivial ways. We close the present section by illustrating SugarJ's support for implementing meta-DSLs, that is, special-purpose languages for implementing DSLs. Specifically, we embed XML Schema into SugarJ to describe languages of statically validated XML documents.

2.5.1 Concrete syntax in transformations

As described in Section 2.4, the SugarJ compiler parses a SugarJ top-level declaration into an abstract syntax tree before applying any desugaring rules. Internally, desugaring rules are expressed as transformations between abstract syntax trees, even when they are *specified* in terms of concrete syntax, as described in Section 2.3.2. Concrete syntax in transformations significantly increases the usability of SugarJ: A sugar library developer who wants to extend the visible surface syntax should not need to reason about the underlying invisible abstract structure.

To support concrete syntax in transformations, we could have changed the SugarJ compiler, leading to a monolithic and not very flexible design. However, the self-applicability of SugarJ allows a more flexible and modular solution: We implement concrete syntax in transformations as a sugar library concretesyntax.Java that extends the syntax for the specification of sugar libraries itself. We have imported this sugar library in the sugar libraries for pairs and closures above.

For example, the desugaring rules for pair expressions can conveniently be written as a transformation between snippets of concrete syntax as follows:

```
desugar-pair :
  |[ (~expr:e1, ~expr:e2) ]|
  -> |[ pair.Pair.create(~e1, ~e2) ]|
```

This rule is desugared into a transformation between abstract syntax trees as follows:

```
desugar-pair:
  PairExpr(e1, e2)
  -> Invoke(
       Method(MethodName(
         AmbName(AmbName(Id("pair")), Id("Pair")),
         Id("create"))),
       [e1, e2])
```

Visser proposed the use of concrete syntax in the implementation of syntax tree transformation [Vis02] for MetaBorg [BV04]. Technically, a transformation that uses concrete syntax expands to a transformation with abstract syntax by parsing the concrete syntax fragments and injecting the resulting abstract syntax tree. Thus, the left-hand and right-hand sides of the former desugar-pair transformation expand to the ones of the latter transformation. This technique is language-independent and has been implemented generically [Vis02], such that the concrete syntax of any language can be injected into Stratego by extending Stratego's grammar accordingly. For example, to enable concrete syntax for Java expressions in transformations, the following productions specify that quoted Java code is written in brackets |[...]| and unquoted Stratego code is preceded by a tilde ~.

```
"|[" JavaExpr "]|" -> StrategoTerm {cons("ToMetaExpr")}
"~" StrategoTerm   -> JavaExpr     {cons("FromMetaExpr")}
```

In SugarJ, the sugar library for concrete syntax in transformations, whenever it is in scope, automatically desugars concrete syntax into abstract syntax as described above. In contrast, in MetaBorg, the desugaring of concrete syntax is a preprocessing step which the programmer needs to enable manually by accompanying the Stratego source file with an equally named "`*.meta`" file pointing to the SDF module used for desugaring [Vis02]. The reason for this obstructive mechanism is that support for concrete syntax is syntactic

sugar at metalevel. Due to the homogeneous integration of metalanguages in SugarJ, however, SugarJ is host language and metalanguage at the same time. Therefore, language extensions of SugarJ can be developed as sugar libraries in SugarJ itself.

The alignment of host language and metalanguage in SugarJ implies that a programmer can develop and apply language extensions within a single language and never has to specify any configuration external to the language such as a build script or MetaBorg's "`*.meta`" file. This has a fundamental consequence: It enables programmers to conduct modular reasoning. Every fact about a given SugarJ program is derivable from its source code and the modules it references; it is not necessary to take build scripts, configuration files, or, in fact, any code into account that is not referenced within the source file. This becomes particularly important when the number of available DSLs grows, as, for instance, in our implementation of the XML sugar library.

2.5.2 XML documents

The embedding of XML syntax [W3C08], as discussed in Section 2.2, is a good show-case for syntactic extension: Many existing APIs for XML suffer from a syntactic overhead compared to direct use of literal XML notation, XML syntax does not follow the lexical structure of most host languages, and neither well-formedness nor validation of XML documents are context-free properties. The implementation of our sugar library for XML syntax furthermore serves as an example to discuss SugarJ's support for modularity.

Typically, XML is integrated into a host language by providing an API such as the Simple API for XML (SAX) or the Document Object Model. Following the MetaBorg XML embedding [BV04], our sugar library for XML syntax desugars XML syntax into an indirect encoding of documents through SAX calls. For example, in Figure 2.8 an XML document is sent to a content handler `ch`. Compared to Scala's XML support (Section 2.2), sugar libraries provide similar syntactic flexibility without changing the host language's compiler.

The XML sugar library statically ensures that all generated XML documents are well-formed and, to this extent, supports the same static checks as the pure embedding approach shown in Section 2.2. In contrast, the SAX API does not statically detect illegal nesting as in `<a><b></a></b>` or mismatching start and end tags as in `<a></b>`. The XML sugar library arranges to check both properties: the former during parsing and the latter during a separate analysis phase.

The XML sugar library illustrates an interesting distinction of the kind of static analyses we can perform in sugar libraries. On the one hand, context-free properties such as legal nesting of XML elements can be encoded into the syntax definition of a language extension; the compiler verifies context-free properties while parsing the source code. On the other hand, context-sensitive properties cannot be encoded into context-free syntax rules; instead, it is possible to encode the checking of context-sensitive properties as a program transformation that traverses a syntax tree and generates a list of error

```
import xml.XmlJavaSyntax;
import xml.AsSax;
```

(a) Importing the XML syntax and desugaring.

```
public void genXML(ContentHandler ch) {
  String title = "Sweetness and Power";
  ch.<book title="{title}">
       <author name="Sidney W. Mintz" />
    </book>;
}
```

(b) Generating an XML document using XML syntax. The unquote operator {...} allows SugarJ code to occur inside XML documents.

Figure 2.8: XML documents are statically syntax-checked and desugar to SAX calls.

messages as needed. For example, the XML sugar library contains a compile-time check that verifies that all XML elements have equal start and end tags. Consequently, an element with mismatching tags is detected at compile time and leads to a compiler error as expected. To support domain-specific analyses, the SugarJ compiler applies context-sensitive checks before desugaring a program.

When developing the XML sugar library, we heavily reused other sugar libraries at metalevel in nontrivial ways, including the library for concrete syntax from the previous subsection. The diagram in Figure 2.9 depicts the structure and dependencies of the components involved in embedding XML. Package xml contains three sugar declarations. XmlSyntax defines the abstract and concrete syntax of XML, which is embedded into the syntax of Java by XmlJavaSyntax. AsSax defines how to desugar an XML document into a sequence of SAX library calls. Since XML documents are integrated into Java at expression level but the SAX library is accessed via statements, calls to SAX have to be lifted from expression level to statement level. To this end, we adopted the use of expression blocks EBlock from MetaBorg [BV04]. Accordingly, AsSax uses these expression blocks and concrete syntax to generate Java code.

Evidently, composing and reusing language extensions is essential in the implementation of XML. Since in SugarJ the primary means of organizing language extensions and DSLs are libraries, programmers can import sugar libraries to build their DSL or language extension on top of existing ones. For example, in the implementation of AsSax, we desugar XML trees into Java with expression blocks. The concrete syntax of expression blocks is directly available in desugaring rules, even though the support for concrete syntax in transformations was defined independently in concretesyntax.Java. This is possible because both sugar libraries extend the same Java nonterminals imported from

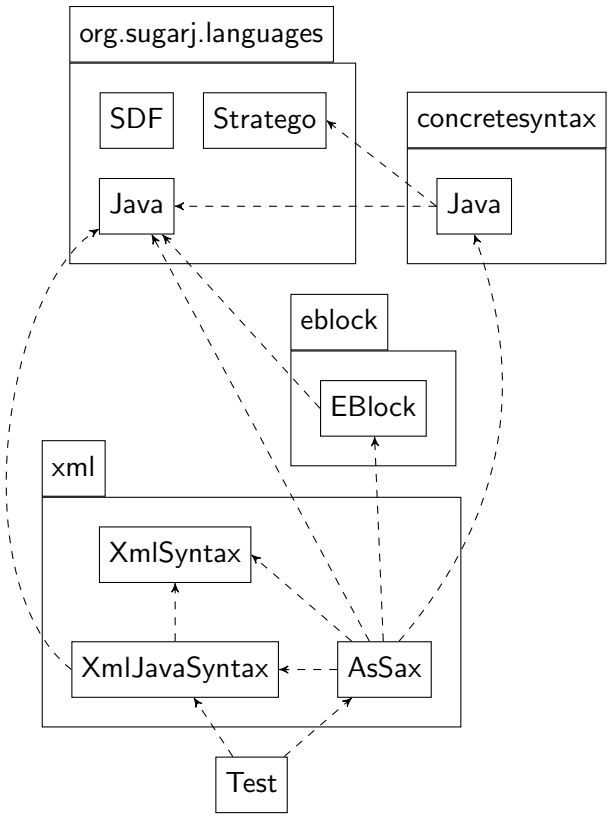


Figure 2.9: The structure of the XML case study: Arrows depict dependencies between sugar libraries and are resolved through sugar library imports.

Java. However, like for ordinary libraries, in general, it might be necessary to write *glue code* to compose individual sugar libraries meaningfully.

The XML case study illustrates how sugar libraries can be composed to make joint use of distinct syntactic extensions. It is important to note that the embedding of XML is not the end of the line of extensibility but itself a sugar library that can be used to build further language extensions. We demonstrate this feature in the following case study, where we implement a type system for XML as a sugar library.

2.5.3 XML Schema

A meta-DSL is a DSL with which one can define other DSLs. The definition of meta-DSLs is natural in SugarJ since SugarJ enables syntactic extensions of the metalanguage and the object language uniformly. Sugar libraries can thus provide new frontends for building other sugar libraries without any limitation on the number of metalevels

involved. To exemplify this, we have embedded XML Schema [W3C04] declarations into SugarJ as a sugar library for *validating* XML documents. Each concrete XML Schema specification stipulates a DSL of valid XML documents; a language of XML specifications is a meta-DSL.

To validate XML documents through the compiler, we have integrated a subset of XML Schema into SugarJ as a sugar library. As shown in Figure 2.10(a), a programmer can define an XML schema using a top-level **xmlschema** declaration that contains a conventional XML Schema document.[3] A programmer can require the validation of an XML document by annotating it with @Validate, as we illustrate in Figure 2.10(b). During compilation, the XML schema of the corresponding namespace traverses the XML document to check its validity and generate a (possibly empty) list of error messages.

Technically, we have defined a program transformation that desugars an XML schema into transformation rules for validating XML documents. An XML Schema element declaration

```
<xsd:element name="book" type="BookType" />
```

for example, desugars into a program transformation that matches on XML elements book and checks whether their attributes and children conform to BookType. According to the structure of an XML schema, validation rules like this one are composed to form a full validation procedure for matching XML documents and collecting possible errors. The XML Schema sugar library tries to validate an XML document against any validation procedure that is in scope. The sugar library issues a corresponding error message if no schema exist for the XML document's namespace.

The XML Schema case study not only demonstrates SugarJ's support for compile-time checks, but moreover its self-applicability support: The sugar library introduces syntactic sugar (XML Schema declarations) for the specification of metaprograms. This support of applying SugarJ to itself allows programmers to build meta-DSLs.

SugarJ's extensive support for self-application was also helpful in our implementation of the XML Schema sugar library itself. Although standard XML Schema cannot describe itself in general [MS06], we identified a self-describable subset of the language. This allowed us to bootstrap the sugar library for XML Schema declarations from a description of its syntax as an XML Schema declaration.

In summary, we have presented five case studies showing the expressiveness and applicability of SugarJ for implementing language extensions and syntactically embedding DSLs. Especially the more complex sugar libraries reuse simpler libraries, and with XML Schema we demonstrate SugarJ's flexibility as well as the benefits of context-sensitive checks and self-application.

[3] For simplicity, we currently do not support namespace abbreviations `xmlns:abc="xyz"` that enable the more conventional notation `<abc:node />`. However, this feature is syntactic sugar and can be implemented in an additional sugar library.

```
import xml.schema.XmlSchema;

public xmlschema BookSchema {
  <xsd:schema  targetNamespace="lib">
    <xsd:element name="book" type="BookType" />

    <xsd:complexType name="BookType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="author" type="Person" />
        <xsd:element name="editions" type="Editions" />
      </xsd:choice>
      <xsd:attribute name="title" type="string" />
    </xsd:complexType>

    <!-- more schema content here -->
  </xsd:schema>
}
```

(a) Definition of an XML schema for the namespace lib.

```
import xml.XmlJavaSyntax;
import xml.AsSax;
import BookSchema;

public void genXML(ContentHandler ch) {
  @Validate
  ch.<lib:book title="Sweetness and Power">
       <lib:author name="Sidney W. Mintz" />
    </lib:book>;
}
```

(b) SugarJ statically validates XML documents when validation is required by the @Validate annotation. To relate XML elements to their schema definition, element names are qualified by namespaces, here lib.

Figure 2.10: Definition and application of an XML schema.

2.6 Discussion and future work

In the present section, we discuss SugarJ's current standing, its limitations, and possible future development with respect to language composability, context-sensitive checks, tool support, and a formal consolidation.

2.6.1 Language composability

Composing languages with SugarJ is very simple because it only involves importing libraries. However, when composing multiple DSLs, ambiguities can arise in composed grammars and composed desugaring rules, or additional glue code might be necessary to integrate both languages more carefully (introduce intended interactions and prevent accidental interactions).

Nonetheless, when composing language extensions, our experience with SugarJ suggests that ambiguity problems do not occur frequently in practice or are easily resolvable. For instance, no composition problems arise in the case studies presented in the previous sections. In Chapter 7, we study language composability in depth and compare the performance of existing approaches to domain abstraction. Here, we discuss the problem from a more explorative viewpoint.

In general, the composition of grammars may cause conflicts, which manifest as parse ambiguities at compile time. For instance, when composing our XML sugar library with a library for HTML documents, the parser will recognize a syntactic ambiguity in the following program, because the generated document could be part of either language:

```
import Xml;
import Html;

public void genDocs(ContentHandler ch) {
  ch.<book title="Sweetness and Power">
        <author name="Sidney W. Mintz" />
     </book>;
}
```

It is always possible to resolve parse ambiguities without changing the composed sugar libraries: Besides using one of the predefined disambiguation mechanisms provided by SDF [vdBSVV02], one can add an additional syntax rule which allows the user to write, say, ch.xml<...> or ch.html<...> to resolve the ambiguity. This is similar to using fully-qualified names to avoid name clashes.

Another potential composition problem arises when importing multiple desugarings for the same extended syntax. Currently, the compiler does not detect the resulting conflict in the desugaring rules but selects one rule for application. This may lead to unexpected compile-time errors during desugaring or, worse yet, to generated code that ill-behaves

at run time. However, we believe that conflicting desugaring rules are not a practical problem for syntactic sugar and DSL embedding, since usually each DSL comes with its own syntax and hence desugaring rules do not overlap.

That said, detecting syntactic and semantic ambiguities or conflicts is an interesting research topic, related to detecting feature interactions [CKMRM03]. Although not in the scope of this work, in future work, we plan to evaluate existing technologies for detecting ambiguities in grammars and program transformations. For example, we want to investigate the applicability of Axelsson et al.'s encoding of context-free grammars as propositional formulas, which allows the application of SAT solving to verify efficiently the absence of ambiguous words up to a certain length, but may fail to terminate in the general case [AHL08]. Alternatively, Schmitz proposed a terminating algorithm that conservatively approximates ambiguity detection for grammars and generalizes on the ambiguity check build into standard LR parse table construction algorithms [Sch07]. For the detection of conflicting desugaring rules, we want to assess the practicability of applying critical pair analysis to prohibit all critical pairs—even joinable ones—reachable from the entry points of desugaring. This idea has previously been applied for detecting conflicts in program refactorings [MTR05]. To rule out fewer critical pairs, we could combine critical pair analysis with automatic confluence verification [AYT09] to determine the joinability of critical pairs.

Since SugarJ treats the host language and the metalanguage uniformly, all of these ambiguity checks could be implemented as metalanguage compile-time checks in SugarJ. However, these checks operate on the fully desugared base language, whereas SugarJ performs checking before desugaring. Thus, SugarJ would need to support more fine-grained control over *when* checks are executed.

2.6.2 Expressiveness of compile-time checks

Sugar libraries support checking programs for syntactic and semantic correctness: Each syntactic extension specifies what correctness means in terms of a context-free grammar and compile-time assertions. During parsing, conformance to an extension's grammar is checked. For example, we ensure matching brackets in our pair and closure DSLs.

Context-sensitive properties occur, for example, in context-sensitive languages or statically typed DSLs. For context-sensitive properties the question arises when to check them: before, during, or after desugaring.

In addition to encoding constraints as part of desugaring rules, our current implementation of SugarJ also offers initial support for a more direct implementation of error reporting: Sugar libraries can specify a Stratego transformation which transforms the syntax tree prior to desugaring into a list of error messages. This approach enables the definition of context-sensitive properties in terms of surface syntax and comprises pluggable type systems [Bra04]. For instance, the check for matching start and end tags of XML documents and XML Schema validation is naturally specified in terms of XML

syntax.

However, performing static analyses before desugaring restricts the extensibility of compile-time checks. Consider, for example, a syntactic extension that introduces JavaScript Object Notation (JSON) syntax as an alternative syntax for describing tree-structured data, which desugars to XML code:

```
{
  "book": {
    "title" : "Sweetness and Power",
    "author" : { "name" : "Sidney W. Mintz" }
  }
}
```

Even though this code desugars to XML code eventually, our current implementation of XML Schema validation fails to process the JSON document before desugaring, because the validation can match on XML documents. To reuse XML Schema validation for JSON, we require some interleaving of compile-time checking and desugaring to enable compile-time checks not only on nondesugared surface syntax, but also on desugared base language syntax and intermediate stages of desugaring. To this end, in future work, we would like to investigate the applicability of a constraint system that separates constraint generation from constraint resolution and performs both interleaved with desugaring. We plan to let constraints keep track of the actually performed desugarings, so that constraint verification does not interfere with the application of desugarings.

2.6.3 Tool support

In order to efficiently develop software in the large, error reporting, debugging and other IDE support is essential [Fow05b, KV10, RGN10]. Due to the fluent change of syntax, and thus language, sugar libraries place extraordinary challenges on tools: all language-dependent components of an IDE depend on the sugar libraries in scope. Consider syntax highlighting, for example, in which keywords are colored or highlighted in a bold font. Since syntactic extensions can introduce new keywords to the host language, syntax highlighting needs to take sugar-library imports into account. In fact, we have been working on an integration of SugarJ and Spoofax [KV10], which we describe in the subsequent Chapter 3. In a nutshell, we implement domain-specific editor services in *editor libraries*, which in conjunction with a language's sugar library supplies the necessary information for providing advanced editor services in a library-centric fashion.

2.6.4 Core language

In the study of sugar libraries, we used SugarJ to evaluate the expressiveness and applicability of our approach, for instance, by developing complex case studies such as

XML Schema. However, it would be interesting to formally consolidate sugar libraries and study them more fundamentally.

One aspect we intend to study is the relation between syntactic extensions and scopes. It is not obvious how to support sugar libraries in languages that allow "local" import statements such as in Scala or ML. For example, consider the following program, in which we assume s1 after s2 to desugar to s2; s1, that is, to swap the order of the statements s1 and s2.

```
(17,"seventeen") after import pair.Sugar;
```

After swapping the two statements, the scope of the import of pair.Sugar includes (17,"seventeen"), which, thus, is a syntactically valid expression. However, to parse a program of the form s1 after s2, the parser already requires knowledge of how to parse (17,"seventeen") before it can even consider parsing **import** pair.PairSugar; this is a paradox.

Another interesting aspect of such core language is to identify the minimal components of a syntactically extensible language such that a full language like SugarJ can be boot-strapped from this core language.

2.6.5 Module system

The semantics of imports in SugarJ is intended to closely match the semantics of imports in Java. In our proof-of-concept implementation, however, imports are split into Java, SDF and Stratego by reproducing them in the respective syntax. Unfortunately, though, the scoping rules of these languages differ: Imports are transitive in Stratego and SDF but nontransitive in Java. Therefore, in the current implementation of SugarJ, if A imports syntactic sugar from B, which in turn imports syntactic sugar from C, the syntactic sugar from C will be available in A. In contrast, A cannot access Java declarations from C without first importing C or using fully qualified names. We plan to investigate whether this mismatch can be resolved using systematic renaming.

Java, the base language for SugarJ, has a rather simple module system in which the interface of a library is often rather implicit because users of a library just import the library's implementation.

In future work, we would like to make syntactic extensions a formal part of a dedicated interface description language. In this context, we want to address also the question of whether there should be some kind of abstraction barrier in an interface that hides the details of the desugaring of a syntactic extension. In the current SugarJ programming model, a programmer has to understand the associated desugaring to reason about, say, the well-typedness of a program written in extended syntax. Hence the desugaring rules must be part of the interface. We believe that this is acceptable as long as transformations are simple and compositional—which typically is the case for syntactic sugar. However, for more sophisticated transformations, it makes sense to have an abstraction mechanism

that hides the details of the transformation, yet allows programmers to reason about their code in terms of the interface.

2.7 Chapter summary

We introduced sugar libraries as a mechanism to extend a host language with domain-specific syntax while preserving modular reasoning. Developers can import syntax extensions and their desugaring as libraries, for instance, to develop statically checked domain-specific programs. Sugar libraries preserve the look-and-feel of conventional libraries and facilitate composability and reuse: A developer may flexibly select from multiple syntactic extensions and import and combine them, and a library developer may reuse sugar libraries when developing other sugar libraries (even in a self-applicable fashion). Composition conflicts can occur, but we believe that they are rare in practice. Nevertheless, we would like to have better support for avoiding (by better scoping constructs) and detecting (by better analyses) composition conflicts statically.

To demonstrate flexibility and expressiveness, we have implemented sugar libraries in the Java-based language SugarJ. With SugarJ, we have implemented five case studies with growing complexity: pairs, closures, XML, concrete syntax for transformations, and XML Schema. The latter of these case studies heavily reuse syntax extensions imported from former and the last one implements a meta-DSL for which self-applicability is a significant advantage. In contrast to many other metaprogramming systems, a SugarJ programmer never has to reason outside the language since SugarJ comprises full metaprogramming facilities. In conclusion, sugar libraries are both flexible and principled devices for syntactic domain abstraction.

3 Integrated Development Environments for Extensible Languages

This chapter shares material with the GPCE'11 paper "Growing a Language Environment with Editor Libraries" [EKR+11a].

Large software projects consist of code written in a multitude of different (possibly domain-specific) languages, which are often deeply interspersed even in single files. While many proposals exist on how to integrate languages semantically and syntactically, the question of how to support this scenario in integrated development environments (IDEs) remains open: How can standard IDE services, such as syntax highlighting, outlining, or reference resolving, be provided in an extensible and compositional way, such that an open mix of languages is supported in a single file?

Based on SugarJ, our library-based extensible language for Java (Chapter 2), we propose to make IDEs extensible by organizing editor services in *editor libraries*. Editor libraries are libraries written in the host language, SugarJ, and hence activated and composed through regular import statements on a file-by-file basis. We have implemented an IDE for editor libraries on top of SugarJ and the Eclipse-based Spoofax language workbench [KV10]. We have validated editor libraries by evolving this IDE into a full-fledged and schema-aware XML editor as well as an extensible LaTeX editor.

3.1 Introduction

Extensible programming languages are an old research topic that has gained new relevance by the trend toward DSLs and the vision of language-oriented programming [War95, Dmi04, Fow05b]. Researchers have proposed a variety of different approaches to extend the syntax and semantics of languages and to embed languages in other languages, such as libraries [Hud98, THSAC+11], extensible compilers [EH07a, NCM03, VKBS07], macro systems [BP01, BS02, THSAC+11, Tra08], and metaobject protocols [RGN10, TCKI00]. However, while languages themselves have gained flexibility, tool support in the form of integrated development environments (IDEs) cannot keep up with the rapid development and composition of new languages.

IDEs assist programmers, who spend a significant amount of time reading, navigating, adapting, and writing source code. They provide *editor services* that improve a program's layout and support programmers in performing changes to the code, including syntax highlighting, code folding, outlining, reference resolving, error marking, quick fix proposals,

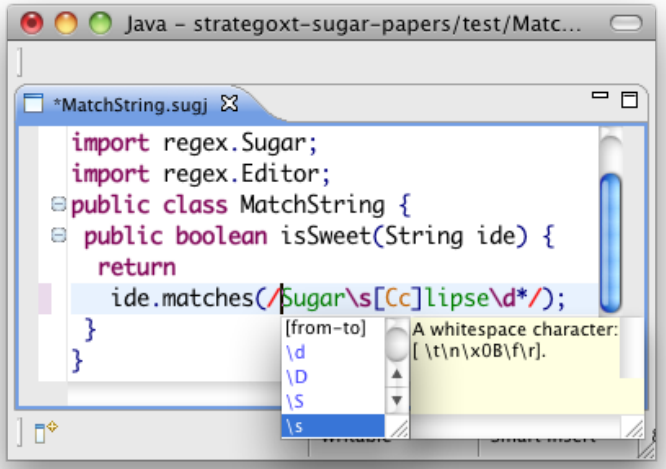


Figure 3.1: Alongside the sugar librariy regex.Sugar that provides a syntactic extension for regular expressions, we import the editor library regex.Editor that provides a corresponding IDE extension for regular expressions.

code completion, and many more. The quality of IDE support for a language is a significant factor for the productivity of developers in that language. Therefore, it is desirable to provide the same level of tool support for extended and DSLs that programmers are familiar with from mainstream programming languages.

However, as our own and the experience of others show, developing tool support for a new or extended language requires significant effort [Cha06, MO06, KTS+09]. Although there are several advances to generate tool support from declarative specifications [KV10, EV06], generation has to be repeated for every combination of language extensions because the generated editor services neither compose nor grow with the language.

Composable and growable editor services are especially important in the context of growable languages [Ste99, BS02, ACN+09] that support flexible and composable language extensions, e.g., for the embedding of multiple DSLs. In the previous chapter, we presented SugarJ, a variant of Java which is extensible via sugar libraries. A sugar library exports, in addition to ordinary types and methods, a syntactic extension and a transformation from the extended syntax back into the syntax of the host language. Sugar libraries are imported via the usual import mechanism of the host language. Multiple syntactic extensions can be composed by importing them into the same file, allowing a local mix of multiple embedded languages.

In this chapter, we present *editor libraries* that generalize library-based extensibility

towards IDEs. Editor libraries compose: When multiple languages are mixed within the same file (such as XML, SQL, and regular expressions within Java), we import and thereby combine all corresponding editor services. Editor libraries (as other libraries) are self-applicable, that is, editor libraries can be used to develop other editor libraries. Furthermore, editor libraries encourage a generative approach through *staging*: We generate editor services from high-level specifications (yet another DSL) at one stage and use the generated services at a later stage. Staging enables the coordination of editor services that span several source files or languages.

We have developed an Eclipse-based IDE with support for editor libraries called the *SugarJ IDE*. For each file, the SugarJ IDE considers all editor libraries in scope, interprets the associated editor services and presents the decorated source code and editing facilities to the programmer. The SugarJ IDE is based on the *Spoofax* language workbench [KV10], which supports the generation and dynamic reloading of Eclipse-based language-specific editors from declarative editor configurations. In Figure 3.1, we illustrate an example usage of the SugarJ IDE: The import of regex.Sugar activates a syntactic extension for regular expressions, which integrates regular expression syntax into the surrounding Java syntax (instead of the usual string encoding). The import of the editor library regex.Editor enables corresponding editor services for regular expressions such as syntax coloring and code completion. The SugarJ IDE automatically composes the editor services of the host language, here Java, with the editor services of the extension, here regular expressions, to provide uniform IDE support to the programmer. While our SugarJ IDE and this chapter focus on editor libraries for SugarJ, the concept of editor libraries is similarly useful for embedded languages in syntactically less flexible languages (cf. Section 3.7).

With several case studies, we demonstrate the practicality of editor libraries and the power of their composability. Beyond small editor libraries such as regular expressions illustrated above, we implemented full-fledged editor libraries for XML (including XML Schema) and Latex. We used the latter for writing a conference article [EKR+11a] on the subject of the present chapter.

We present the following contributions:

- We introduce the novel concept of *editor libraries* for organizing IDE extensions in libraries of the host language, in particular, to provide IDE support for embedded DSLs. This addresses our design goal on domain-specific editor services.
- Editor libraries are activated using the host language's standard import mechanism, and editor libraries *compose* to support multiple DSLs and the host language in a single file.
- We describe a pattern of *editor-library staging* to generate editor services from high-level specifications and to coordinate editor services between several source files or languages.
- We present SugarJ IDE, an extensible IDE for SugarJ based on the Spoofax language workbench. Our growable IDE complements the syntactic extensibility of SugarJ

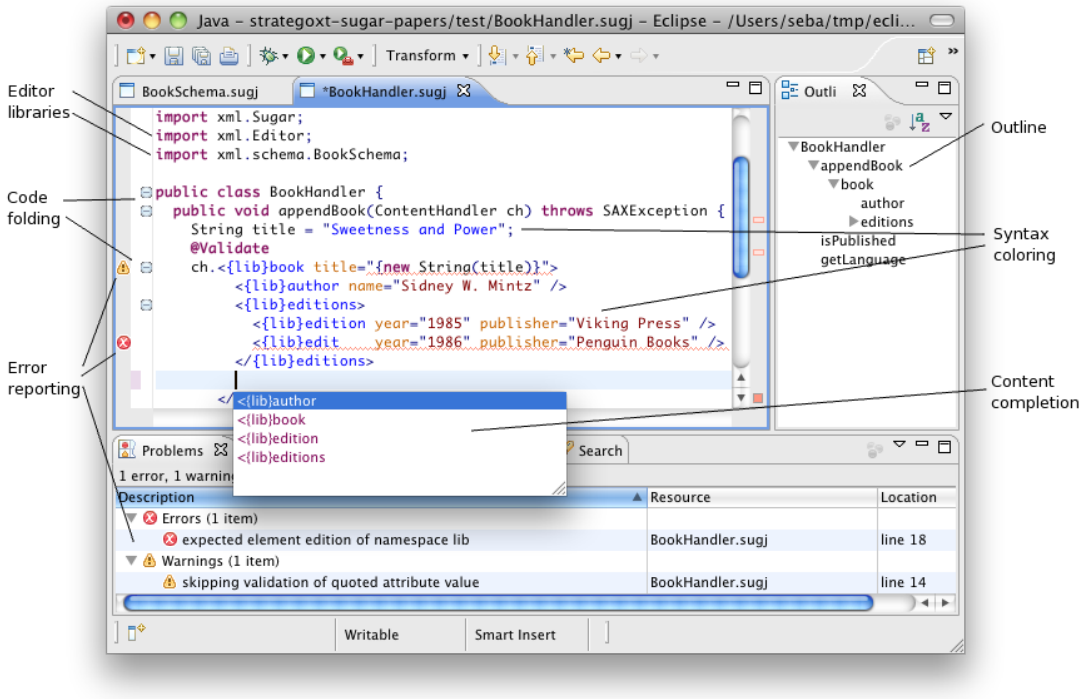


Figure 3.2: An XML embedding shown in the SugarJ IDE. The imported editor libraries extend the SugarJ IDE and compose with its basic Java editor services.

with the capability of visualizing the result of domain-specific static analyses and providing domain-specific editor services that conform to the embedded DSLs.

- We validate our approach through realistic *case studies* of full-fledged editors for XML and Latex. We demonstrate how our IDE supports domain-specific and programmer-defined editor configuration languages as well as deriving editor services from language specifications.

3.2 An overview of the SugarJ IDE

The SugarJ IDE, as shown in Figure 3.2, consists of an editor that features services such as syntax coloring, error marking and code completion. The SugarJ IDE has built-in support for Java syntax only, but all of the SugarJ IDE's editor services are user-extensible: Additional syntax and editor services can be imported from libraries.

3.2.1 Using the SugarJ IDE

A user of the SugarJ IDE activates editor support for an additional language by importing a corresponding editor library. For example, in Figure 3.2, the sugar library xml.Sugar provides a grammar for embedded XML documents, and the editor library xml.Editor provides editor services for XML. This editor library specifies syntax coloring, outlining, code folding, and more for embedded XML documents without invalidating the built-in services for Java. For example, the resulting editor contains code folding and outlining for both Java and XML combined. The additional editor support only affects the XML part of the document and leaves the remaining editor support intact. This is most visible in Figure 3.2 from the nested syntax highlighting, including correct highlighting of the quoted Java expression **new** String(title) nested inside XML.

We can further extend editor support for XML if we know the XML schema that the document adheres to. Given a document's schema, the SugarJ IDE provides even more domain-specific editor services for the embedded XML document, including error reporting for validation errors and content completion, which provides a list of all valid tags. To activate the additional editor support, the user imports the editor library xml.schema.BookSchema, which is specified by a concrete XML schema for books.

3.2.2 Editor services

A user of the SugarJ IDE can also assume the role of editor-service developer, because editor services are specified declaratively within the SugarJ IDE. This is more expressive than setting options in the Eclipse menu and significantly easier than manually extending Eclipse by writing a corresponding plugin. In addition to error marking, the SugarJ IDE lifts and extends eight different editor services from Spoofax [KV10]. Each service can be declaratively specified in a DSL.

- *Syntax coloring* highlights source code using a colored, bold or italic font.
- *Code folding* supports collapsing part of the source code to hide its details.
- *Outlining* gives a hierarchical overview over the current document and enables fast navigation.
- *Content completion* provides proposals for complementing the current source code.
- *Reference resolving* resolves a construct (typically a name) to its declaration and provides facilities to navigate to the declaration directly ("CTRL-click").
- *Hover help* displays documentation as a tooltip when hovering over a documented entity with the mouse.
- A *refactoring or projection* applies a transformation to (parts of) the source code and writes the result either in the original or a separate file.

```
package xml;

import editor.Colors;
import xml.XmlSyntax;

public editor services Editor {
  colorer
    ElemName  :  blue        (recursive)
    AttrName  :  darkorange  (recursive)
    AttValue  :  darkred     (recursive)
    CharData  :  black       (recursive)

  folding
    Element

  outliner
    Element
}
```

Figure 3.3: Editor library for coloring, folding and outlining of XML code.

- *Parentheses matching* marks matching parentheses in the source code and adds closing parentheses automatically. This service is also essential for *automatic indentation* after line breaks.

Conceptually, editor services can be understood as procedures that decorate syntax trees, for example, with coloring information. The SugarJ IDE then interprets these decorated trees and maps the decorations to the original source code or other means of visualization such as a separate outline window or a completion proposal viewer. Since editor services are mere tree decorators, their definitions are fairly simple in most cases (the definition of refactorings and projections being an exception). To reflect this simplicity in editor service implementations, we use an extended version of the declarative editor-service configuration language provided by Spoofax [KV10].

Developers can bundle multiple editor-service specifications in an editor library declared as a top-level **public editor services** entity. For example, the xml.Editor library shown in Figure 3.3 provides editor services for coloring, folding and outlining XML documents using declarative tree decoration rules. Each tree decoration rule specifies a syntax-tree pattern to match against and the decoration to apply to matched trees. For example, the XML coloring rules match on trees of the kind ElemName, AttrName, AttValue and CharData, that is, trees derived from these non-terminal sorts as defined by the imported

sugar library xml.XmlSyntax. The coloring rules thus declare that XML element names are shown in a blue font, XML attribute names in a dark orange font, etc., and that the coloring recursively applies to all nodes in the matched trees. Similarly, the folding and outlining services declare that XML elements are foldable and XML documents show up in the outline of source files.

We specifically support the development of editor libraries by providing as part of our SugarJ IDE an editor library for writing editor libraries. In similar fashion, we encourage other developers of language embeddings to accompany their embeddings with editor support in the form of editor libraries.

3.3 Editor libraries

Editor libraries provide a principled means for organizing, scoping, and activating editor services. Before discussing the composability of editor libraries in detail, we describe a number of advanced usage patterns for editor libraries in SugarJ.

3.3.1 Domain-specific editor configuration languages

SugarJ supports syntactic abstraction over all of its ingredients, that is, Java code, syntactic sugar, static analysis specifications, and, now as well, editor configurations. This design enables the development of customized and domain-specific editor-service configuration languages. For example, we have applied SugarJ's syntactic extensibility to provide an XML-specific editor service configuration syntax in the style of Cascading Style Sheets (CSS):

```
import editor.Colors;
import xml.CSS;
import xml.XmlSyntax;

public css CSSEditor {
  Element     { folding; outlining }
  ElemName    { rec-color : blue }
  AttrName    { rec-color : darkorange }
  AttValue    { rec-color : darkred }
  CharData    { rec-color : black }
}
```

This CSS-style editor configuration corresponds and, in fact, desugars to the editor configuration in standard editor service syntax shown in Figure 3.3. CSS is just another syntax for specifying editor services.

```
import xml.schema.XmlSchema;

public xmlschema BookSchema {
  <xsd:schema  targetNamespace="lib">
    <xsd:element name="book" type="BookType" />

    <xsd:complexType name="BookType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="author" type="Person" />
        <xsd:element name="editions" type="Editions" />
      </xsd:choice>
      <xsd:attribute name="title" type="string" />
    </xsd:complexType>

    <!-- more schema content here -->
  </xsd:schema>
}
```

Figure 3.4: An excerpt of the Book XML Schema. The xml.schema.XmlSchema library provides validation and editor services for XML schemas themselves.

3.3.2 Staged editor libraries

Many editor services are not static, but rather depend on the contents of the file being edited and imported files. For example, hover help for non-local Java methods depends on the method definitions in other files and code completion for XML elements depends on the corresponding schema. Hand-written IDEs support such editor services by managing a set of files as a project, explicitly coordinating between the information retrieved from each file. Unfortunately, neither SugarJ nor Spoofax has a notion of projects: In Spoofax, editor services for different files are independent, and in SugarJ, files are processed one after another. The SugarJ IDE, however, supports separate *generation and application stages* for editor libraries from different source files, which enables rich patterns of interaction between editor services of individual source files.

The central idea of our *staging pattern* is to first generate editor services from domain-specific declarations in one file and to later use them in another file. The generated editor services may well be of auxiliary nature such as a mapping from method names to the documentation of these methods, which a hover help editor service can query to display documentation of a method as a tooltip. In general, the SugarJ IDE employs the transformation language Stratego [VBT98] for auxiliary editor services, and an import statement brings the generated editor services into scope.

For example, we applied the staging pattern to promote XML schemas as domain-specific declarations of XML editor services that are specific to an XML dialect. Such editor services include XML validation and tag completion. Figure 3.4 shows an excerpt of the Book XML schema, which declares a dialect of XML for describing books. From this schema, we generate the definition of a static analysis as well as code completion. For the former, we desugar an XML schema into a set of Stratego rules that traverse a given XML document to check whether this document conforms to the schema. In other words, we generate a type checker for each XML schema. The result of applying the XML Book type checker is shown in Figure 3.2, where quoted Java expressions within an XML document are marked but ignored otherwise. Furthermore, our XML Schema embedding desugars each schema into a set of schema-specific completion templates. For instance, the following completion template results from desugaring the above Book schema.

```
completion template : Content =
  "<{lib}book title=\"" <string> "\">"
  "</{lib}book>"
```

When the parser expects XML Content, this completion template proposes a book element with a title attribute to the programmer. Accordingly, when importing the Book schema, the SugarJ IDE recognizes the accompanying editor services and provides code completion to the programmer as shown in Figure 3.2.

As this case study illustrates, the SugarJ IDE supports the implementation of editor services that involve multiple files using a generative approach; the staging pattern effectively facilitates data flow from one source file to another. In this example, we modeled data flow from an XML Schema declaration to clients of the schema, by generating completion templates. We present a more advanced example in Section 3.6.2, where we model data flow from a Bibtex bibliography to a Latex file that cites bibliography entries.

3.3.3 Self-applicability

Like conventional libraries, editor libraries are self-applicable, that is, editor services can be used during the development of other editor libraries. For example, we have implemented code completion for the code completion editor service using an editor library:

```
public editor services Editor {
  completions
    completion template : EditorServiceCompletionRule =
      "completion template" " : " <Sort> " =\n\t"
        "\"" <prefix> "\" <" <placeholder> ">"
}
```

This template provides content completion for completion templates themselves. Completion templates are represented as sequences of strings and placeholders such as <Sort>, which the SugarJ IDE marks for the user to replace. The above completion template expands into the following code on selection, where the underlined fragments are placeholders:

```
completion template : Sort =
  "prefix" <placeholder>
```

More generally, we provide full editor support for writing editor libraries in the SugarJ IDE using editor libraries.

3.4 Editor composition

A key feature of the SugarJ IDE is the ability to compose editor libraries. For example, we can import editor libraries for regular expressions and XML in the same document. The IDE then supports both language extensions with corresponding syntax highlighting and other facilities. Editor libraries cooperate to present a coherent user interface even though their respective authors might not have anticipated the exact combination of editor libraries.

We can compose editor libraries developed independently, such as regular expressions and XML, but we can also develop editor libraries that extend other libraries and editor libraries that explicitly interact with other editor libraries through extension points. Let us illustrate such interaction with an example from the domain of text documents (which we describe in more detail in Sec. 3.6.2): We express a bibliography database in one language (e.g., Bibtex-like) and write the text with references to bibliography entries in another language (e.g, Latex-like). When composing both languages, we would like to add editor services to navigate from bibliography references to their definitions, to suggest available references with content completion, to provide hover help, and so forth. These editor services need to bridge elements in different files and from different languages.

Although different kinds of interactions and even conflicts between editor services are possible, we argue that editor services are largely independent and have local effects. In addition, for many services, interactions can be implicitly resolved using generic strategies. Finally, for intended interactions as in the bibliography example, we apply the staging pattern for explicitly coordinating editor services.

3.4.1 Local variation and global consistency

Editor libraries extend the *local* behavior of the SugarJ editor. There are different notions of locality:

- Editor libraries are modular and affect only files that import them explicitly. In these files, only the part after the import is affected.

- Editor libraries that extend distinct editor services compose naturally. For example an editor library defining syntax coloring will not conflict with another editor service providing content completion.
- Editor libraries usually reason about small and local subtrees of the abstract syntax tree. For example, an editor library typically defines syntax highlighting for specific syntactic forms, not for the overall program, and editor libraries that accompany a DSL embedding reason over tree fragments of that DSL only. Editor libraries that act on different parts of the abstract syntax tree naturally compose. For example, the XML editor library shown in Figure 3.3 only decorates XML fragments of the syntax tree and does not affect Java fragments.

The *global* behavior of the SugarJ editor, however, is fixed and cannot be extended by editor libraries. For example, the SugarJ editor supports a fixed set of editor services such as syntax highlighting, reference resolving, hover help, etc. as discussed in Section 3.2.2. The SugarJ editor presents a coherent user interface to access these editor services. For example, key bindings or the visual appearance of error markers are defined by the SugarJ editor directly and are therefore consistent across error libraries.

Together, global consistency and local variation go a long way ensuring that the SugarJ IDE supports arbitrary languages while still providing a coherent user interface. Some interactions between editor libraries cannot be resolved by locality, however, and require implicit or explicit coordination between editor libraries.

3.4.2 Implicit coordination

Although most editor libraries work locally, their results can conflict or overlap. For most editor services, conflicts can be resolved implicitly following generic strategies: aggregation and closest match.

For many editor services, aggregating results of different editor libraries is sufficient. For example, in our XML embedding, both Java and XML code completion services would respond to a prefix `ch.`, which could be followed by a Java method name or an XML element. The SugarJ IDE simply shows all completion proposals. Aggregation works similarly for code folding, outlining and error marking.

For some other services, primarily syntax highlighting and hover help, simple heuristics can resolve conflicts implicitly. For example, when one editor library specifies that all tokens in assignments should be blue, whereas another editor library specifies that all tokens in while loops should be red, the SugarJ IDE needs to coordinate between these editor libraries and decide in which color to display tokens in an assignment nested within a while loop. As heuristic, we propose a closest-match rule, as used for style sheets in HTML: Color information, hover help, and other specifications on an AST node overrule corresponding specifications of the parent node; always the most specific information is used for presentation. For our example above, the closest-match rule displays the

assignment blue, because the match on assignments is more specific (closer to the tokens in question) than the match on the while loop.

Aggregation and the closed-match rule resolve many conflicts implicitly in a natural way. Explicit coordination is usually necessary only for intended interactions.

3.4.3 Explicit coordination

Not all editor libraries are supposed to be independent. Editor libraries might explicitly extend the behavior of other libraries or interact with them in controlled ways.

An editor library can add additional editor-service specifications to another library. For example, the XML-Schema library builds on top of the XML library and extends it with code completion and error checking. In addition, different editor libraries can interact explicitly through the staging pattern to share data and coordinate editor services. The staging pattern, described in Section 3.3.2, enables communication from one editor library to another through the generation of auxiliary editor services. In our example, the bibliography database shares information about all known entries by generating an auxiliary editor service (technically: Stratego rules) that maps entry names to their definitions:

```
bibtex-entry : "Hudak98" ->
  BibtexEntryStm(
    "@inproceedings",
    BibtexEntryName("Hudak98"),
    [ BibtexAttribute(BibtexAttributeName("author"), "Paul Hudak"),
      BibtexAttribute(
        BibtexAttributeName("title"),
        "Modular domain specific languages and tools"),
      BibtexAttributeUnwrapped(
        BibtexAttributeName("booktitle"),
        BibtexConstName("ICSR")),
      BibtexAttribute(BibtexAttributeName("year"), "1998"),
      BibtexAttribute(BibtexAttributeName("pages"), "134--142"),
      BibtexAttribute(BibtexAttributeName("publisher"), "IEEE")])
```

Auxiliary editor services are scoped via editor libraries. Accordingly, other editor library can use the information of an auxiliary editor service whenever the corresponding editor library is in scope. For example, our Latex editor library integrates with the bibliography editor library to supply hover help and content completion for citations (\cite{...}), and checks for undefined references by querying the auxiliary editor service bibtex-entry.

Technically, explicit coordination with auxiliary editor services relies on the self-applicability of SugarJ. We rely on the fact that SugarJ libraries can generate Stratego

code that is available at compile time of other libraries. Accordingly, one editor library can provide auxiliary editor services as Stratego rules to be used in other editor libraries.

3.4.4 Limitations

Although editor-library composition is usually straightforward in practice, there are limitations. Most significantly, we cannot provide modular guarantees about editor services in hostile environments.

Editor services use a global namespace without hiding. In principle, editor libraries could access (auxiliary) services of all other imported editor libraries and extend them. We discourage uncontrolled sharing and use naming conventions (similar to fully qualified names in Java) to avoid accidental name clashes. The staging-based communication between editor libraries relies on conventions and implementation patterns; there is no explicit scoping concept for staged services yet.

Furthermore, editor services should make little assumptions about the global structure of the AST. Editor services are used in a context where the AST of a file typically contains structures from different languages. For example, navigating from an AST element to its direct parent should be avoided, instead one should search for a direct or indirect parent of the expected type. Such strategies make editor libraries more robust against additional language extensions. However, the SugarJ IDE currently does not enforce locality and cannot detect violations modularly.

Building a module system to provide explicit namespaces and checked interfaces for the SugarJ IDE and the underlying SugarJ is an interesting avenue for future work. Such a module system should prevent name clashes and control what kind of information (technically: which Stratego rules) can be shared between editor libraries. To a large degree this seems to be a straightforward adoption of concepts from other module systems, such as the compilation manager in Standard ML [BA99]. On top, semantic interfaces could enable modular detection of conflicts between two editor libraries at link time.

In our experience, conflicts between editor libraries are rare and patterns for explicit coordination are easy to implement when required. Naming conventions and implementation patterns seem sufficient to avoid conflicts in practice. Hostile environments (deliberate attacks against editor libraries) are currently not a practical concern for editor extensions. Our SugarJ IDE appears useful for many practical tasks, even without modular guarantees. We revisit the composability of domain-specific editor services in Chapter 7, where study language composition in a broader context.

3.5 Technical realization

In the SugarJ IDE, we combine the sugar libraries of SugarJ (Chapter 2) with the IDE foundation of Spoofax [KV10] to support editor libraries for growing an IDE. SugarJ parses a file incrementally, because each declaration can extend the grammar of the

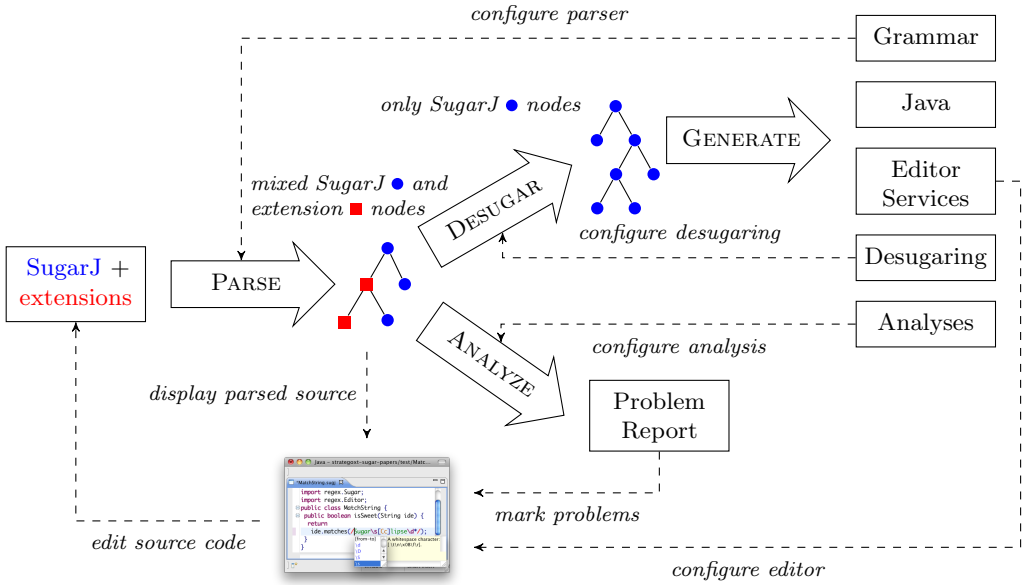


Figure 3.5: Data flow in the SugarJ IDE. The results of the processing pipeline (⟹) are used to configure (- - ➝) the earlier stages.

rest of the file, and like in Spoofax, we use a generic editor component which can be configured to support different languages. The SugarJ IDE adds editor libraries into the mix: Sugar libraries can desugar source code into editor libraries, and editor libraries in scope reconfigure the editor while a source file is edited. Together, these components enable to grow the IDE with editor libraries.

3.5.1 Architecture

Source code documents are often processed in many stages, compile time and run time traditionally being the most well-known. A library can affect several of these stages. For example, a Java class library contains, among other things, type definitions and method bodies. Clients of the library are type-checked against the type definitions in the library at compile time, but method calls to method definitions in the library are executed at run time. In our previous work on sugar libraries in SugarJ, we have broadened the applicability of libraries by considering additional stages: parsing, desugaring, and analysis. Sugar libraries contain grammar or desugaring rules to affect these stages of the SugarJ implementation. In the present work on editor libraries, we consider an integrated development environment as an integral part of the language implementation, that is, we consider an additional editor stage, which can be affected by editor libraries.

The interaction of these stages in the SugarJ IDE is shown in Figure 3.5. The diagram

extends Figure 2.7 from Chapter 2 with stages for the editor and analysis. The editor stage is depicted by the SugarJ IDE screenshot, all other stages are depicted as block arrows ($\Longrightarrow$). The parsing stage transforms a source-code document into an heterogeneous abstract syntax tree with nodes from different language extensions. The desugaring stage expands all nodes corresponding to language extensions into nodes of the base language, and the generation stage transforms the resulting homogeneous abstract syntax tree into separate source code artefacts containing grammar extensions, desugaring rules, editor services, and so on. At the same time, the analysis stage checks the heterogeneous abstract syntax tree and produces a problem report listing all found errors and warnings.

The results of compilation can configure earlier stages as depicted with dashed arrows ($\dashrightarrow$) in Figure 3.5. For example, generated grammars configure the parsing stage for clients of a sugar library and the generated analyses are applied in the analysis stage. In addition to these stages, the results of compilation also configure the editor, as we detail in the following subsections. In particular, the editor displays the input file's content with syntax highlighting according to the parsed source code, marks problems found by the analysis stage and behaves according to the editor services currently in scope. When the programmer changes code in the editor, the processing pipeline is run again to produce updated grammars, desugarings, etc., and any changes in these artifacts are reflected in the various stages.

3.5.2 Incremental parsing

Our SugarJ IDE supports languages with extensible syntax by relying on SugarJ for incremental parsing. Parsing with SugarJ is an incremental process because import declarations and syntax definitions can change the syntax for the rest of the file. To this end, SugarJ repeatedly parses a single top-level entity (e.g., import or class declaration) followed by the remainder of the file as a string. For each such parse, SugarJ extends the grammar according to the parsed entity before continuing to parse the remainder of the file. See Section 2.4 for details.

In the context of the SugarJ IDE, two additional concerns arise. First, the parser must associate every node of the abstract syntax tree with position information which the editor needs for marking errors, moving the cursor for reference resolving or outline view navigation, and so on. Second, the parser must associate some nodes of the abstract syntax tree with tokens that are used for syntax highlighting.

To reconcile incremental parsing of SugarJ with creating tokens and collecting position information, we use the same tokenizer for each parse. After each parse, we partially retract the tokenizer to ignore all tokens after the top-level entity and to reset the parser position accordingly. After parsing, we combine the trees of all top-level entities and ensure that the tree nodes have pointers to corresponding tokens and position information.

3.5.3 Dynamic loading of editor services

The SugarJ IDE supports editor libraries by relying on Spoofax to provide a generic Eclipse-based editor which can dynamically load and reload editor services. Although Spoofax still distinguishes the building and loading of editor services into separate phases, its dynamic loading capability forms the basis for editor services that are transparently built and loaded with library imports in the SugarJ IDE.

In the context of the SugarJ IDE, two additional concerns arise. First, parse tables and editor services need to be adapted on-the-fly whenever the corresponding language or editor libraries change. This is accomplished by running the full processing pipeline whenever a file has been changed and needs to be reparsed. The editor then dynamically reloads the possibly regenerated editor services. To ensure optimal responsiveness of the editor, generation and reloading happens in a background thread. Any services that were already loaded and parse tables that were already built are cached. Second, in the SugarJ IDE, each file determines the required language components and editor components by means of library imports. The SugarJ IDE therefore needs to maintain a separate set of editor services for each file. In contrast, Spoofax normally uses a language-level factory class. We subclass that factory with a specialized implementation that loads editor services in a file-specific fashion.

To conclude, in the present section we presented the architecture of our SugarJ IDE, which augments SugarJ's processing pipeline with an additional editor stage that can be configured via editor libraries. The editor stage connects to the processing pipeline through presenting the parsed syntax tree, marking errors and loading the (possibly staged) editor services. The following section reports on experiments with this realization of the SugarJ IDE.

3.6 Case studies

We applied the SugarJ IDE implementation to demonstrate the practicability of editor libraries. We have developed editor libraries for a small number of simple language extensions such as regular expressions, where editor services only act locally and no explicit coordination is necessary. These simple editor services compose with the basic SugarJ editor services and other simple editor services through implicit coordination. For example, our regular expression editor library would compose easily with an editor library for SQL to provide editor services for regular expressions nested within SQL statements, because each library acts on syntax tree of the respective DSL only.

In addition to these simple editor libraries, we have conducted three realistic case studies to evaluate the practicability and composability of editor libraries for larger languages: XML and LaTeX, which we describe here, and Java Server Pages, which we describe in Appendix A. In all three case studies, we demonstrate the support of the SugarJ IDE for the staging of editor services, and in the Latex case study we additionally

apply explicit coordination to compose editor libraries.

3.6.1 Growing an XML IDE

XML and XML Schema demonstrate many interesting facets of editor libraries, including domain-specific editor configuration languages and editor-library staging as described in Section 3.3. Although the XML Schema editor library extends the editor library for XML with schema-specific tag completion and validation, both libraries compose with editor services such as Java or SQL. This composability is based on locality and implicit coordination in the form of aggregation and the closest match rule (cf. Section 3.4).

Examples of the use and definition of editor libraries for XML and XML Schema have appeared throughout this chapter. In summary, we have grown our SugarJ IDE through the use of syntactic extensions and editor libraries into an XML-aware IDE that features coloring, folding, outlining, schema-specific tag completion and XML validation. Several potential editor services have not been implemented so far, but qualify as future student projects, for example, reference resolving according to XML Schema references or hover help to display documentation from the schema within the XML document.

3.6.2 Growing a Latex IDE

Language extensions such as XML or regular expressions extend the Java fragment of SugarJ and provide editor services that compose with Java services. Compared to Java, these language extensions are relatively small and do not cross-cut Java programs too much. Therefore, we also wanted to gain experience with incrementally growing a language from scratch by composing multiple sublanguages and their editor services into one unified language. To this end, we grew the SugarJ IDE into a Latex IDE by composing a Latex core with libraries for mathematical formulas, listings of (statically parsed and IDE-supported) source code, and Bibtex bibliographies and citations. However, we only provide an IDE frontend for the Latex language and its libraries: Latex code in the SugarJ IDE compiles to regular Latex files, which use regular Latex libraries. In Figure 3.6, we show a screenshot of our library-based Latex IDE.

The basic Latex language consists of environments **begin**{abstract}...**end**{abstract}, macro calls **emph**{arg}, structure declarations **section**{A}, **paragraph**{B}, and so forth, and of course text. We support these concepts in our core Latex syntax definition and editor library, which, for example, highlights section headers in a bold, blue font, proposes code completions for macro calls, and provides a structural document outline. In separate libraries, we define the syntax and editor support for various extensions of the Latex core.

First, the math library introduces a new language construct for formulas $n \to n + 1$ and according editor services (e.g., highlighting). These services act locally and thus compose with other Latex extensions.

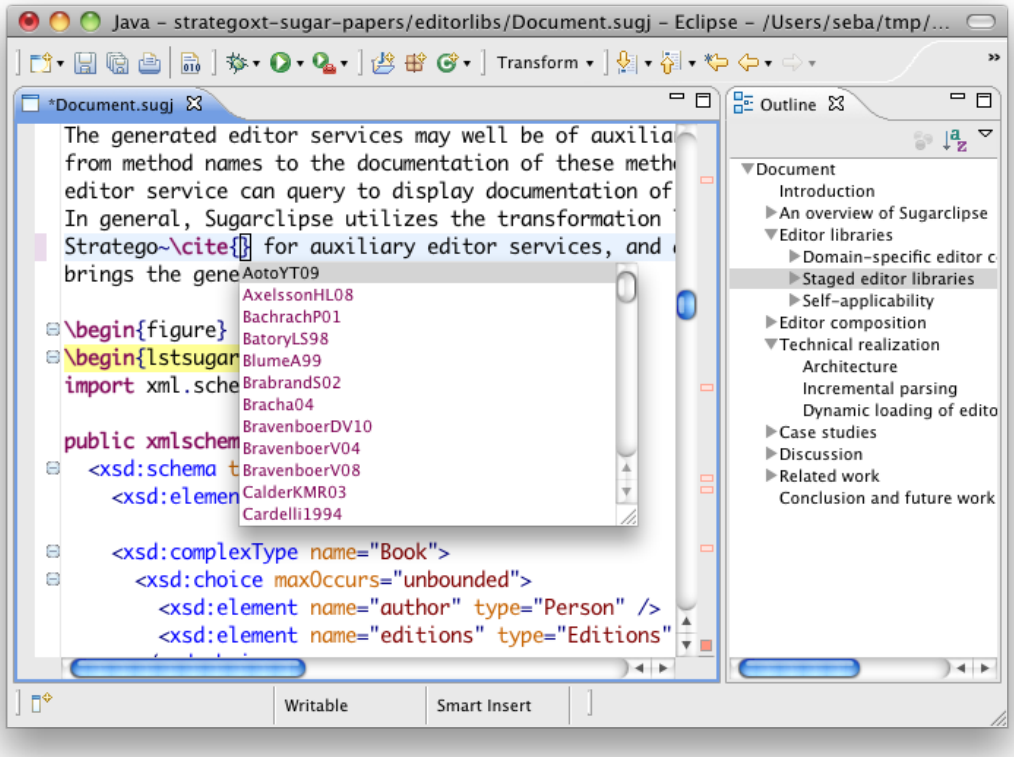


Figure 3.6: Editor services for Latex in the SugarJ IDE: outline, nested syntax coloring, citation completion, reference checking, code folding.

Second, the listings library supports source code listings in a document. Typically, such source code listings are unparsed, unchecked and, often enough, erroneous. In contrast, we provide a library for code listings that statically parses the code to prevent any syntactic errors to slip into a published article. Within our code listing, all language-specific editor services are available if the corresponding editor libraries are in scope. This way, we compose the Latex editor services with editor services for Java, services for editor libraries themselves, and services for language extensions such as XML Schema. For example, while writing a conference article on editor libraries[EKR+11a], the SugarJ IDE provided us syntax coloring, code folding, and error checking for the schema in Figure 3.4, as shown in the screenshot of Figure 3.6.

Third, we separately implemented a syntactic extension and editor library for Bibtex, which, for instance, provides reference resolving and hover help for string constants (such as conference acronyms) within a bibliography. Bibtex and Latex interact via citations

\cite{...} that occur in a Latex document and refer to Bibtex entries. However, the according editor services do not compose automatically in a meaningfully way; explicit coordination is necessary to provide code completion, hover help, or checking for undefined references. We provide these editor services for citations by generating and explicitly coordinating services as described in Section 3.4.3. This way, a Latex document can use any citation key that is provided by an imported Bibtex bibliography. In fact, our encoding allows a Latex document to rely on multiple Bibtex libraries simultaneously.

The key feature of our Latex IDE is its extensibility: users can extend the IDE through syntax definitions and editor libraries to support, for instance, vector-graphics libraries such as TikZ. Staging and explicit coordination of editor services provides the conceptual means for implementing a wide range of powerful IDE extensions.

3.7 Discussion

In this chapter, we have focused on the integration of editor libraries into a syntactically extensible host language such as SugarJ. In this section, we point out a number of further application scenarios for editor libraries and discuss whether it is sensible at all to organize editor services as part of source files.

3.7.1 Language embedding

There are several approaches to embed DSLs, even when the host language is not syntactically extensible. Typical examples are string-based embeddings and embedding a language with standard abstraction mechanisms of the host language, known as pure embedding [Hud98]. The latter works even better if the host language has a flexible syntax, as in Scala. In Figure 3.7, we illustrate three typical embeddings: embedding regular expressions as plain strings in Java, embedding XML as API calls in C#, and embedding LINQ-style queries in Scala.

Even for DSL embeddings in a nonextensible language, we want to add domain-specific IDE support. Even if regular expressions are embedded as strings or XML is embedded as API calls, we want to provide domain-specific editor services such as syntax coloring and content completion. Using editor libraries, DSL implementers can accompany their DSL embeddings with editor services to support programmers.

In the case of string-based embedding, the SugarJ IDE attempts to parse the document in more detail than the host language. In the pure-embedding scenario, we provide editor-service declarations that reason about more complex syntactic structures, for example nesting of XAttribute instantiation inside XElement instantiation. The library mechanism works equally well for languages that are syntactically extensible or not.

```
s.matches("a\\.*[0-9]")
```

(a) String-based embedding of regular expressions in Java.

```
new XElement("book",
  new XAttribute("title", new String(title)),
  new XElement("author", new XAttribute("name", "Mintz")))
```

(b) Pure embedding of XML in C#.

```
from(books)(b => where(b.isPublished) select(b.title))
```

(c) Pure embedding of SQL in Scala.

Figure 3.7: Typical DSL embeddings in Java, C#, and Scala.

3.7.2 Library-based pluggable type systems

The notion of pluggable type systems was first proposed by Bracha and describes type systems that accept extensions (plugins) to enforce additional static analyses on demand [Bra04]. Programmers can configure a pluggable type system by selecting a set of extensions to activate. Due to the support of the SugarJ IDE for marking user-defined errors and warnings visually in the source file and problems view, the SugarJ IDE is especially well-suited for the application of library-based pluggable type systems. In a library-based pluggable type system, type system extensions are organized in libraries and activated through usual import statements.

Pluggable type systems enable the definition of specialized language subsets for various purposes: Pedagogical language subsets prohibit the use of certain language constructs, convention-based language subsets enforce the compliance with code style or author guide lines, language subsets for a particular platform (e.g., Java targeting Google Web Toolkit (GWT)) often support part of the standard library only. However, more sophisticated language restrictions are possible as well. For instance, we implemented XML validation as a library-based type system plugin.

3.7.3 Language integration of editor services

Our SugarJ IDE raises the question whether it is a good idea to have editor definitions as part of the sources of a program. One could argue that such metadata should be kept separate, because it is not part of the program semantics and it potentially couples the sources to a specific IDE. Our answer to this objection is twofold: First, SugarJ and the SugarJ IDE are attempts to tear down stratifications into base and metalevel. This enables self-applicability and the use of the same mechanisms for abstraction, versioning, deployment, evolution and so forth at all metalevels. Second, we tried to reduce the

conceptual coupling to a specific IDE by making the editor definitions as abstract as possible, such that functionality as provided by the SugarJ IDE can be adopted for many IDEs. While more experience is necessary for the final word on editor libraries, we believe that the positive evidence we collected so far makes further research worthwhile.

3.8 Related work

Our work follows in a line of previous work on extensible and customizable code editors, IDEs, and language workbenches. We compare these works to our extensible IDE.

Extensibility of code editors and IDEs. Notable early examples of extensible code editors are Emacs and Vim. They support extensibility by means of plugins, written in dynamic languages such as Lisp and Vim Script. Using APIs and hooks to coordinate actions in the editor, these plugins can introduce syntax highlighting and shortcuts or commands specific to a language. Plugins that introduce more advanced features, such as inline error markers, are rare for these editors.

Modern IDEs distinguish themselves from the traditional code editors and programming environments by combining a rich set of programmer utilities such as version management with a variety of sophisticated language-specific editor services [Fow05a]. These IDEs parse the source code as it is typed, rather than treating it as text with regular-expression-based syntax highlighting. The parsed abstract syntax tree is used for semantic editor services such as inline error marking and content completion. Examples of these IDEs are Eclipse, IntelliJ IDEA, and Visual Studio. Each provides extensibility by means of plugins written in general-purpose languages such as Java or C#, for which APIs and hooks are provided to customize the IDE experience.

Extensible code editors and IDEs use a plugin model for the organization and distribution of editor components. In contrast to our library-based approach, plugins are not part of the object language but are externally implemented and integrate into an editor's architecture directly. This has a number of significant implications. First, editor libraries can be activated through object language imports on a *per-file basis*, whereas plugins require external activation instead, for example, on a per-editor mode or per-language basis. Second, independent editor libraries typically *compose* based on locality and implicit coordination, whereas plugins have to be designed for composition a priori. Third, editor libraries are *declarative* and describe how to perform editor services, rather than imperatively changing the editor execution. Finally, while IDEs such as Eclipse or Visual Studio require the environment to be restarted whenever the implementation of editor service changes, editor libraries ensure a transparent compilation model.

Customizability of code editors and IDEs. IDEs usually provide some adaptability through configurations such as custom coloring schemes or user-defined code templates.

However, these facilities are often coarse-grained and hard to deploy or share. For instance, Eclipse's standard Java plugin JDT defines a fixed set of colorable entities (decimal and hexadecimal numbers must look the same), requires completion templates to apply either to Java statements or type members only, or to complete Java (no completion templates for expressions only) and does not support an import and export mechanism for all editor configurations. In contrast, editor libraries are deployable just like usual Java libraries and enable precise configuration of editor services based on the language's full syntactic structure. Furthermore, since editor libraries are part of the object language, it is possible to package them with conventional programming libraries. This enables library-specific editor services such as code completion templates for typical use cases of an API or warnings for depreciated uses.

Language workbenches. Language workbenches are tools that integrate traditional language engineering tools such as parser generators and transformation systems and tools to develop IDE support [Fow05b]. By combining these tools and by providing IDE support for these metaprogramming tasks, language workbenches enable developers to efficiently create new languages with IDE support.

Language workbenches based on free text editing and parsing include EMFText [HJK+09], MontiCore [KRV08], Rascal [vdS11], Spoofax [KV10], TCS [JBK06] and Xtext [EV06]. These workbenches provide modern editor service facilities such as content completion, following in a line of work on extensible IDEs with metaprogramming facilities, such as the Meta-Environment [Kli93, vdBvDH+01]. Similar to our work, these workbenches provide support for developing and using editor services. However, they strictly separate metaprogramming and programming. Languages and editor services are deployed together in such a way that they apply to a certain file extension. Any changes to the language or editor service can only be applied at language-definition level. In contrast, in our work editor services can be freely imported and composed as editor libraries across any number of metalevels, which enables the self-application of editor services.

In addition to language workbenches designed to implement arbitrary textual languages, there are also tools that are based on a fixed host language. Examples include Helvetia [RGN10], a Smalltalk-based environment, and DrRacket [FCF+02], aimed at the Racket programming language (formerly known as Scheme). Helvetia supports syntactic extensibility and custom syntax highlighting for extensions through a dynamic meta-object protocol, but has no support for more sophisticated editor services such as reference resolving or content completion. DrRacket does not provide the same syntactic flexibility as Helvetia or our IDE, but does provide autogenerated reference resolving editor services. In Helvetia, language definitions can be loaded for a Smalltalk image and activated in parts of the application. In DrRacket a language definition can only be selected at file level using the `#lang` directive. Both tools are highly tied to their respective host languages, using dedicated metaprogramming systems. For instance,

reference resolving in DrRacket demands that new constructs for binding identifiers are defined in terms of predefined binding constructs of the Racket language. In contrast, our editor libraries approach is language-agnostic as our Java-independent case study for Latex shows.

MPS is a language workbench based on projectional editing rather than free text editing [Völ10, VS10], notable for its support for language composability. It allows language extensions to be activated in specific parts of an application, but does not organize them as true libraries. MPS strictly separates metaprogramming and programming by providing fixed templates for syntactic and semantic customization of language components.

3.9 Chapter summary

Our main idea for the SugarJ IDE is the application of libraries for organizing IDE extensions as reusable units. This combines the flexibility of extensible tooling with the principles of libraries, in particular modular reasoning, code reuse, and composability. As our case studies show, editor libraries are particularly beneficial in combination with syntactically extensible programming languages such as SugarJ and represent an important step towards our ultimate goal of *language libraries*. Language libraries enable the implementation of all aspects of a language as a library. Currently, we support the library-based adaptation of parsing, desugaring, analyzing and editor presentation, but lack library-based extensibility for implementing the semantics of a language extension. In our future work, we would like to support the configuration of builder services that declare the semantics of embedded languages and integrate into the SugarJ IDE naturally. Builder services should replace traditional build scripts completely and specify the order as well as the tool used to build a set of source files.

In addition, we would like to further investigate the modularity and composability of editor libraries. In particular, we would like to explore scoping mechanisms for editor libraries that retain composability while providing clearer interfaces for explicitly coordinating services with staged editor libraries. We also plan to conduct a large-scale case study to evaluate the composability of editor libraries more accurately, namely Java Server Pages. Java Server Pages brings together a number of languages such as HTML, Java, JavaScript and CSS. We plan to provide editor libraries for each of these language separately and to compose the resulting editor libraries to form an editor library for Server Pages. While conducting this case study, we would furthermore like to explore new declarative means for explicitly coordinating editor libraries.

4 Declarative Syntax Descriptions for Layout-sensitive Languages

This chapter shares material with the SLE'12 paper "Layout-sensitive Generalized Parsing" [ERKO12].

One of the goals of SugarJ is to provide programmers with the flexibility that typically is reserved for developers of a programming language, namely to define extensions. We promote language extensions as first-class language constructs that programmers can directly rely on to define domain abstractions specific to their needs. However, when programmers become language developers as in SugarJ, one important aspect is to provide declarative language-definition mechanisms that are easy to use.

In particular, SugarJ language definitions consist of a parser, a transformation, and editor services. As described in the previous chapters, SugarJ employs the SDF, Stratego, and Spoofax's editor-service configuration language for language definitions. The reuse of these declarative languages was essential in the development of SugarJ, because it allowed us to focus on the novel concept of library-based extensibility (see Chapter 2 and Chapter 3). However, due to this reuse, SugarJ also inherits the respective limitations of SDF, Stratego, and Spoofax. One particular profound limitation for SugarJ is SDF's confinement to context-free languages, which restricts the possible extensions and host languages that SugarJ can support.

The theory of context-free languages is well-understood and context-free parsers like SDF 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 or language extension *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.

The work described in this chapter is an essential stepping stone for making SugarJ-like flexibility available for layout-sensitive languages. In particular, in the subsequent Chapter 5, we present the extensible programming language SugarHaskell that brings flexible and principled domain abstraction to the layout-sensitive language Haskell.

4.1 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 [MB90], ANTLR [PQ95, PF11], PEGs [For02, For04], parsec [LM01], or SDF [Vis97b], 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 [PQ95]), one particularly relevant class of languages is not supported declaratively by any existing parser framework: layout-sensitive languages.

Layout-sensitive languages were proposed by Landin in 1966 [Lan66]. 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 words, a token is offside if it occurs further to the left than the starting token of a structure; an offside token must denote 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 4.1 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 extend these languages with new syntax or to create tools for them, such as refactoring engines 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

[1] `http://en.wikipedia.org/w/index.php?title=Off-side_rule&oldid=517733101`

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

Figure 4.1: Layout-sensitive languages use indentation instead of curly braces.

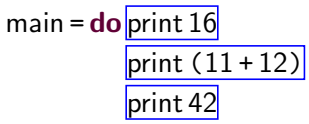
grammar and restrict the applicability of annotated productions to text with valid layout. For example, for conditional expressions in Python, we annotate (among other things) that the `if` keyword must start on the same column as the `else` keyword and that all statements of a `then` or `else` branch 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 [Vis97b] that supports layout constraints. The standard parsing algorithm for SDF is scannerless generalized LR parsing [Vis97a]. 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 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.

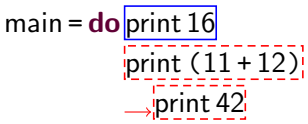
To validate the correctness and to evaluate the performance of our layout-sensitive parser, we have build layout-sensitive SDF grammars for Python and Haskell. In particular, we applied our Haskell parser to all 33 290 Haskell files in the open-source repository Hackage. We compare the result of applying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

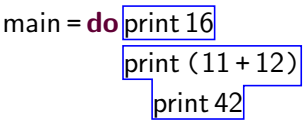
- We identify common idioms in existing layout-sensitive languages. Based on these



(a) Three statements with correct vertical alignment.



(b) Wrong parse: Statements have to begin at 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.

Figure 4.2: Simple Haskell programs.

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 implemented a layout-sensitive SDF grammar for Python and extended an existing layout-insensitive SDF grammar for Haskell[2] 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 application.

4.2 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 4.2(a) contains three statements which are horizontally 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 4.2(b) displays an incorrect parse tree that wrongly identifies print 42 as a separate statement, even though it is further indented than the other statements. Figure 4.2(c) visualizes the correct parse tree for this example: A do block

[2]Based on a grammar from the Haskell transformation framework HSX (`http://strategoxt.org/Stratego/HSX`).

```
catch ( do print 16
          print (11 +
            12) )
        (\e -> do putStr "error:  "
                  print e )
```

(a) Exception handler.

```
catch ( do print 16
           print (11 +
             12) ) (\e -> do
           putStr "error:  "
           print e )
```

(b) Means the same as (a).

Figure 4.3: More complicated Haskell programs.

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 4.2(a) from programs as in Figure 4.2(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[3] 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 [KVW10].

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 [Mar10]. For example, the Haskell program in Figure 4.3(a) is valid: The statement print (11 + 12) only includes one closing parenthesis, because the second closing parenthesis cannot be consumed inside the statement. An algorithm for layout handling could 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

[3] `http://docs.python.org/3/reference/`

```
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}
```

Figure 4.4: Excerpt of our layout-sensitive Haskell grammar. Statements with implicit layout (Impl) have to follow the offside rule. Statements have to align horizontally. Statements with explicit layout (Expl) are not layout-sensitive.

consumed. As a second and more extreme example, consider the program in Figure 4.3(b) that has the same parse tree as the program in Figure 4.3(a). In particular, the statements belong to different do blocks even though they line up horizontally. 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 low-level, algorithmic specifications of layout rules prevalent in existing language specifications and implementations.

In this section, 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.

4.3 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 4.2 and 4.3. 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 4.4 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-

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

Figure 4.5: Syntax of layout constraints c that can annotate SDF productions.

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 4.5. Layout constraints can refer to direct subtrees (including terminals) of the annotated production through numerical indexes.

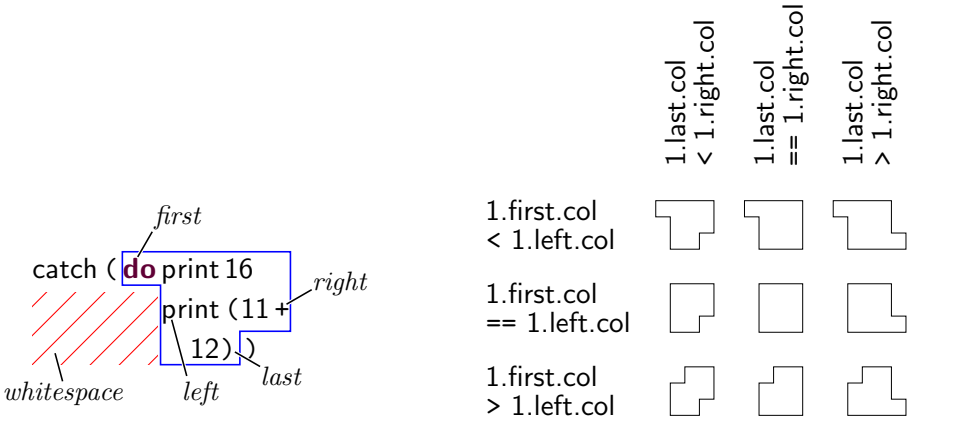
Each subtree exposes its shape via the source location of four tokens in the subtree, which describe the relevant positions in the token stream. Layout constraints use *token selectors* to access these tokens: first selects the first non-whitespace token, last selects the last non-whitespace token, left 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 4.6(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, the constraint 1.first selects the first token of the function application, yielding the character p for scannerless parsers, or the token print otherwise. 1.left selects the left-most token not on the first line, that is, the operator symbol ∗. 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:



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

(b) Layout constraints that mention only one tree restrict the shape of the surrounding box.

Figure 4.6: Example layout constraints and the corresponding boxes.

```
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 4.6(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 4.4 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 horizontal 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 4.4 declares explicit-layout statement lists. Since these lists use explicit layout {stmt;...;stmt}, no additional constraints are needed. 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 skip 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.

4.4 Layout-sensitive parsing with SGLR

We implemented a layout-sensitive parser based on our extension of SDF [Vis97b] with layout constraints. Our parser implementation builds on an existing Java implementation [KdJNNV09] of scannerless generalized LR (SGLR) parsing [Tom87, Vis97a]. 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.[4] 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.

4.4.1 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

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

additional information (e.g., precedence information) specified together with the context-free grammar is used to discard as many of 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 the leaves of 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. [BVVV05] 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 needed.

4.4.2 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, a number on the last line can be argument 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 part 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 [Tom87]. 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 [Tom87] 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 4.4 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, and both parse trees can be generated by the Impls nonterminal (Figure 4.4). 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: Each tree represented by such state would be rejected by the constraint. In particular, such constraints only use 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 horizontally 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 4.2(b) at parse time because the statements are not aligned. However, it does not allow us to reject or distinguish the programs shown in Figure 4.2(a) and 4.2(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 [Vis97a], for each list of subtrees, we validate that the applied production permits the layout of the subtrees. We perform the regular reduce action if the production does not specify a layout constraint, or the constraint is satisfied, or the constraint cannot be checked at parse time. 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.

4.5 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?

4.5.1 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.[5] We have extended an existing SDF grammar for Haskell that required explicit layout[6] 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.[7] 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

[5] Actually, we improved the original implementation by eliminating recursion to avoid stack overflows when parsing files with long comments or long literal strings.

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

[7] `http://hackage.haskell.org/package/haskell-src-exts`

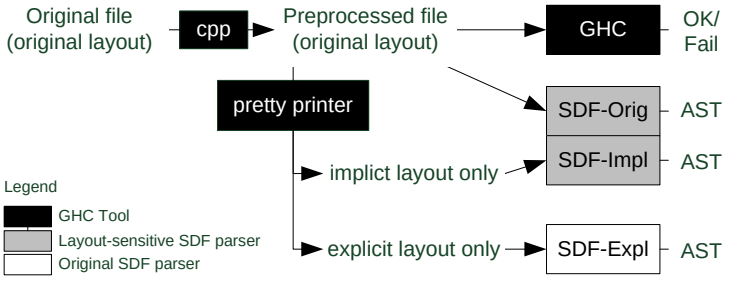


Figure 4.7: Evaluation setup

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 4.7. 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 [Mar10], but additionally supports the following extensions to 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.[8] We extracted the latest version of all 3081 packages that contain Haskell source code on May 15, 2012. In total, these packages

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

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 4.7. 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.

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.

4.5.2 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 4.8. Some differences are due to timeouts; the diagram in Figure 4.9 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 4.10(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

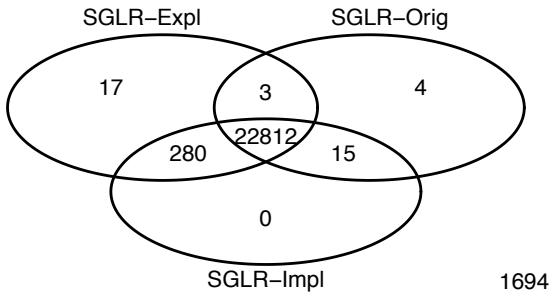


Figure 4.8: Number of files each parser produces the correct AST for.

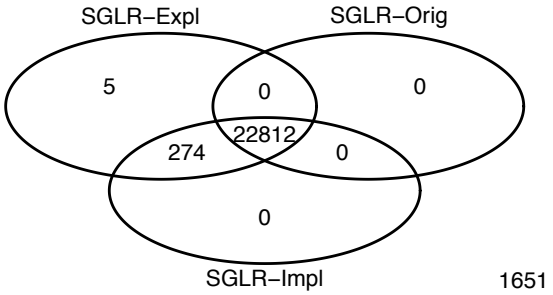


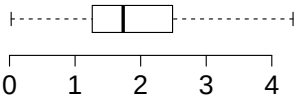
Figure 4.9: Correct parses ignoring files that timeout with at least one parser.

`SGLR-Impl` is 1.8 times slower than `SGLR-Expl`. We show the distribution of performance penalties as box plot in Figure 4.10(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 4.11, 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.

4.5.3 Interpretation and discussion

As shown in Figure 4.8, `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 4.9. 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:



(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.

Figure 4.10: Performance of layout-sensitive parsing.

```
{-# 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 complies with the offside rule. 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 slow down of 1.8 compared to layout-insensitive parsing with `SGLR-Expl`. Given the median parse times per file (Figure 4.10(b)), this slow down is still in the realm of a few milliseconds and suggests that layout-sensitive parsing can be applied in practice. In particular, this slow down seems acceptable given the benefits of declarative specifications

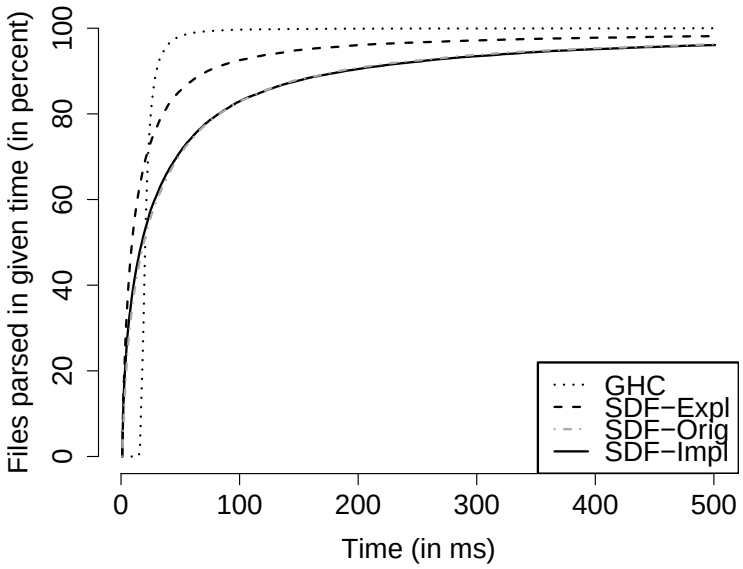


Figure 4.11: Distribution of parsing times.

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 4.6.

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 slow down does not inhibit practical application of our parser.

4.5.4 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).

4.6 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 [vdBSVV02]. 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 4.4 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 4.4.2). 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, for example, from XPath. 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 [McB04]. We would like to investigate whether our approach to layout-sensitivity generalizes to two-dimensional parsers.

4.7 Related work

We have significantly extended SDF's frontend [Vis97b] and its SGLR backend [Tom87, Vis97a] 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.[9] 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 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[10] 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 [DFS09].

Data-dependent grammars [JMW10] support the declaration of constraints to restrict the applicability of a production. However, constraints in data-dependent grammars must be context-insensitive [JMW10, Lemma 4], and therefore cannot be used to describe languages with context-sensitive layout such as Haskell.

4.8 Chapter summary

We have presented a declarative mechanism for specifying layout-sensitive languages based on layout constraints in context-free grammars. We have developed a parser for these grammars based on SGLR. Our parser enforces constraints at parse time when possible but fully validates parse trees at disambiguation time. We have empirically

[9] `http://svn.python.org/projects/python/trunk/Modules/parsermodule.c`
[10] `http://www.haskell.org/alex/`

shown that our parser is correct and the performance penalty is acceptable compared to layout-insensitive generalized parsing. While our parser implementation is based on a scannerless parser, the ideas presented in this chapter are applicable to parsers with separate lexers as well. 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, where regular programmers write syntactic language extensions. This requires a declarative and composable syntax formalism as provided by SDF, but supplemented with support for layout-sensitive language. Based on the work presented in this chapter, we have been able to implement SugarHaskell, a syntactically extensible programming language based on Haskell, which we present in the following chapter.

5 A Framework for Library-based Language Extensibility

This chapter shares material with the HASKELL'12 paper "Layout-sensitive Language Extensibility with SugarHaskell" [ERRO12].

The core idea explored in this thesis is to use library-based language extensibility for flexible and principled domain abstraction. In Chapter 2 and Chapter 3, we investigated library-based language extensibility in SugarJ, an extensible programming language that supports domain-specific syntax, domain-specific static analyses, and domain-specific editor services. In this chapter, we generalize SugarJ to a framework for library-based language extensibility.

SugarJ is based on Java, in which application code is written by the user or generated by desugarings. However, the ideas behind SugarJ do not depend on Java. Instead, we hypothesize that library-based language extensibility can be made available for any programming language that has a notion of *libraries*.

To validate this claim, we have developed a framework for library-based language extensibility that can be instantiated for different base languages. The framework is based on the SugarJ compiler, but abstracts over the Java-specific fragments of the compiler using an abstract class [Rie12]. The resulting compiler framework can be instantiated for different base languages. So far, we have instantiated the framework to build support for library-based language extensibility based on Java, Prolog, F_ω, and Haskell.

In this chapter, we present the extensible programming language SugarHaskell that uses Haskell as a base language. SugarHaskell satisfies the same design goals as SugarJ: domain-specific syntax, domain-specific static analysis, domain-specific editor services, modular reasoning, implementation reuse, declarativity, composability, and uniform self-application. However, in contrast to Java and as discussed in the previous chapter, Haskell is a layout-sensitive programming language. SugarHaskell embraces the layout-sensitivity of Haskell and also supports layout-sensitive language extensions using layout constraints introduced in the previous chapter. Building on our previous work on syntactic extensibility for Java, SugarHaskell integrates syntactic extensions as sugar libraries into Haskell's module system. Syntax extensions in SugarHaskell can declare arbitrary context-free and layout-sensitive syntax. SugarHaskell modules are compiled into Haskell modules and further processed by a Haskell compiler. We provide an Eclipse-based IDE for SugarHaskell that is extensible through editor libraries, and automatically provides syntax coloring for all syntax extensions imported into a module.

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

5.1 Introduction

Many papers on Haskell programming propose some form of domain-specific syntax for Haskell. For instance, consider the following code excerpt from a paper about applicative functors [MP08]:

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

The *idiom brackets* [| ... |] used in this listing are not supported by the actual Haskell compiler; rather, the paper explains that they are a shorthand notation for writing this more elaborate code:

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

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

Another practical problem of syntactic language extension is that integrated development environments (IDEs) should know how to deal with the new syntax and provide domain-specific editor services, for example, for syntax coloring, auto completion, or reference resolving. IDEs can be extended, of course, but this again is not declarative, not modular, and does not support composition.

We propose a generic extension to Haskell, SugarHaskell, with which arbitrary syntax extensions can be defined, used, and composed as needed. In SugarHaskell, a syntactic extension is activated by importing a library which exports the syntax extension and defines a desugaring of the extension to SugarHaskell. Using SugarHaskell, we can realize

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

the code example from above as follows:[2]

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

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

The syntactic extension and its desugaring is defined in the library IdiomBrackets. By importing this library, the notation and its desugaring are activated within the remainder of the current module. When the SugarHaskell compiler is invoked, it desugars the brackets to the code using pure and <*> from above. Modules that do not import IdiomBrackets are not affected by the syntactic extension. If more than one syntax extension is required in the same file, the extensions are composed by importing all of them. Conflicts can arise if the extensions overlap syntactically, but this is rare for real-world examples and can usually be disambiguated easily.

SugarHaskell also comes with an Eclipse-based development environment specifically tailored to support syntactic extensions. By importing the IdiomBrackets library, syntax coloring for the extended syntax is automatically provided. More advanced IDE services can be defined in and imported from *editor libraries* (see Chapter 3).

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

SugarHaskell builds on our earlier work on SugarJ, a syntactically extensible version of Java presented in Chapter 2. The research contributions of this chapter are as follows:

- SugarHaskell demonstrates that flexible and principled domain abstraction is not confined to Java-based languages, but similar extensibility is feasible for other base languages, too.
- To create SugarHaskell, we developed a framework for library-based language extensibility that decouples the syntax-extension mechanism of SugarJ from the underlying programming language. To this end, we generalized the SugarJ compiler by creating an interface that abstracts over the base language. We describe the design of this interface and how we used it to implement SugarHaskell.

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

- Haskell presents a new technical challenge not present in Java: layout-sensitive parsing [Mar10, Sec. 2.7]. SugarHaskell allows the definition of layout-sensitive syntactic extensions and is, to the best of our knowledge, the first declaratively extensible parser for Haskell with layout-sensitive syntax. We validate the extensibility of our parser by developing a layout-sensitive language extension of Haskell, namely arrow notation [Pat01].

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

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

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

5.2 SugarHaskell by example

To illustrate SugarHaskell, let us integrate syntactic sugar for programming with arrows [Hug00]. Arrows are a versatile generalization of monads and, like monads, arrows are somewhat cumbersome to use without syntactic support. For this reason, Paterson proposed *arrow notation* to make programming with arrows more convenient [Pat01]. In this section, we implement arrow notation with SugarHaskell.

We are not the first to support arrow notation for Haskell. Paterson developed a preprocessor[3] that translates Haskell code with arrow notation into Haskell 98 code. Furthermore, GHC supports arrow notation through a compiler extension, which can be activated by the `-XArrows` flag [GHC12, Section 7.13]. In contrast, SugarHaskell empowers regular programmers to integrate custom syntactic extensions that compose.

5.2.1 Arrow notation

Figure 5.1 summarizes the syntactic extension for arrow notation as specified by GHC [GHC12, Section 7.13]. Arrow notation is centered around commands *cmd*, which are like expressions but provide different syntax for applications. The first and second command

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

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

```
calt ::= pat -> cmd [where decls]
       | pat (guards -> cmd)+ [where decls]

cstmt ::= let decls
        | pat <- cmd
        | rec { cstmt ; ... ; cstmt }
        | cmd


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

Figure 5.1: Syntactic additions for arrow notation.

productions specify arrow application where the right-hand-side expression is input to the arrow described by the left-hand-side expression. Here, GHC (and we) distinguish forwarding arrow application (exp -< exp) from the arrow application (exp -<< exp) that uses app from the ArrowApply type class. The third and fourth productions declare application of an expression to commands and vice versa. The brackets (|...|) have been introduced into GHC to syntactically distinguish these two forms of application. The remaining command productions parallel standard expression syntax for commands. Finally, arrow notation integrates into regular Haskell syntax by extending the expression nonterminal *exp* from the Haskell grammar. Arrow notation introduces new expression syntax proc pat -> cmd where proc is a new keyword for building arrows whose input matches pat and whose output is determined by the command cmd.

An example SugarHaskell program that uses arrow notation is shown in Figure 5.2. It activates arrow notation by importing the arrow sugar library Control.Arrow.Syntax alongside the standard arrow library. Arrow notation is only active where the import is in scope, that is, in the current module. Therefore, it is possible to use competing syntactic extensions in different modules, but also to compose different syntax extensions in a single module by importing all of them. For example, idiom brackets (Section 5.1) do not conflict with arrow notation since brackets in arrow notation can only occur inside a command. Therefore, these two sugar libraries can be used within the same module. Let us now look at the implementation of the arrow sugar library.

A sugar library consists of two artifacts: A grammar that specifies an extended syntax and a transformation that translates the extended syntax into Haskell code (or Haskell code extended by other sugar libraries). To specify the syntax, we employ the generalized LR parsing formalism SDF [Vis97b], which we extended to support layout-sensitive languages. SDF has two major advantages over other parsing technologies. First, since SDF uses a generalized LR parser, it supports declarative grammar specifications that

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

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

Figure 5.2: Hughes's λ-calculus interpreter [Hug00] using arrow notation in SugarHaskell.

liberates developers from such concerns as left-recursion or encoding priorities. Second, SDF organizes grammars in composable modules and features a number of disambiguation mechanisms that make it possible to add syntax without changing previous syntax definitions. This enables SugarHaskell users to modularly add syntactic extensions to Haskell without changing the original Haskell grammar.

We have decomposed the syntax definition for arrow notation into three sugar libraries: one for command alternatives, one for command statements, and one for commands themselves. The latter one is shown in Figure 5.3. A SugarHaskell sugar library integrates into Haskell's module system. Accordingly, each sugar library starts with a module declaration and a list of import statements. These imports typically refer to other sugar libraries whose syntax is extended. The body of a sugar library is composed of SDF syntax declarations and desugaring transformations (more on desugarings later). Essentially, the syntax declaration in Figure 5.3 reflects the EBNF grammar from Figure 5.1. In SDF, the defined nonterminal appears on the right-hand side of the arrow `->`. Hence, the first production declares a new syntactic form for Haskell expressions. After a

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

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

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

  HaskellExp "-<" HaskellExp           -> ArrCommand {cons("ArrFirst")}
  HaskellExp "-<<" HaskellExp          -> ArrCommand {cons("ArrHigher")}
  "(|" HaskellExp ArrCommand+ "|)"     -> ArrCommand {cons("ArrForm")}
  ArrCommand HaskellExp                -> ArrCommand {cons("ArrAppBin"), left}
  ArrCommand HaskellQop ArrCommand     -> ArrCommand {cons("ArrOpApp"), right}
  "\\" HaskellFargs "->" ArrCommand    -> ArrCommand {cons("ArrAbs")}
  "do" ArrStmtList                  -> ArrCommand {cons("ArrDo"), longest-match}
  ...
```

Figure 5.3: SugarHaskell syntax extension for arrow notation.

production, a list of annotations can follow in curly braces. The cons annotation specifies the name of the AST node corresponding to a production. The annotations left and right declare a production to be left-associative or right-associative, respectively. Finally, longest-match denotes that in case multiple parses are possible (SDF uses a generalized parser), the longest one should be chosen. These productions are supplemented with priority declarations (left out for brevity), which, for example, specify that the ArrAppBin production has precedence over the ArrOpApp production.

By importing the Control.Arrow.Syntax.Command module, a program using the extended syntax can already be parsed by SugarHaskell. However, compilation will fail because the parsed AST contains arrow-specific nodes like ArrProcedure that will not be understood by the compiler. Therefore, we require a desugaring transformation that relates the arrow-specific nodes to Haskell nodes (or nodes from another syntactic extension). To implement desugaring transformations, SugarHaskell employs the Stratego term-rewriting system [VBT98]. Stratego rules are based on pattern matching but, in contrast to many other systems, Stratego rules are open for extension: A rule can be amended in a separate module to handle more syntactic forms [HKGV10]. This way, all SugarHaskell extensions in scope contribute to a single desugaring transformation that desugars an AST bottom-up.

Figure 5.4 displays an excerpt of the desugaring transformation for arrow notation. First, let us inspect the import statements. The first import just brings the concrete and abstract command syntax into scope, which is the input language of the transformation

we are about to define. However, the second import is special: It activates a SugarHaskell extension that does not affect the object language Haskell but the metalanguage Stratego. The sugar library Meta.Concrete.Haskell activates concrete syntax for transformations [Vis02], that is, it enables metaprogrammers to describe AST transformations by concrete syntax within |[...]| instead of abstract syntax. Since SugarHaskell extensions are self-applicable, syntactic extensions to the metalanguage can be expressed as a sugar library as well. Moreover, in our example, the metaextension is further extended by Control.Arrow.Syntax.Concrete, which enables concrete syntax for arrow commands after the cmd keyword.

Using concrete Haskell syntax in Stratego transformations, the desugaring transformation follows the GHC translation rules for arrow notation [PP04] except for some optimizations. The entry point of our desugaring is the desugar-arrow rule as declared by the **desugarings** block. Each Stratego rule declares a pattern on the left-hand side of the arrow -> and produces the term on the right-hand side of the arrow. In concrete syntax, we use $ to escape to the metalanguage in correspondence with TemplateHaskell [SP02]. Accordingly, in the first transformation rule desugar-arrow in Figure 5.4, the pattern matches on an arrow procedure and binds the Stratego variables pat and cmd. If the matching succeeds, the rule produces a term that constructs an arrow with arr from a lambda expression and composes (>>>) this arrow with the result of desugaring cmd. Note that angled brackets <r> t in Stratego denote an application of the rewrite rule r to the term t.

The module Control.Arrow.Syntax imports and reexports the two modules that define the syntax and desugaring for arrow notation. Since sugar libraries are integrated into Haskell's module system, an import statement suffices to activate the syntactic extension as illustrated in Figure 5.2. Moreover, SugarHaskell modules that contain (possibly sugared) Haskell code compile into a pure Haskell module. Therefore, SugarHaskell programs are interoperable with regular Haskell programs: The application of SugarHaskell in a library is transparent to clients of that library.

5.2.2 Layout-sensitive syntactic extensions

In order for a syntactic extension to integrate into Haskell seamlessly, the syntactic extension needs to adhere to the layout-sensitive rules of Haskell. For example, arrow notation includes arrow-specific do blocks that consists of a sequence of command statements, as visible in the interpreter in Figure 5.2 and the last production in Figure 5.3. All existing layout-sensitive languages we know of employ hand-tuned lexers or parsers. However, since we want regular programmers to write SugarHaskell extension, we need a declarative formalism to specify layout-sensitive syntax.

To this end, as presented in the previous chapter, we have developed a variant of SDF that supports layout-sensitive languages. In our variant, SugarHaskell programmers can annotate productions with *layout constraints* that restrict the context in which this

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

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

desugarings
  desugar-arrow

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

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

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

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

  ...
```

Figure 5.4: Desugaring transformation for arrow notation using concrete syntax.

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

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

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

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

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

Figure 5.5: Layout constraints restrict the context in which a production may be used.

production may be used. Figure 5.5 shows the use of layout constraints in the definition of arrow-specific statement lists. A statement list can employ implicit or explicit layout. In the latter case, the statement list is encapsulated in curly braces and statements are separated by semicolons. Hence, an explicit statement list does not pose any layout constraints. What is more, an explicit statement list may even violate constraints imposed by the surrounding context. For example, the following is a syntactically valid Haskell program where the do block consists of three statements:

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

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

Statement lists with implicit layout are harder to realize. Essentially, they need to adhere to two invariants. First, each statement must adhere to the offside rule [Lan66], that is, every token is further indented than the token that starts the statement. This invariant is expressed by the first constraint in Figure 5.5: 1.first.col selects the column

of the starting token of the first subtree of the current production; in contrast, 1.left.col selects the column of the leftmost non-starting token of the first subtree of the current production. Our parser prevents the application of the annotated production for code that does not satisfy the annotated constraint. The second invariant declares that each statement in a statement list must start on the same column. This invariant is expressed by the second constraint on the last line of Figure 5.5.

Due to the self-applicability of SugarHaskell, our layout-sensitive parser is not limited to the object language. We employ the same layout-sensitive parser for parsing object-level programs and metaprograms. Thus, metaprograms can make use of layout-sensitive syntax, too. In particular, when using concrete Haskell syntax to declare transformations, the quoted Haskell syntax is layout-sensitive. For example, the last rule of Figure 5.4 matches on an arrow-specific do block. The Haskell snippet used to match on such expressions is parsed layout-sensitively, that is, indenting or dedenting the remaining statement list $*cs will lead to a parse error. While this may seem overkill for such small example, it becomes essential when generating code that nests let, do, case, and where blocks.

5.3 Technical realization

We realized SugarHaskell on top of our previous work on SugarJ. Like SugarJ, SugarHaskell is a syntactically extensible programming language that integrates syntactic extensions into the module system of the base language. However, to realize SugarHaskell, we significantly reengineered the SugarJ compiler to factor out base-language-specific components and to hide them behind an abstract data type. In the resulting framework for library-based language extensibility it is relatively easy to realize syntactic extensible for additional base languages.

5.3.1 Base-language-specific processing of the SugarJ compiler

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

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

sugar library, the compiler activates it right away by adapting the parser and desugaring transformation.

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

5.3.2 The Haskell language library

We reengineered the SugarJ compiler to support base languages other than Java [Rie12]. To this end, we designed an abstract data type LanguageLib that encapsulates base-language-specific components of the compiler. To date, we have implemented four instances of LanguageLib: JavaLib, HaskellLib, PrologLib, and FomegaLib for a syntactically extensible variant of F_ω.

The important categories of abstract methods in LanguageLib are:

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

Notably, the SugarJ compiler handles declarations of sugar libraries and editor services independent of concrete language libraries. Moreover, a language library can perform

static checking and notify the programmer at compile time. For example, HaskellLib ensures that imports of Haskell modules are resolvable by calling ghc-pkg.

5.4 Case study

We evaluated our framework for library-based language extensibility by instantiating it for SugarJ, SugarProlog, SugarFomega, and SugarHaskell. In this section, we demonstrate that the framework in fact provides the same flexibility and principles as the original SugarJ compiler. To this end, based on SugarHaskell, we implemented a sugar library that extends Haskell with a DSL for syntax declarations, namely EBNF. A Haskell programmer can use this extension to specify an EBNF grammar, which we desugar into an algebraic data type (the abstract syntax) and Haskell functions to parse a concrete-syntax string into instances of that data type. Moreover, from a concrete EBNF grammar we generate yet another syntactic extension that allows programmers to use their own concrete syntax in Haskell code to pattern-match or construct values of their abstract syntax (the generated data type). In addition, we defined an analysis that checks for left-recursion in a grammar, and our IDE provides simple editor services for EBNF.

This case study demonstrates that our framework provides flexible and principled domain abstraction with domain-specific syntax, domain-specific static analysis, domain-specific editor services, modular reasoning via imports, implementation reuse, declarative definitions, extension composition, and uniform self-application to generate other extensions.

5.4.1 EBNF: A DSL for syntax declarations

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

To make writing examples easier, one could implement a parser. However, writing parsers is tedious, distracting, and produces additional maintenance overhead when the abstract syntax changes. For that reason, we propose a syntactic integration of EBNF with which programmers can simultaneously declare the abstract and concrete syntax of the language under design. For example, Figure 5.6 shows a SugarHaskell program that specifies the concrete and abstract syntax of the lambda calculus using our EBNF embedding.

EBNF grammars are organized by nonterminal. For the lambda calculus, we use three nonterminals Var, Exp, and String, where String is primitive and describes sequences of

```
module Lambda.Syntax where

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

Var ::= String {Var}

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

Figure 5.6: Declaration of concrete and abstract syntax of the lambda calculus using the EBNF sugar library.

non-whitespace characters. The concrete syntax of all other nonterminals is user-supplied. In addition to concrete syntax, a programmer specifies abstract syntax by supplying the names of AST nodes in curly braces. If no node name is supplied, the corresponding production only forwards its children to the surrounding production but does not produce an AST node itself. For example, according to the lambda-calculus grammar, the string `"lambda f. lambda x. (f x)"` is concrete syntax for:

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

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

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

Accordingly, the grammar from Figure 5.6 desugars into the following data-type declarations:

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

To encode the concrete syntax of an EBNF grammar, we generate the definition of a Haskell function that parses a string into instances of the previous data types. The generated functions employ Parsec [LM01] to parse the input and are used to derive an instance of the Read type class. Hence, the following declarations are generated for the lambda-calculus grammar:

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

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

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

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

We have designed the EBNF sugar library such that clients can configure which artifacts to generate from the grammar. To this end, the main desugaring of EBNF calls a fixed set of pattern-matching Stratego rules, each of which supports no input at all and always fails. Stratego's extensibility mechanism allows programmers to amend those rules in other modules to handle further input (a rule is only executed once even if definitions overlap) [HKGV10]. Thus, by bringing further sugar libraries into scope, a programmer can effectively configure the desugaring of an EBNF grammar. This design is visible in Figure 5.6, where we activate the desugaring into data-type and parser declarations through the imports of Data and Parser, respectively. If we do not want a parser, we can drop the corresponding import to deactivate its generation. On the other hand, it is not possible to only deactivate the data-type generation because the generated parser depends on it. Hence, Parser reexports Data and an import of Parser activates Data as well. In addition to Data and Parser, a client of the EBNF sugar library can import Data.EBNF.MetaSyntax to activate a desugaring that employs SugarHaskell's

self-applicability as we explain in the following subsection.

5.4.2 EBNF: A meta-DSL

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

To address these concerns, we provide another desugaring of EBNF grammars defined in Data.EBNF.MetaSyntax. This desugaring generates a syntactic extension of Haskell specific to a concrete EBNF grammar. To illustrate the generated sugar, Figure 5.7 displays a definition of the small-step operational semantics of the lambda calculus.

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

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

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

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

```
module Lambda.Eval where

import Lambda.Syntax

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

subst v e |[ Exp | Var$w ]|
  | v == w    = e
  | otherwise = |[ Exp | Var$w ]|
subst v e |[ ($e1 $e2) ]|
  = |[ ($(subst v e e1) $(subst v e e2)) ]|
subst v e |[ lambda $w. $b ]|
  | v == w    = |[ lambda $w.  $b ]|
  | otherwise = |[ lambda $w'. $b' ]|
      where w' = nextFreeVar w (freeVars e ++ freeVars b)
            b' = subst v e (subst w |[ Exp | Var$w' ]| b)

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

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

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

Figure 5.7: Small-step operational semantics of the lambda calculus using MetaSyntax.

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

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

In addition, MetaSyntax provides a generic desugaring that translates concrete-syntax expressions into Haskell expressions. For example, this desugaring translates the AST of identEta in Figure 5.7

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

into the corresponding Haskell expression:

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

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

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

5.5 Discussion and future work

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

5.5.1 Haskell integration

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

any generated grammars and desugaring rules. Together, this supports the following interoperation scenarios:

- A Haskell program is compiled by SugarHaskell. This is supported because pure Haskell programs are forwarded unchanged to GHC.
- A Haskell library is used in a SugarHaskell program. This is supported because SugarHaskell uses the GHC package database to locate the Haskell library.
- A SugarHaskell library is used in a Haskell program. This is supported because extensions are just syntactic sugar: SugarHaskell programs always desugar into pure Haskell programs and no special run-time support is required. Hence, a library author can use SugarHaskell to develop a library and deploy the library as desugared Haskell code. Thus, the use of SugarHaskell is transparent to users of the library.

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

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

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

5.5.2 Extension composition

SugarHaskell achieves composability by employing composable metalanguages, namely SDF and Stratego. More specifically, SugarHaskell supports the composition of sugar libraries that are syntactically unambiguous, which is the common case. Such sugar libraries provide productions that extend different parts of the language or extend the same part with different syntax. Furthermore, since desugaring transformations typically only translate a sugar library's new syntax, there is no conflict between desugaring transformations of independent sugar libraries. All sugar libraries presented in this chapter (idiom brackets, arrow notation, EBNF, EBNF metasyntax) are syntactically unambiguous and can be used within the same module.

In case two sugar libraries overlap syntactically, programmers can often use one of the disambiguation mechanisms of SDF [vdBSVV02, Vis97b]. For example, priorities declare precedence of one production over another, whereas reject productions can be used to restrict what can be parsed by a nonterminal. For example, we used reject productions

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

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

5.5.3 Transformation language

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

However, we use Stratego for a good reason. As previously discussed in Section 5.2 and Chapter 2, the definition of a single Stratego rule can be separated into multiple equations that are located in different modules. Essentially, each equation corresponds to a pattern-matching case that can fail or succeed. When applying a transformation rule, Stratego tries each equation currently *in scope* until one succeeds or all have failed [VBT98, HKGV10]. SugarHaskell makes heavy use of this extensibility mechanism.

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

until a fixed point is reached. Hence, a sugar library can also desugar into an AST that another sugar library handles.

5.5.4 Referential transparency

Hygienic transformations enable the transparent use of names in code transformations and avoid two potential conflicts [CR91]. First, when generating code that refers to a variable, this variable may not be captured at the transformation's call site. Instead, the variable must be resolved in the context of the transformation's definition. For example, the IdiomBrackets sugar library from Section 5.1 generates references to pure and (<*>). This should be transparent to users of the sugar library and should not interfere with local declarations of functions of the same name. Second, a name capture can occur when a transformation introduces new variable bindings. These bindings may not capture any variables at the transformation's call site.

SugarHaskell does not support referential transparency. Hence, sugar libraries may produce accidental name capture. However, we employ the convention of fully qualified names, which at least avoids most potential naming conflicts of the first category. For example, in the IdiomBrackets sugar library from Section 5.1, we in fact generate references to Control.Applicative.pure and Control.Applicative.(<*>) as well as a qualified import of Control.Applicative. In our experience, this convention makes unhygienic transformations much less harmful.

However, a clean solution to hygiene is desirable. Unfortunately, we cannot directly apply existing solutions to hygiene [DHB92] as known from macro systems such as Scheme [SDF+09]. The reason is threefold. First, hygienic macro expansion relies on the compositionality of macros. However, our program transformations are more flexible and can affect a syntax tree non-locally. Second, we want to support user-defined binding mechanisms that do not necessarily translate into a binding of the base language. Therefore, we cannot infer variable scoping in the sugared syntax from the desugaring. Third, since we pretty print and compile regular Haskell code, we cannot enhance identifiers with context information; ultimately, each identifier must be represented as a simple string. We plan to investigate these issues in our future work.

5.5.5 Type-awareness

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

- Sugar libraries could declare extension-specific error messages in case the generated

code fails to type-check. One interesting avenue of future work is to analyze the applicability of type-inference instrumentation [HHS03] to achieve extension-specific error messages.

- Type-dependent transformations could be used to generate specialized code for input of certain types, for example, to increase efficiency or to circumvent run-time ad-hoc polymorphism.
- Type-based syntax disambiguation [BVVV05] could be used to select a parse tree in case there is a syntactic ambiguity. For example, arrow notation would not need a separate syntactic category `command` since arrows can be distinguished by type. Similarly, the EBNF metasyntax disambiguation `|[ Exp | ... ]|` would often be unnecessary because the expected syntactic category is implied by the expected type.

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

5.6 Related work

Syntactic extensibility has been the focus of researchers for a long time, from macro processors [McI60, Lay85, THSAC+11], to attribute grammars [Knu68, VBGK10], extensible compiler frameworks [EH07a, NCM03], and language workbenches [KV10, VS10]. We discuss the relation of our approach to existing works in detail in Chapter 8. In a nutshell, we are different from most other approaches because our syntactic extension can use the full class of context-free languages, our extensions compose, and our extensibility mechanism is self-applicable. Here, we focus on related work that is more specific to Haskell.

5.6.1 TemplateHaskell

GHC supports compile-time metaprogramming with the *TemplateHaskell* language extension [SP02]. TemplateHaskell supports arbitrary compile-time computation via TemplateHaskell macros. They are written in Haskell and invoked explicitly with a special call syntax `$(...)`. Macros can only be called in a fixed set of syntactical contexts.

TemplateHaskell is tightly integrated with GHC, and macros can even access GHC's typing environment to analyze the program currently being compiled. TemplateHaskell is therefore not available for other Haskell compilers.

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

GHC also supports a limited form of syntax extension via *quasiquotation* [Mai07]. Syntax extensions are specified by writing a quasiquoter in Haskell, that is, essentially a stand-alone TemplateHaskell macro of type `String -> Q Exp`. The new syntax is used by explicitly invoking the quasiquoter with special call syntax `[foo|...|]`. The part ... can be arbitrary text and is processed by the quasiquoter `foo` at compile time. Quasiquotation is only available in a fixed set of syntactical contexts. Quasiquotation nests badly, because the outer quasiquoter would need to implement the quasiquotation mechanism manually in order to correctly handle the inner quasiquoter.

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

5.6.2 Preprocessors

The Haskell toolbox contains numerous preprocessors. The Haskell platform[4], a collection of blessed Haskell libraries and developer tools, includes the following preprocessors: the parser generator Happy[5], the lexer generator Alex[6], and hsc2hs[7], a generator for bindings to C functions. The Haskell Common Architecture for Building Applications and Libraries (Cabal)[8], the most common build and distribution system for Haskell, additionally supports two other binding generators (c2hs[9] and greencard[10]) as well as cpphs[11], a reimplementation of the C preprocessor with better support for Haskell's

[4] `http://hackage.haskell.org/platform/`
[5] `http://www.haskell.org/happy/`
[6] `http://www.haskell.org/alex/`
[7] `http://www.haskell.org/ghc/docs/latest/html/users_guide/hsc2hs.html`
[8] `http://www.haskell.org/cabal/`
[9] `http://www.cse.unsw.edu.au/~chak/haskell/c2hs/`
[10] `http://hackage.haskell.org/package/greencard`
[11] `http://projects.haskell.org/cpphs/`

lexical syntax. The standard C preprocessor is directly supported by GHC and, on Windows, even distributed with GHC.

The Strathclyde Haskell Enhancement (SHE)[12] is a handwritten preprocessor for Haskell. It is not based on a complete layout-sensitive Haskell parser, but on a lexer with layout heuristics. We have modeled our implementation of idiom brackets after SHE's implementation.

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

Priebe proposes a light-weight framework to implement preprocessors using Template Haskell [Pri05]. His key idea is to use an universal preprocessor that wraps a Haskell source file in a call to a Template Haskell macro. The actual preprocessing is then done by the macro, which can be defined in a library. Unlike SugarHaskell, Priebe's approach does not address syntactic extensions or the composition of different preprocessing libraries. Nevertheless, the idea of combining a preprocessor (to define concrete syntax) and Template Haskell (to define desugarings) seems promising. Future work could investigate whether and how such a combined approach can be implemented as a SugarHaskell library.

The Utrecht Haskell compiler (UHC) [DFS09] is an extensible compiler for Haskell. It is heavily based on preprocessors that compose implementation fragments for different language levels. Extensions have to be compiled into UHC. Parsing is implemented with a hand-written combinator parser. In contrast, SugarHaskell supports extensions as libraries and declarative grammar extensions.

5.7 Chapter summary

Syntactic concerns are important for programmers in practice. While semantics make code run, it is syntax that programmers interact with every day. Therefore, we believe it is important to support programmers in describing not only what their programs do, but also how their programs look. SugarHaskell addresses this belief and provides programmers with syntactic extensibility that allows extensions to use the full class of context-free languages enriched with layout sensitivity. SugarHaskell extensions compose and can

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

affect object language and metalanguages equally easily. While there are some open issues regarding integration with Cabal, HackageDB, and Haskell's type system, SugarHaskell is operational and we invite programmers and language designers to experiment with SugarHaskell and its IDE.

In the development of SugarHaskell, we generalized the SugarJ compiler to a framework for library-based language extensibility that can support different base languages. So far, we instantiated the framework for Java, Prolog, Haskell, and F_ω. However, Haskell is special because the Haskell community seems to be open toward syntactic extensibility, as the large number of existing syntactic preprocessors and compiler extensions suggests. For example, the most recent release 7.6.1 of GHC [13] defines new syntactic sugar for lambda expressions with case distinction (*lambda-case*) and if expressions with more than two branches (*multi-way if-expressions*). Therefore, we believe that the Haskell community is more likely to adopt a system like SugarHaskell, which would enable us to substantiate our evaluation of sugar libraries through the feedback of others.

[13] `http://www.haskell.org/ghc/docs/7.6.1/html/users_guide/release-7-6-1.html`

6 Polymorphic Domain Abstraction and Communication Integrity

SugarJ and its variants provide programmers with flexible extensibility for the syntax, static analysis, and editor support of the host language. In a sense, SugarJ resembles a macro system with particularly flexible macro-application syntax: Programmers can select any context-free syntax to trigger a desugaring transformation. As consequence, and similar to other macro systems, domain-specific syntax is strongly coupled to the desugaring transformation that translates it into base syntax: SugarJ does not fulfill our design goal on polymorphic domain abstraction.

SugarJ furthermore lacks referential transparency and hygienic code generation, as discussed in the previous chapter. Hygiene is a hard problem for SugarJ because of the use of the general-purpose transformation language Stratego, whose expressiveness hinders the application of standard approaches known from macro systems with more restricted macro-expansion engines as, for example, used in Scheme [DHB92, CR91]. Therefore, SugarJ currently can neither prevent the generation of unhygienic references to artifacts outside the lexical scope of the generator, nor the generation of unhygienic bindings that captures references outside the lexical of the generator: SugarJ does not fulfill our design goal on referential transparency.

In this chapter, we approach polymorphic domain abstraction and referential transparency by reviewing and revising SugarJ from the perspective of model-driven development (MDD) and software architecture. We present a novel programming paradigm called model-oriented programming, which is both a framework for MDD and a programming language. Like in MDD, model-oriented programming supports the decomposition of software systems into models and transformations. In particular, in model-oriented programming, programmers can apply multiple transformations to the same model. This enables polymorphic domain abstraction. Unlike MDD, models and transformations are tightly integrated into the module system of the programming language. From the perspective of MDD, the most distinguishing feature of model-oriented programming is *communication integrity* [MQR95, LV95, ACN02], which we adopted from the field of software architecture. Communication integrity ensures that dependencies between modules are explicit in the original source code and that transformations cannot introduce or manipulate dependencies. Consequently, communication integrity enables programmers to reason about module references transparently. This is a promising first step toward referential transparency for other kinds of references such as variable occurrences.

We designed and implemented a programming language for model-oriented program-

ming on top of the Java-based SugarJ compiler (Chapter 2); our framework for library-based language extensibility (Chapter 5) currently does not support model-oriented programming, which is why we refer to our model-oriented programming language as JProMo (Java Programming with Models). In this chapter, we describe the design of JProMo, present a formal semantics for a core of JProMo, and demonstrate the expressiveness and applicability of JProMo through case studies.

6.1 Introduction

Increasing the level of abstraction in software development has been a permanent research goal since the beginning of programmable computers. A recent trend toward this goal is MDD [CH06, KBJV06], which, in the context of this work, is understood as the idea to decompose a software system into models, metamodels, and transformations. Metamodels represent domain-specific abstractions; models that conform to a metamodel represent particular instances of the abstraction. Transformations give meaning to a model by translating it (directly or via intermediate metamodels) to a metamodel whose meaning is already given (such as the Java programming language). A model is not coupled to a particular transformation but can be reused multiple times with different, independent transformations.

While the basic idea of MDD is quite powerful [Béz05], it is not obvious how it fits to basic principles from software architecture and component-based software development. For instance, it is not clear how to structure such software systems hierarchically into layers of abstraction, how to compose them from reusable parts, or how to compile and reason about them in a modular and compositional way. Also, ordinary programs (whether written by a programmer or generated by a transformation) do not seem to fit very well into the MDD idea, which has led to a significant gap between programming and modeling [MMP10].

The goal of our approach, called *model-oriented programming*, is to improve MDD with regard to these issues by tightly integrating models and transformations into a programming language. In model-oriented programming, models, metamodels, and transformations are represented as libraries and *all* dependencies are explicitly declared by import statements. In particular, a dependency on the result of applying a transformation Trans to a model Model is denoted by the import statement **import** Model<Trans>. The most significant consequence of the explicit representation of dependencies is that model-oriented programming guarantees *communication integrity* [MQR95, LV95, ACN02], which means that a module only depends on imported modules; transformations cannot inject module dependencies. Communication integrity is a cornerstone for modular program understanding and an important first step toward referential transparency.

We present the design and implementation of model-oriented programming language for Java called JProMo. JProMo builds on our work on syntactic extensibility and the SugarJ

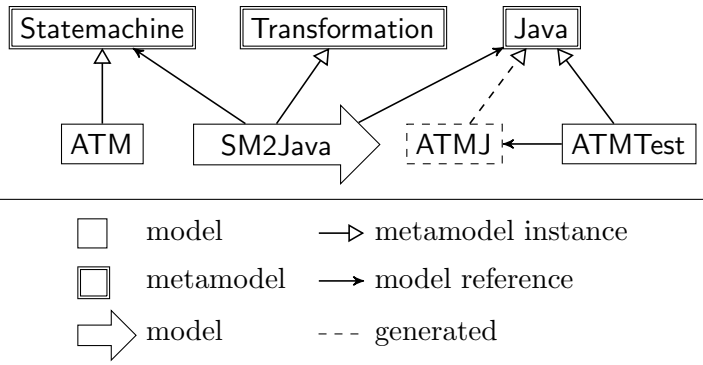


Figure 6.1: ATM statemachine with transformation to Java and a test suite.

programming language (Chapter 2). JProMo goes beyond SugarJ regarding its first-class support for models and transformations, explicit application of transformations in import statements, and guarantees for separate compilation and communication integrity. In this chapter, we make the following contributions:

- We discuss the deficiencies of MDD from the perspective of component-based software development and motivate model-oriented programming as an integration of MDD into a programming language (Section 6.2).
- We give an introduction to model-oriented programming, formalize its semantics, and prove communication integrity and separate compilation theorems (Section 6.3 and 6.4).
- We present a Java-based model-oriented programming language called JProMo that adheres to the formal properties of model-oriented programming (Section 6.5).
- We demonstrate the expressiveness and applicability of JProMo through three case studies that apply model-oriented programming for software decomposition, metamodeling, and the encoding of #ifdef-based product lines (Section 6.6).

6.2 Requirements for model-oriented programming

To motivate requirements for model-oriented programming, we consider a typical kind of model in MDD: a finite statemachine. Figure 6.1 illustrates the typical components in an MDD scenario to support finite statemachine models. We designed a visual notation that makes all dependencies between components explicit:

1. The ATM model is an instance of the Statemachine metamodel, the model transformation SM2Java is an instance of the Transformation metamodel, and ATMJ and

ATMTest are instances of the Java metamodel.[1]

2. The transformation SM2Java depends on the source and target metamodels, which is expressed via model references.
3. The ATMJ model (dashed box) has been generated by applying SM2Java to the ATM model (solid box). The generated ATMJ model thus depends on the original ATM model and the transformation.
4. The Java model ATMTest uses the generated ATMJ Java model to execute test cases. ATMTest thus depends on a model that must be generated.

Conventional MDD frameworks allow ATMTest to declare a dependency only on ATMJ, e.g., by importing the generated ATMJ Java class. Actually, however, ATMTest depends on ATM and SM2Java. For instance, if either of these components changes, the change may affect ATMTest. In other words, conventional MDD frameworks violate *communication integrity*, which postulates that each component in the implementation of a system may only depend on those components to which it is explicitly connected. Communication integrity has been recognized as a pillar of component-based software architecture [MQR95, LV95, ACN02].

The only way to find out about these dependencies is to consider the build scripts (often called workflows or generator models), which specify that SM2Java should be run on input ATM and that the result should be called ATMJ. Conversely, the build scripts specify that ATMJ does not depend on any other models or transformations that may exist in our software system. However, since build scripts are global entities, the dependency on the generation process remains implicit and nonmodular.

The lack of communication integrity is important from the perspective of component-based software architecture. For instance, it becomes hard to understand the impact of changes to one component on the rest of the system. It also prevents abstraction: Ideally, the programmer of ATMTest should only need to reason about the interfaces and documentation of ATM and SM2Java. Looking at the structure and details of the generated ATMJ code violates the abstraction barrier which the statemachine model is supposed to maintain.

Reasoning about dependencies in conventional MDD frameworks becomes even harder when one considers the dependencies of generated models. For instance, ATMJ may depend on a Java library for statemachines. However, this dependency cannot be seen by considering the dependencies of ATM and SM2Java. Rather, this dependency is *generated* and hence hidden in implementation details of SM2Java. The dependency might also depend on details of ATM; for instance, based on the size or structure of the statemachine, SM2Java may generate a dependency on a different library. Hence, the overall dependency structure can only be seen by a closed-world assumption after all transformations have been executed in the complete software project. Furthermore,

[1] We treat Java programs as models, too, hence the terms 'Java model' and 'Java program' are equivalent.

the dependency structure is highly fragile, because it can depend on implementation details of models or transformations. Fragile, implicit dependencies are at odds with basic software architecture principles, in which dependencies are seen as a high-level architectural concern [AG94]. From this discussion, we derive our first requirement for model-oriented programming:

(R1) Model-oriented programming must guarantee communication integrity.

This lack of modularity also has negative technical consequences, most notably a lack of separate compilation: Since the dependencies are not explicit and may in fact arise during transformation application, the build process is global. Kuhn et al. [KMT12] report that developers complain that long build cycles, which often take multiple hours in bigger MDD projects, prevent 'live modeling' and consequently more effective adoption of modeling. Consequently, our second requirement for model-oriented programming is:

(R2) Model-oriented programming must enable separate compilation.

A related problem, which has been expressed many times in the literature, is that MDD entails a gap between modelling and programming [MMP10, HJSW09, FL08]. In our example, we glossed over the difference between models and normal programs, but in all MDD environments we know, the two are rather separated. For instance, there is no integration between the dependency management by the module system of the programming language (such as import statements), and dependency declarations in models. Although it is simple to convert back and forth between programs and a representation of the program as a model, it is not obvious how models and code can be composed with each other and interact with each other in a principled, explicit way. Ordinary programs and their dependencies are somewhat external to the MDD methodology. Consequently, it is not surprising that developers are worried that models and code become inconsistent [FL08]. Henceforth, we derive:

(R3) Model-oriented programming must tightly integrate models and code by a common dependency management and provide the possibility to intermingle models and code in a principled way.

A final architectural concern about model-oriented programming is its applicability at different metalevels. Kuhn et al. report that MDD developers often have no tool support to create tools similar to the ones they use themselves [KMT12]. MDD tools such as EMF [SBPM08] also suggest a strict stratification into metalevels; for instance, using a generated editor entails starting a new Eclipse instance with a separate workspace. For scalability, we believe that it is important that everything is a model [Béz05], including programs, transformations, and metamodels, such that the programming model stays the same regardless of the metalevel. For instance, it should be straightforward to have

higher-order transformations, to transform metamodels, or to generate editor support for metamodels. Model-oriented programming shares this design goal with SugarJ:

(R4) Model-oriented programming must be uniformly applicable across metalevels.

Since communication integrity is a quite strong restriction, it is not clear whether it will have negative consequences for the expressiveness of model-oriented programming with regard to partitioning a large application into parts. Therefore, our last requirement is:

(R5) Model-oriented programming must be applicable for component-based software development in practical applications.

6.3 Model-oriented programming with JProMo

In this section, we give an overview about model-oriented programming with JProMo, which satisfies above requirements. We exemplify JProMo by modeling an ATM for money withdrawal as a statemachine. The complete definition of the ATM model is shown in Figure 6.2. The ATM model specifies an initial state Init, a set of events the ATM reacts to, and a set of states. Each state describes a partial transition function from events to target states.

JProMo employs domain-specific textual syntax to describe models. The domain-specific syntax is part of the metamodel, which also describes the abstract syntax for representing models internally. A JProMo model refers to its metamodel explicitly using an import statement. Besides expressing the metamodel dependency, the import statement also activates the metamodel's domain-specific syntax in the current JProMo file. The domain-specific syntax is declared by a grammar as part of a metamodel declaration. For example, here is an excerpt of the statemachine metamodel:

```
package statemachine;
public metamodel Metamodel {
  context-free syntax
    Statemachine                                  -> ToplevelDeclaration
    Mod* "statemachine" Id "{" SMBody "}"         -> Statemachine
    InitialState EventsDec* StateDec*             -> SMBody
    ...
}
```

Like SugarJ, JProMo uses the grammar formalism SDF [Vis97b]. Beyond context-free syntax, a metamodel can also specify domain invariants in the form of domain-specific static analyses. As in SugarJ, JProMo import statements activate the context-free syntax and static analyses of a metamodel library.

```
package banking;

import statemachine.Metamodel;

public statemachine ATM {
  initial state Init

  events DoWithdraw, Cancel, PinOK, PinNOK, [...]

  state Init {
    DoWithdraw => Withdraw
    Cancel => Init
  }
  state Withdraw {
    PinOK => GiveMoney
    PinNOK => RevokeCard
    Cancel => Init
  }
  state GiveMoney { MoneyTaken => ReturnCard }
  state ReturnCard { CardTaken => Init }
  state RevokeCard { CardRevoked => Init }
}
```

Figure 6.2: Model of an ATM statemachine in JProMo.

As usual in MDD, JProMo metamodels do not declare the semantics but only the syntax and invariants of a domain. Instead, a semantics is formalized by a model transformation converting a model from a source metamodel to a target (or the same) metamodel. Like SugarJ, JProMo uses the Stratego term rewriting language [VBT98] for describing model transformation.

Figure 6.3 shows how a transformation from a statemachine model to a Java implementation of the statemachine looks like. The transformation generates a method `step` for firing events. To write a test case for the ATM statemachine, we want to use the generated Java representation of the ATM and run it on a sequence of events as illustrated by the method `test` in Figure 6.4.

Figure 6.4 also illustrates our solution to support uniformity and communication integrity: JProMo organizes models, metamodels, and model transformations uniformly as libraries, which prevents an undesirable heterogeneous stratification. For instance, a transformation could also transform or output another transformation or a metamodel. For communication integrity, the application of a model transformation is

```
package statemachine;

import transformation.Compile;
import statemachine.Metamodel;
import org.sugarj.languages.Java;

public transformation SM2Java {
  main = compile-after(sm-to-java)

  sm-to-java :
    CompilationUnit(pkg, imps, Statemachine(init, events, states)) ->
    CompilationUnit(pkg, imps, JavaClass(...<map(state-to-class)>...))

  state-to-class :
    |[ state ~sname { ~transitions } ]| ->
    |[ class ~sname implements State {
         public State instance = new ~sname();
         public State step(Event e) {
           ~(<map(event-handle)> transitions)
           return null;
         }
       }
    ]|

  event-handle :
    |[ ~ename => ~target ]| ->
    |[ if (e == ~name) return ~target.instance; ]|
}
```

Figure 6.3: Transformation from statemachines to Java code.

specified with an import statement as part of the client of a generated model: We write **import** Model<Trans> **as** Ident to declare a dependency on the result of applying Trans to Model. A client thus does not depend on any external modeling artifacts except for the ones explicitly declared with import statements. In conventional MDD frameworks, the generation of ATMJ would be specified in a build script, and ATMTest would import ATMJ by name. JProMo avoids such fragile dependencies and enables modular reasoning.

The state-machine case study illustrates the spirit of model-oriented programming: We use model-based domain abstraction where useful, but write pure application code in Java where appropriate. In model-oriented programming, model-driven and code-driven

```
package banking;

import banking.ATM<statemachine.SM2Java> as ATMJ;

public class ATMTest {
  public void test() {
    ATMJ machine = new ATMJ();
    machine.step(machine.event_DoWithdraw());
    machine.step(machine.event_PinOK());
    machine.step(machine.event_MoneyTaken());
    machine.step(machine.event_CardTaken());
    assert machine.currentState() == machine.state_Init();
  }
}
```

Figure 6.4: ATMTest depends on a generated Java implementation of ATM.

development are fully integrated and connected through the unifying library concept. Due to the uniform handling of dependencies and transformation application by imports, communication integrity is maintained and all concepts are applicable across metalevels. Our formalization will illustrate how exactly this works and why we can in fact guarantee these properties.

6.4 Formalization

The previous subsection illustrates model-oriented programming by example of our Java-based realization JProMo. However, to abstract from details of our implementation and to describe model-oriented programming in its full generality, we specify an abstract core of model-oriented programming that illustrates the meaning of models, transformations, and imports, and is sufficient to state and prove communication integrity and separate compilation.

Figure 6.5 shows the syntax and semantic domains of our abstract core language for model-oriented programming. Programs are organized in modules m. A module consists of a sequence[2] of names $\overline{n}$ bound to a sequence of module imports $\overline{i}$ in the module's body e. For instance, the import statement from Figure 6.4 would be written as ATMJ = ATM<SM2Java>. An import references another module either by name n, or by transformation $i_1\langle i_2\rangle$ of a model i_1 with a transformation i_2. The syntax of a module body e is not of relevance for the formal development; we leave it unspecified. In JProMo,

[2]We use the standard notation of writing $\overline{x}$ for a sequence $x_1 \ldots x_n$

Syntax:

n	$\in$ Name	
m	$::= (\overline{n} = \overline{i}$ in $e)$	modules have imports and a body
i	$::= n \mid i\langle i \rangle$	import by name, import transformed
e	$::= \ldots$	module body left abstract

Semantic domains:

$\mathbb{D}$	$= \mathbb{M} \times (\mathbb{B} + \mathbb{T} + \{\bullet\})$	
$\mathbb{B}$	$= \ldots$	base semantics left abstract
$\mathbb{M}$	$= m \times \Gamma$	models close over the dependencies of a module
$\mathbb{T}$	$= \mathbb{M} \to \mathbb{D}_\perp$	transformations map models to semantic artifacts
Γ	$=$ Name $\to \mathbb{D}_\perp$	environments

Figure 6.5: Syntax and semantic domains of model-oriented programming.

e includes Java programs, transformations, and grammars.

We specify the semantics of our core language for model-oriented programming as a denotational semantics, that is, by a compositional mapping of each program to a mathematical object of the corresponding semantic domain [Sto77]. Our semantics is a *compile-time* semantics; the dynamic semantics of the final program is not in the scope of our formalization.

The semantic domain of modules is $\mathbb{D}$: Each module is mapped to a pair composed of other semantic domains. $\mathbb{B}$ stands for the semantic domain of compiled application code, which we leave abstract in the formalization. For JProMo, $\mathbb{B}$ would correspond to the domain of Java class files. $\mathbb{M}$ stands for the semantic domain of models, which consist of a syntactic module representation and an environment that provides mappings for the dependencies of the module. In analogy with function closures, we say that a model closes over the dependencies of a module. Transformations $\mathbb{T}$ are functions that map models to semantic artifacts. Since a transformation may fail, we allow the error value $\perp$ as a result of a transformation. Coming back to the semantic domain of modules $\mathbb{D}$, the first component denotes the model corresponding to the closed module and is always present, whereas the second component depends on whether the module describes application code, a transformation, or a model. In the latter case, we do not require a second representation of the model and use the unit element $\bullet$.

The design of the semantic domains already illustrates how we realize *uniformity*. First, every semantic artifact from $\mathbb{D}$ can be reified as a model to serve as the input of a model transformation. In addition to models, this includes regular application code such as a Java class in JProMo. We realize this by accompanying each semantic artifact with an explicit model representation. Second, a model transformation from $\mathbb{T}$ can produce any semantic artifact. In particular, we support higher-order transformations, that is,

Semantics:

$$\textit{sem-mod} \;:\; m \times \Gamma \to \mathbb{D}_\perp$$

$$\textit{sem-mod}(\overline{n} = \overline{i} \text{ in } e, \rho) = \begin{cases} \perp, & \text{if } \perp \in \overline{d} \;\text{ or }\; \textit{body} = \perp \\ (m, \textit{body}), & \text{otherwise} \end{cases}$$

$$\text{where } \; d_x \in \overline{d} = \textit{resolve}(i_x, \rho) \text{ for } i_x \in \overline{i}$$
$$\sigma = \textit{mkenv}(\overline{n}, \overline{d})$$
$$\textit{body} = \textit{sem-exp}(e, \sigma)$$
$$m = (\overline{n} = \overline{i} \text{ in } e, \sigma)$$

$$\textit{sem-exp} \;:\; e \times \Gamma \to (\mathbb{B} + \mathbb{T} + \{\bullet, \perp\})$$
$$\textit{sem-exp}(e, \rho) = \ldots$$

$$\textit{resolve} \;:\; i \times \Gamma \to \mathbb{D}_\perp$$

$$\textit{resolve}(i, \rho) = \begin{cases} \rho(n), & \text{if } i = n \\ d_2(m_1), & \text{if } i = i_1\langle i_2 \rangle \\ & \text{and } (m_1, d_1) = \textit{resolve}(i_1, \rho) \\ & \text{and } (m_2, d_2) = \textit{resolve}(i_2, \rho) \\ & \text{and } d_2 \in \mathbb{T} \\ \perp, & \text{otherwise} \end{cases}$$

$$\textit{mkenv} \;:\; \overline{n} \times \overline{\mathbb{D}} \to \Gamma$$
$$\textit{mkenv}(\varepsilon, \varepsilon) = \lambda n.\, \perp$$
$$\textit{mkenv}(n \cdot \overline{n}, d \cdot \overline{d}) = \lambda n'.\, \begin{cases} d, & \text{if } n = n' \\ \textit{mkenv}(\overline{n}, \overline{d})(n'), & \text{otherwise} \end{cases}$$

Figure 6.6: Denotational semantics of model-oriented programming.

transformations producing other transformations. Since transformations also need to produce a transformed model (the first component of $\mathbb{D}$), it is also possible to compose transformations sequentially.

We provide the denotation semantics of our core language in Figure 6.6. Function *sem-mod* defines the semantics of modules. It accepts a module and an environment and yields an element of $\mathbb{D}$. *sem-mod* first resolves each import $i_x \in \overline{i}$ of the module using function *resolve*. For a named import $i = n$, function *resolve* looks up the name in the environment. For an application $i_1\langle i_2 \rangle$, the function resolves i_1 and i_2, and, if i_2 resolves to a transformation, applies this transformation to the model reification of i_1. *sem-mod* uses the resolved imports $\overline{d}$ to construct an environment σ that binds the imported artifacts to the names $\overline{n}$. This environment only contains artifacts explicitly referenced through imports, i.e., the domain of σ is $\overline{n}$. *sem-mod* uses σ to evaluate the

body of the module with *sem-exp*, which we leave unspecified. If the evaluation of any of the imports or the module body fails (yielding $\bot$), *sem-mod* yields $\bot$ itself. Otherwise, *sem-mod* yields a pair consisting of the module's closure and the result of evaluating the module body.

We deliberately designed the semantics in a way that achieves *communication integrity*. This required two design choices. First, we chose a representation of models that is closed over external dependencies. All dependencies of the encapsulated module can be resolved within the accompanying environment. This is similar to lexical scoping of functions, which is typically achieved by a closure consisting of the function definition and its lexical environment. Second, in contrast to other MDD frameworks, we require transformations to produce semantic artifacts instead of syntactic artifacts. Typically, a model-oriented-programming transformation achieves this by first applying a syntactic rewriting and then calling the compiler (function *sem-mod*) on the resulting artifact. However, in the call to *sem-mod*, the transformation has to provide an environment, too. Since the transformation does not receive any input apart from the original model, this environment can only map to artifacts available within the input model or the transformation itself.

Therefore, communication integrity holds: It is not possible for a transformation to inject implicit communication channels between modules. We formalize this in the following. First, we define function *deps-mod*, which computes the explicit dependencies of a module, which are locally declared with imports:

$$
\begin{aligned}
&\textit{deps-mod} \;:\; m \to 2^{\mathrm{Name}} \\
&\textit{deps-mod}(\overline{n} = \overline{i} \text{ in } e) = \textstyle\bigcup_{i_x \in \overline{i}} \; \textit{deps-imp}(i_x)
\end{aligned}
$$

$$
\begin{aligned}
&\textit{deps-imp}(n) && = \{n\} \\
&\textit{deps-imp}(i_1\langle i_2 \rangle) && = \textit{deps-imp}(i_1) \cup \textit{deps-imp}(i_2)
\end{aligned}
$$

Communication integrity then states that the semantics of a module only depends on these explicitly declared dependencies.

Lemma 1. *For all imports i and environments ρ and σ, if $\rho|_{\textit{deps-imp}(i)} = \sigma|_{\textit{deps-imp}(i)}$ then $\textit{resolve}(i, \rho) = \textit{resolve}(i, \sigma)$.*

Proof. By structural induction on i. The base case $i = n$ follows from $\rho|_{\textit{deps-imp}(i)} = \sigma|_{\textit{deps-imp}(i)}$, which entails $\rho(n) = \sigma(n)$. The step case follows directly using the induction hypothesis twice. □

Theorem 1 (Communication integrity)**.** *For all modules m and environments ρ and σ, if $\rho|_{\textit{deps-mod}(m)} = \sigma|_{\textit{deps-mod}(m)}$ then $\textit{sem-mod}(m, \rho) = \textit{sem-mod}(m, \sigma)$.*

Proof. By Lemma 1, each $d_x \in \overline{d}$ from the definition of *sem-mod* evaluates to the same value under ρ and σ. Since the body of the module is evaluated under the constructed

environment $mkenv(\overline{n}, \overline{d})$, the result of *sem-mod* is the same under ρ and σ, independent of the definition of *sem-exp*. □

As direct consequence of communication integrity, we get separate compilation: A module m can be compiled in separation of any other module n if m is independent of n.

Theorem 2 (Separate compilation)**.** *For all modules m, environments ρ, and names n , if $n \notin$ deps-mod(m) then sem-mod$(m, \rho) =$ sem-mod$(m, \rho|_{dom(\rho)\setminus\{n\}})$.*

Proof. By Theorem 1 with $\sigma = \rho|_{dom(\rho)\setminus\{n\}}$. □

JProMo complies to these formal properties and implements and refines the abstract semantics of model-oriented programming, as we see next.

6.5 Technical realization of JProMo

JProMo is a model-oriented programming language with application code written or generated in Java. We developed a compiler for JProMo. Similar to SugarJ (Chapter 2), the compiler resolves imports, applies transformations, and compiles the resulting Java code to provide an executable to the user.

However, in contrast to SugarJ compiler, the JProMo compiler conforms to the specification of model-oriented programming: It supports polymorphic domain abstraction, guarantees communication integrity, and allows the reification of any software artifact as a model. However, the implementation of the compiler deviates from above semantics in one important aspect. There is no environment of dependencies in a compiler; a compiler looks up and compiles source files from the *source path* on demand. Therefore, our compiler cannot satisfy communication integrity by design: A model transformation can generate code that contains imports beyond the dependencies of the original model and transformation. The compiler would resolve such imports from the source path. This violates communication integrity and would circumvent separate compilation and modular reasoning about dependencies. The problem cannot occur in the above semantics, because a transformation has to produce an environment for the dependencies of generated code in order to successfully call *sem-mod*.

The JProMo compiler guarantees communication integrity by checking *after* each transformation application that the result of the transformation has no dependencies beyond those of the original model and transformation. If this check fails, the compiler reports a violation of communication integrity as a compile error to the user. Another difference to the formal semantics is that our compiler also supports circular dependencies of models and Java code.

6.6 Case studies

We designed model-oriented programming as a modeling framework with desirable requirements such as communication integrity, separate compilation, and uniformity. In Section 6.4, we have formally validated that model-oriented programming fulfills communication integrity (R1), separate compilation (R2), and uniformity (R4) from Section 6.2. In this section, we experimentally validate the integration of code and models (R3), uniformity (R4), and in particular the applicability (R5): We illustrate that model-oriented programming is useful and expressive enough for practical component-based development with models.

The purpose of the case studies is to generate evidence for the following three hypotheses: (i) A model-oriented-programming application can be decomposed into independent and reusable parts. (ii) Modeling at the metalevel is useful and practical. (iii) Modeling and programming can be tightly intertwined. We have conducted one case study for each of these hypotheses.

6.6.1 Model-oriented software decomposition

To illustrate model decomposition, we develop an entity schema for managing banking entities. Such a schema may be concerned with hundreds of different concepts, from customers, to bank employees, ATMs, and different kinds of credit and debit cards. To develop, maintain, and use such a large schema effectively, a decomposition into multiple smaller schemas seems unavoidable.

To divide entity schemas into subcomponents, we map each domain concept into a single library. For instance, we represent a schema for customers as one library, and a schema for accounts as another library. However, these concepts are not independent because a customer has a collection of accounts and each account has a corresponding customer. We use imports to model the functional dependencies of customers and accounts as shown in Figure 6.7(a) and 6.7(b). In fact, if we left out these imports, communication integrity would guarantee us that customers and accounts do not interact in any way.

The Account and Customer schemas instantiate the same entity metamodel. However, it is also possible to connect models that conform to different metamodels, as exemplified in Figure 6.7(c). The statemachine DataATM instantiates a metamodel that integrates data-dependent features from the entity metamodel into the statemachine metamodel, which we already encountered in Section 6.3. In addition to regular state machines, a data-dependent statemachine can manage and act upon internal as well as event-supplied data. For example, DataATM declares internal data fields using the data keyword followed by a property declaration. The acc field stores the account that is served during a withdrawal, while pinCount totals the number of failed pin requests. Since DataATM depends on the account schema, we use an import to make this dependency explicit.

```
package banking.entity;

import entity.Metamodel;
import banking.entity.Customer;

public entity Account {
  uid     :: Integer
  owner   :: Customer
  balance :: Integer
  pin     :: String
}
```

(a) A schema for bank accounts.

```
package banking.entity;

import entity.Metamodel;
import banking.entity.Account;

public entity Customer {
  name     :: String
  address  :: String
  accounts :: Set<Account>
}
```

(b) A schema for bank customers.

```
package banking;

import statemachine.data.Metamodel;
import banking.entity.Account;

public statemachine DataATM {
  initial state Init

  data acc :: Account
  data pinCount :: Integer

  events InsertCard(Account), Pin(String), [...]

  state Init {
    InsertCard(clientAcc) => Withdraw {
      acc := clientAcc
      pinCount := 0
    }
  }
  state Withdraw {
    Pin(p) if p == acc.pin                    =>  HowMuch
    Pin(p) if p != acc.pin && pinCount < 2    =>  Withdraw { pinCount := pinCount + 1 }
    Pin(p) if p != acc.pin && pinCount >= 2   =>  RevokeCard
  }
}
```

(c) A data-dependent ATM that instantiates the composed metamodel and references the Account model.

Figure 6.7: JProMo supports the decomposition of an application into multiple interlinked models that conform to different metamodels.

```
package statemachine.data;

import entity.Entity2Java;
import statemachine.SM2Java;
import transformation.Recursive;

import statemachine.data.Metamodel;
import org.sugarj.languages.Java;

public transformation DataSM2Java {
  main = recursively-transform(
    switch get-metamodel
      case ?"entity.Metamodel":
        main-entity_Entity2Java
      case ?"statemachine.data.Metamodel":
        ?model;
        main-statemachine_SM2Java;
        handle-internal-data-decls(|model)
    end)

  handle-internal-data-decls(|model) = ...
}
```

Figure 6.8: Transformation for interlinked models can be built by extending and reusing existing transformations.

Data-dependent statemachines represent a deep composition of entity schemas and statemachines. For example, the entity metamodel declares an expression language for querying and manipulating with schema instances. The metamodel for data-dependent statemachines reuses this expression language unchanged in declarations of premises and side-effects of state-machine transitions. To support metamodel composition, we rely on the grammar formalism SDF [Vis97b] and its support for language unification. Technically, SDF allows a grammar to refer to and extend nonterminal definitions from other grammars. SDF generates a parser for the composed extended grammar that we use to process model instances.

An instance of a composed metamodel refers to concepts from different domains, such as statemachine transitions and entity expressions. Therefore, a transformation for composed metamodels needs to understand the composition and recursively transform referenced models such as the account schema. Figure 6.8 shows such a transformation DataSM2Java, which largely builds on existing transformations. DataSM2Java calls the

auxiliary function recursively-transform that accepts a transformation as argument, tries to apply it to each referenced model recursively, and calls the JProMo compiler on artifacts for which the transformation succeeds. DataSM2Java uses a different transformation depending on a model's metamodel. If the model conforms to the entity metamodel, DataSM2Java simply delegates to the existing transformation main-entity_Entity2Java, which is the main rewrite rule of the imported entity.Entity2Java transformation.[3] If the model conforms to the extended statemachine metamodel, DataSM2Java delegates to the transformation statemachine.SM2Java and only handles constructs that SM2Java cannot handle, such as internal data.

We thus successfully composed two independent metamodels and their transformations for handling models that address multiple domains. We rely on SDF to extend metamodels and Stratego to compose model transformations.

6.6.2 Modeling at higher metalevels

Conventional modeling frameworks provide fixed metamodeling and transformation languages that a developer can not easily configure, let alone replace. Since JProMo organizes metamodels, models, and model transformations as libraries that are models, too, the modeling and transformation mechanism of JProMo is uniformly applicable at all metalevels. In particular, while JProMo provides default metamodeling and model-transformation languages, custom domain-specific metamodeling languages can be used instead.

The default model transformation language of JProMo is the term-rewriting language Stratego. While writing model transformations as term rewrites is feasible, template engines seem to be more prominent in practice. Templates focus more on the generated code, whereas term rewrites primarily follow the structure of the input model to decompose it.

Since model transformations are models, too, we can define a template-engine as a metamodel, use this metamodel to describe templates, and transform the templates into executable model transformations. To illustrate this idea, we have built a template engine with JProMo. In contrast to template engines in other MDD frameworks, our template engine is not built into the JProMo compiler but user-defined within libraries.

Figure 6.9 shows the full entity.Entity2Java template, which transforms an entity model such as Account (Figure 6.7(a)) into a Java class. The template generates a private field with accessor methods for each property. In a template, a dollar sign $ constitutes an escape to Stratego, for example, to reference metavariables or to query the input model with **$for**. Note that collect-one(p) and collect-all(p) are Stratego strategies that retrieve the first, respectively all, subtrees for which the given predicate p succeeds. We applied our template engine to implement all the transformations for statemachines.

[3] Unfortunately, Stratego provides neither qualified names nor a hierarchical namespace, so that we depend on renaming in a global namespace.

```
package entity;

import template.Metamodel;
import entity.Metamodel;
import org.sugarj.languages.Java;

public template Entity2Java {
  $$firstUpper = string-as-chars([to-upper|id])
  $$sort-to-javatype = ...
  $pkg = collect-one(?PackageDec(_,<id>))
  $classname = collect-one(?ModelDecHead(_,<id>))

  package $pkg;
  $*{?CompilationUnit(_,<id>,_)}
  public class $classname {
    $for(Property(name, sort) in collect-all(?Property(_,_))) {
      $type = <sort-to-javatype> sort
      $upName = <Id(first-upper)> name

      private $type $name;
      public $type get#$upname() { return $name; }
      public void set#$upname($type $name) { this.$name = $name; }
    }
  }
}
```

Figure 6.9: JProMo supports custom model-transformation languages in libraries.

Using JProMo's support for higher-order metamodeling, we also defined an alternative metamodeling language that allows the separate specification of abstract and concrete syntax (not shown for brevity). This separation enables the use of different concrete syntaxes for the same metamodel in different parts of a software project. Moreover, it allows developers to write model transformations for a metamodel independent of any concrete syntax. JProMo is expressive enough to support such profound changes to the metamodeling and model transformation languages through user libraries.

6.6.3 Mixing models and code

As final case study, we have built a framework for feature-oriented software development (FOSD) [AK09] using JProMo. A FOSD product line consists of (i) a feature model that

```
package graph;

import variability.model.Metamodel;

public featuremodel GraphFeatureModel  {
  features EdgeImpl, OnlyNeighbors, NoEdges, Weighted, ...

  constraint Connected && EdgeImpl
  constraint Connected -> (Directed xor Undirected)
  constraint EdgeImpl -> (OnlyNeighbors xor NoEdges)
  ...
}
```

(a) Feature model with mutually exclusive features Directed/Undirected and OnlyNeighbors/NoEdges.

```
package graph;

import variability.config.Metamodel;
import graph.GraphFeatureModel;
import graph.GraphFeatureModel<variability.CheckConfig>;

public config DirectedNeighbors for GraphFeatureModel {
  enable Connected, EdgeImpl, Directed, OnlyNeighbors, Weighted, ...
  disable Undirected, NoEdges, ...
}
```

(b) Feature configuration that is valid for GraphFeatureModel.

Figure 6.10: MOP can encode other programming paradigms such as FOSD as libraries.

declares available features and constraints their combination, (ii) feature configurations that determine the activated features and adhere to the constraints of the feature model, and (iii) variable libraries that expresses conditionally included code fragments using the declared features. FOSD is a challenging case study for mixing models and code, because the feature conditions in variable libraries are deeply intertwined with normal program code.

We encode a variable library as a model that connects to other variable and invariable Java libraries through import statements. A feature configuration then corresponds to a model transformation that transforms a variable program into a regular program. Accordingly, feature configurations can be seen as a domain-specific model-transformation language. A feature model gives rise to additional static checks that determine whether

all used feature names are declared and whether a feature configuration adheres to the constraints on feature combinations. With JProMo, all of these static analyses can be generated from a feature model using transformations.

To demonstrate our encoding of FOSD, we asked an undergraduate student to implement a configurable graph library, proposed by others as a standard benchmark for FOSD [LHB01]. Figure 6.10 shows excerpts of the library's feature model and a configuration. The feature model declares that every variant must support the Connected and EdgeImpl features, which entail exactly one of Directed or Undirected, and one of one of OnlyNeighbors and NoEdges. The feature configuration DirectedNeighbors (Figure 6.10(b)) selects and deselects features and satisfies the constraints of the feature model, as checked by the generated static analysis GraphFeatureModel<variability>.

We represent feature conditions in variable libraries with syntactic #ifdef statements. The metamodel variability.Java integrates #ifdef statements into the Java grammar at appropriate places. In contrast to the #ifdef implementation of the C preprocessor CPP, our metamodel only supports disciplined #ifdef statements [LKA11] and all occurrences of #ifdef statements are parsed together with the Java code.

Figure 6.11 shows part of the configurable graph library, declared as a variable class. Within such declaration, we write #ifdef(COND) CODE to conditionally include CODE based on the condition COND. The condition is a Boolean expression over the features declared by the feature model. The graph library uses #ifdef statements to conditionally include the field edges and to describe conflicting method declarations that would be considered duplicate in ordinary Java, because the method declarations share the exact same signature.

We can configure a variable Java class by applying a feature configuration as a transformation to it. For example, the configuration DirectedNeighbors deselects the former and preserves the latter addEdge method declaration of the class Graph. To apply it, we first transform the feature configuration into a model transformation using variability.ConfigTrans, and then apply the resulting transformation to the graph library:

```
import graph.Graph< graph.DirectedNeighbors<variability.ConfigTrans> >;
```

Model-oriented programming is particularly well-suited for encoding #ifdef-based software product lines for two reasons. First, model-oriented programming supports separate compilation and checking of models (which represent variable libraries). Thus, a static analysis can provide immediate feedback to programmers before the product line is fully implemented or configured, by checking each model in separation including its variability. For example, we could employ variability-aware type checking [KATS12] to ensure, without configuring the product line, that no valid configuration of a library contains type errors. Second, model-oriented programming supports the integration of models and regular programs. Therefore, variable programs can be easily integrated into invariable programs through importing a configured variant, that is, by applying a configuration in the import. There are no distinct technical spaces for variable and

```
package graph;

import variability.Java;
import graph.GraphFeatureModel;

import impl.Vertex;
import impl.EdgeIfc;
import impl.Neighbor;

public variable class Graph {
  LinkedList<Vertex> vertices;

  #ifdef(Connected && !NoEdges)
  private LinkedList<EdgeIfc> edges;

  #ifdef(NoEdges)
  public EdgeIfc addEdge(Vertex start, Vertex end, #ifdef(Weighted) int weight) {
    start.addAdjacent(end);
    #ifdef(Undirected)
      end.addAdjacent(start);
    #ifdef(Weighted)
      start.setWeight(weight);
    #ifdef(Undirected && Weighted)
      end.addWeight(weight);
    return (EdgeIfc) start;
  }

  #ifdef(OnlyNeighbors)
  public EdgeIfc addEdge(Vertex start, Vertex end, #ifdef(Weighted) int weight) {
    Neighbor e = new Neighbor(end, #ifdef(Weighted) weight);
    addEdge(start, e);
    return e;
  }

  ...
}
```

Figure 6.11: Excerpt of a variable Java class Graph in JProMo. In ordinary Java, the methods would be duplicate and result in a compilation error.

invariable programs that hinder the development process; model-oriented programming seamlessly integrates domain abstractions like #ifdef with regular programming in a single programming paradigm.

6.7 Discussion and future work

We designed model-oriented programming to bridge the gap between programming and modeling. Specifically, model-oriented programming should provide fundamental programming-language features (communication integrity, separate compilation, modular reasoning) in combination with the flexibility of modeling (multiple transformations, code generation across all metalevels). We review our design in the present section.

Model-oriented programming integrates modeling and programming by (i) representing models, metamodels, and model transformations as programming libraries and (ii) enforcing the explicit declaration of interdependencies between artifacts through library imports. In particular, model-oriented-programming libraries declare dependencies on *generated* code by specifying how this code is generated, that is, by providing the original model and a transformation. This is different from SugarJ, where a fixed desugaring transformation is associated with each domain abstraction. However, like SugarJ, model-oriented programming promotes library imports as its single dependency mechanism and applies imports across metalevels, for example, to generate a model transformation.

Model-oriented programming satisfies the code/model-integration requirement (R3) and the uniformity requirement (R4) from Section 6.2 by design. To validate that model-oriented programming satisfies communication integrity (R1) and separate compilation (R2), we formalized a denotational semantics of model-oriented programming and verified corresponding theorems. Despite these strong requirements, model-oriented programming is expressive and applicable to a large range of problems (R5) as our case studies indicate.

We believe that model-oriented programming represents a significant improvement over previous MDD approaches, there is still a lot of potential for further improvement in future work:

Information hiding. Model-oriented programming supports the decomposition of software into many interconnected artifacts. However, there are no explicit interfaces to communicate the behavior of a generated artifact without exposing its internals. For example, in order to find out the available methods of a generated class, a programmer has to either look into the generated code, or understand the transformation and its input well enough to predict the available methods. Both solutions are at odds with information hiding. While comments can be used as informal interfaces, they are not enforced and therefore likely become outdated when a program evolves. A modular solution would be to derive the interface of the generated entity from the interfaces of the transformation and its input. The main open issue here seems to be to identify what

good interfaces for model-oriented programming look like and how to enforce them. For example, we would like to guarantee that the result of transforming a statemachine to Java defines a step(Event) method. To this end, we want to explore whether previous work on interfaces for type-safe metaprogramming [KO10] can be used for model-oriented programming as well.

Error reporting. JProMo performs separate compilation to compile and check libraries in isolation. Therefore, compile errors are reported per library, which enables developers to locally reason about code to find the error's cause. However, as usual for MDD frameworks, the quality of error messages is rather unsatisfactory: If there is a problem with generated code, such as a Java type error, JProMo reports it in terms of the generated artifact. However, providing high-quality error reports is a hard problem and deserves separate treatment.

Multiple target languages. One goal of MDD that JProMo currently does not achieve is support for multiple code-generation target languages (also known as platform independence). To support separate compilation, the JProMo compiler calls a platform-specific build tool such as *javac* on each library. Therefore, JProMo currently only supports the generation of Java code. As described in the previous chapters, we generalized the SugarJ base implementation to support multiple alternative host languages, such as Haskell for SugarHaskell. However, more work is required to enable the simultaneous use of multiple target languages.

6.8 Related work

Model-oriented programming is related to earlier works on metaprogramming with macros, domain-specific languages, and models.

Macros were an important inspiration for this work, because they illustrate how code transformation can be tightly integrated into programming languages, including explicit dependencies [Fla02]. The main difference between macros and this work is that each macro argument is coupled to a specific macro invocation; there is no notion of a 'model' whose existence is independent of any particular transformation. This simplifies dependency management in macro systems significantly.

Our earlier work on which the implementation of JProMo is partially based, SugarJ, can be understood as a macro system with a particularly powerful macro call syntax. SugarJ allows programmers to define syntactic language extensions in libraries and to activate them using imports. This is similar to metamodels in JProMo. However, each SugarJ syntax extension must define a unique *desugaring transformation* that is immediately used to desugar client code into Java before continuing compilation. In contrast to most other macro systems, SugarJ does not support communication integrity because it allows

unrestricted program transformations for macro expansion. JProMo is a major extension of SugarJ both conceptually and technically. In contrast to SugarJ, JProMo guarantees communication integrity, provides models and transformations as first-class concepts, models can exist independent of any transformation, arbitrary modules can be reified as models, transformations are explicitly applied in import statements, and transformations receive a callback to the JProMo compiler for compiling generated code.

There have been some works in the domain of classical embedded DSLs to achieve DSL programs (which correspond to models) that are independent of a particular interpretation, such as polymorphic embedding [HORM08], or the finally-tagless approach [CKS09]. A significant difference to these works is that they offer a better integration into the host language type system but no domain-specific syntax. They also limit the range of possible interpretations to compositional specifications in the underlying programming language. For instance, it would not be possible to generate datatypes, documentation, or a database schema using these approaches.

In comparison with existing MDD frameworks, model-oriented programming as realized in JProMo is, to the best of our knowledge, the first system to fully bridge modeling and programming: JProMo is the only MDD framework that organizes models, metamodels, and model transformations as modules of a well-behaved programming language. In particular, model-oriented programming ensures communication integrity, that is, all dependencies of models, metamodels, and especially model transformations are explicit in a code/modeling fragment. As support for this statement, we compare model-oriented programming to Xtext from the Eclipse Modeling Project, the Meta Programming System (MPS), and the model-oriented technology Umple as representatives.

Xtext [EV06, Xte12] is a MDD framework that supports textual metamodel-specific notation and a template-based transformation language. The application of transformations is specified in an application-specific build script called workflow. A workflow is a global, sequential description of which metamodel to use for parsing a model, which transformation to apply to which model, and how models are connected to (possibly generated) artifacts. As consequence, Xtext is not modular: Dependencies are not explicit in a module and the framework does not support separate compilation of models. Furthermore, Xtext does not provide a uniformly applicable modeling mechanism: It cannot be used to provide an alternative transformation or metamodeling language.

The Meta Programming System (MPS) [Völ11] is a MDD framework that avoids parsing and uses projectional editing to modify models directly. In MPS, dependencies between modeling artifacts are specified within a property dialog for each artifact separately. However, these dependencies are not part of the textual projection of an artifact. Furthermore, the application of a model transformation is not specified as part of the client code, but within the original model itself. Accordingly, when a new client requires a different transformation, this has to be specified in the property dialog of the original model. Finally, MPS does not automatically deduct the set and order of models for recompilation. In contrast, after changing any model, JProMo leverages

communication integrity to determine the set of dependent models that require separate recompilation.

Umple [FBLS12] is a programming language for model-oriented programming that integrates modeling constructs such as associations or statemachines into the Java programming language. In contrast to JProMo, Umple only supports a fixed set of modeling constructs with a fixed semantics and is not extensible by the user. Thus, Umple does not support custom domain abstractions. Moreover, Umple retains a stratification of artifacts into metalevels, because generated code cannot be integrated into the user's source program.

There is a wide variety of other MDD frameworks which are too numerous to discuss here; to the best of our knowledge, none of them supports model-oriented programming as described in this paper. In particular, they do not provide a uniform modeling methodology and do not ensure communication integrity.

A number of authors have envisioned MDD frameworks that inspired us in the development of model-oriented programming. First, Kent calls for families of domain-specific languages that come with tool support [Ken02]. He specifically argues for uniform metamodeling support to enable the generation of tool support and semantics. Second, in analogy with the everything-is-an-object idea from object-oriented programming, Bézivin proposes the unifying view that everything is a model, including metamodels and model transformation [Béz05]. While Bézivin is more rigorous in his vision than we are—for example, he proposes to regard the trace of running a program as a model as well—model-oriented programming realizes the everything-is-a-model idea to a large degree. Finally, France and Rumpe challenge the research community with respect to model-driven development of complex software [FR07]. They argue that future MDD frameworks should support domain abstraction, formal semantics, extensibility, separation of concerns, and model maintenance. Model-oriented programming addresses separation of concerns and maintainability with communication integrity, and extensibility through our uniform modeling methodology.

6.9 Chapter summary

We have presented model-oriented programming, a software-development approach that extends SugarJ with (i) polymorphic domain abstraction and (ii) communication integrity. On the one hand, polymorphic domain abstraction allows a single domain-specific program to have different semantics in different contexts. This increases the flexibility of SugarJ. On the other hand, communication integrity is an important step toward referential transparency and prevents transformations from injecting dependencies into generated code. This represents a new principle in SugarJ.

However, model-oriented programming not only forwards SugarJ, but also consolidates MDD. In particular, model-oriented programming represents an MDD programming

language that uses libraries as a unifying architectural device. All dependencies between software artifacts are declared by library imports. Most importantly, model-oriented programming abolishes the need for global build scripts, enables separate compilation, and guarantees communication integrity. Furthermore, model-oriented programming does not stratify modeling artifacts into stages or levels; in model-oriented programming, metamodels and model transformations are models and languages at every metalevel are customizable. Our case studies suggest that model-oriented programming can provide modular and effective solutions to a wide range of software-development scenarios.

7 Composability of Domain Abstractions

This chapter shares material with the LDTA'12 paper "Language Composition Untangled" [EGR12].

One of the most important principles supported by SugarJ and its variants is composability of domain abstraction. Composability enables programmers to use concepts from different domain-specific languages (DSLs) within a single program, for example, to query a database with SQL statements and present the result as synthesized XML code. Support for composing DSLs is especially important for SugarJ and its variants, because they promote small, reusable language extensions as sugar libraries. Therefore, from a user's perspective, composing DSLs is as simple as importing all corresponding sugar libraries.

More generally, in language-oriented programming and modeling, software developers are largely concerned with the definition of DSLs and their composition. While various implementation techniques and frameworks exist for defining DSLs, language composition has not obtained enough attention and is not well-enough understood. In particular, there is a lack of precise terminology for describing observations about language composition in theory and in existing language-development systems. To clarify the issue, we specify five forms of language composition: language extension, language restriction, language unification, self-extension, and extension composition. We illustrate this classification by various examples and apply it to discuss the performance of different language-development systems with respect to language composition. We hope that the terminology provided by our classification will enable more precise communication on language composition.

7.1 Introduction

DSLs are a prominent candidate for bridging the gap between domain concepts and software developers. DSLs enable software developers to think about the components and relations of a domain rather than about how these components and relations might be represented. DSLs thus provide abstraction over the concrete realization of domain concepts.

Not least due to the success of DSLs in practice, many language-development systems have been investigated [MHS05]. To implement a DSL, a language developer can, for example, write a parser and interpreter, apply an attribute grammar system [EH07a, VKBS07], use a language workbench [EV06, KV10], write a compiler plug-in for an extensible compiler [EH07a, NCM03], or provide a library for domain primitives using

regular functions [Hud98], macros [Tra08, THSAC+11], or sugar libraries. Advances in DSL implementation techniques have led to a proliferation of DSLs in today's software engineering research and practice, and DSLs for many problem domains are available.

However, realistic software projects are not just concerned with a single problem domain but also with many secondary domains such as data serialization and querying, communication, security, data visualization, graphical user interfaces, concurrency, or logging. Following the idea of language-oriented software development [Dmi04, Fow05b, War95], we want to provide a separate DSL for each domain that occurs in a project and to use all of these DSLs together. Support for this large and changing amount of domains can only be efficiently provided if DSLs can be implemented independently, and then composed together. Consequently, realistic software projects in a language-oriented context require *language composition*. Most recent work on language-development systems addresses language composition in one way or another.

At conceptual level, however, language composition is treated rather vaguely in the literature. In particular, there is no account the authors are aware of that specifies what language composition exactly means. This lack of a clear conceptual framework hinders our ability to reason about the composability of languages or to compare the support for language composition in different implementation techniques.

To this end, our goal is to provide precise terminology for language composition that enables effective communication on language composition and can serve as a basis for comparing existing and future language-development systems. In this chapter, we make the following contributions.

- We present a *classification of language composition* that distinguishes five cases: language extension, language restriction, language unification, self-extension, and extension composition. We illustrate this classification through various examples.
- We demonstrate that our classification provides precise terminology to explain language-composition support in existing technology and therefore clarifies our understanding of these systems.
- We apply our terminology to show that many language-development systems employ multiple forms of language composition. Without precise terminology, these different applications of language composition can easily be confused.
- Our classification reveals unexpected room for improvement for language-composition support in existing language-development systems. In fact, only one of the systems we investigated supports the composition of independent languages.

In this chapter, we focus entirely on language composition and try to clarify its meaning. We discuss related work of SugarJ with respect to our other design goals in the subsequent Chapter 8.

7.2 Language composition

The term "language composition" can refer to mechanisms and usage scenarios that significantly differ in terms of flexibility and reuse opportunities. In fact, the composability of languages is not a property of languages themselves: any two languages can be composed by stipulating a new syntax and semantics for the composed language. Rather, language composability is a property of language definitions, that is, whether two definitions work together without changing them.

To clarify the situation, we develop a taxonomy of language composition based on the idea of unchanged reuse, that is, whether a language definition can be reused without modifying it. Existing language-development systems differ significantly in their support for unchanged reuse. For example, some systems support the unchanged reuse of a base language through extension (e.g., macro systems), whereas other systems even allow to compose independently developed languages unchanged (e.g., JastAddJ). To avoid ambiguous statements, authors need to be aware of the equivocality of language composition and we recommend to consciously use language composition only as an umbrella term for our more precise terminology.

7.2.1 Language extension ($\triangleleft$)

When the first stable version of Java was released, it lacked many features that we are used to today. For example, before version 1.5, Java had no support for the *foreach* loop or generics. Java was only extended with these features later on. Similarly, earlier versions of Haskell did not include support for `let` expressions (introduced in Haskell 1.1), monads, or `do` notation (both introduced in Haskell 1.3) [HHPW07]. By now, these later-added features have become characteristic for Java and Haskell, respectively. More generally, languages evolve over time and subsequent introduction of language features is nothing surprising.

This brings us to the first form of language composition: *language extension*. A language designer composes a base language with a language extension. A language extension is itself a language fragment, which often is meaningless when regarded independent of the base language. This dependency of the language extension on the base language is the main characteristic of this form of language composition.

Often, implementing a language extension involves changing the implementation of the base language. Examples include the integration of generics into Java and `do` notation into Haskell. However, the language-engineering community has brought forward language-development systems that particularly support language extensibility. These systems share a common property, which we capture in the following definition.

Definition 1. *A language-development system supports language extension of a base language if the implementation of the base language can be reused unchanged in implement of the extended language.*

Importantly, this definition only demands the reuse of the base language's implementation but does not regulate how language extensions are implemented. In particular, this definition does not prescribe whether multiple language extensions can be used jointly. In addition to describing terminology, we also introduce an algebraic notation for language composition. We will later use this notation to explain how different forms of language composition integrate. We denote the result of composing a base language B with a language extension E as $B \lhd E$. The asymmetry of the language-composition operator $\lhd$ reflects the dependency of the extension on the base language.

Language restriction. Especially in education, it sometimes makes sense to restrict an existing programming language. For example, to teach students functional programming in Haskell, monads and type classes are rather hindering. It might be more instructive to rigorously forbid the use these constructs. We call this language restriction as opposed to language extension.

Interestingly, language restriction does not require special support by language-development systems. Instead, a language restriction can be implemented as an extension of the static analyses of the base language: The extension rejects any program that uses restricted language constructs. The same idea is used in pluggable type systems [Bra04]. Since language extension subsumes language restriction, we do not treat language restriction specifically in the remainder of this chapter.

7.2.2 Language unification ($\uplus$)

Language extension and language restriction assume the existence of one dominant (typically general-purpose) language that serves as the base language for other languages. However, sometimes it is more natural to compose languages on equal terms. For example, consider the composition of HTML and JavaScript. Both languages serve a purpose and can be used independently: HTML for describing web pages and JavaScript as a prototype-based object-oriented programming language. If anything, it would make sense to use the general-purpose language JavaScript as a base language for the generation of dynamic HTML content. However, in the domain of dynamic web pages, the HTML-based view appears to be the central program artifact.

Accordingly, we want to compose languages in an unbiased manner. Furthermore, the language composition should be deep and bidirectional, that is, program fragments from either language should be able to interact with program fragments from the other language. For example, in the composition of HTML and JavaScript as defined by the W3C [W3C99], JavaScript programs can manipulate and generate HTML documents using the DOM tree or the function `document.write()`, and dynamic JavaScript-based behavior can be attached to HTML elements using attributes like `onMouseOver="showPopup()"`. Thus, to compose HTML and JavaScript, we need change both languages: We add support to

JavaScript for generating and inspecting HTML document trees and we supplement the definition of HTML elements to allow event attributes.

This illustrates the next form of language composition: *language unification*. A language designer composes two independent languages by unification. Like in mathematical unification, language unification requires that parts of the languages are equalized. For example, deep integration often requires sharing of primitive data types such as numbers or strings. Also, like in mathematical unification, the unified language subsumes its two constituents.

Language unification is very difficult to achieve in practice and rarely supported by language development systems. Often language unification requires the composition of language implementations by hand. The reason for this seemingly incompatibility of languages is the lack of a common back-end, for example, in languages that are compiled for different VMs or implemented by different interpreter engines. Unification is simpler if the same language-development system implements both languages. In particular, for languages that do not integrate bidirectionally, support for language extension suffices to unify both languages, such as Java and regular expressions, where the latter does not support references to Java artifacts. More generally, though, we apply the following definition.

Definition 2. *A language-development system supports language unification of two languages if the implementation of both languages can be reused unchanged by adding glue code only.*

Notably, this definition permits the adaption of the unified languages as long as their implementations remain unchanged. Generally, we can assume that some program weaves the two language implementations together. As usual in component engineering and modularity discussions, we refer to the program that weaves two languages as glue code.

We write $L_1 \uplus_g L_2$ to denote the language that unifies L_1 and L_2 with glue code g. The symmetry of the language operator $\uplus$ reflects that unification composes languages on equal terms. Due to glue code, though, $\uplus$ is not necessarily a symmetric relation, that is, $L_1 \uplus_g L_2$ only equals $L_2 \uplus_g L_1$ for different glue code g. Moreover, the unification of two languages is typically not unique. For example, in HTML $\uplus_g$ JavaScript, the glue code g determines the attribute name `onMouseOver`, which might as well be called `onPointerOver` by different glue code.

7.2.3 Self-extension ($\hookleftarrow$)

For many subdomains of a software project, there are special-purpose languages that provide functionality specific to the subdomain. Examples of such DSLs include SQL for data querying, XML for data serialization, and regular expressions for string analysis. Since these languages each only tackle a small part of a software system, it makes sense

to make their functionality available in a general-purpose language that can serve as a bridge between these DSLs.

Traditionally, this form of language composition is called language embedding: A domain-specific language is (purely) embedded into a host language by providing a host-language program that encapsulates the domain-specific concepts and functionality [Hud98]. However, the term "language embedding" is ambiguous since it only characterizes the result of integrating one language into another language. Pure embedding is not the only technique for achieving such integration. For example, a compiler plugin can describe the embedding of one language into a base language, too. Since the decisive difference to other forms of language composition is *how* we integrate languages, our terminology should reflect that. In particular, we aim to exclude systems where the extensibility is external to the host language.

We call this form of language composition *self-extension*. To compose a host language with an embedded language, a language implementer develops—in the host language—a program which defines the embedded language. Often the definition of the embedded language simply consists of a host-language API for accessing domain-specific concepts and functionality. More advanced languages also enable the self-extension of the host language's syntax, static analyses, or IDE support. Because the implementation of an embedded language is itself a regular program of the host language, the host language extends itself.

There are various ways of self-extending a language, but two extension styles are most popular: string embedding and pure embedding. In string embedding, a program of the embedded language is represented as a string of the host language and the embedded language provides an API for evaluating embedded programs. A good example of string embedding is the integration of regular expressions into Java (similar for many other host languages). A programmer writes a regular expression `"a[b-z]*"` as a string and passes it to the library function `Pattern.match` as in `Pattern.match("a[b-z]*","atext")`. `Pattern.match` parses and compiles the regular expression at run time and matches it against the given input text `"atext"`. Another example for string embedding is the integration of SQL into Java, where SQL queries are represented as Java strings (see package `java.sql`). Generally, string-embedded programs do not compose well with each other because string embedding reifies a lexical macro system [EO10]. Moreover, string embeddings are vulnerable to injection attacks [BDV10].

Alternatively, programs of the embedded language can also be expressed as a sequence of API calls in the host language. Paul Hudak coined the term pure embedding for this kind of self-extension [Hud98]. As an example, consider the embedding of XML into Java using JDOM. A program of the embedded language XML is simply a Java program that utilizes the JDOM API:

```
Element book = new Element("book");
book.setAttribute("title", "Sweetness and Power");
```

```
Element author = new Element("author");
author.setAttribute("name", "Sidney W. Mintz");
book.addContent(author);
```

A purely embedded language does not provide its own syntax but instead reuses the syntax of the host language. Therefore, programs of a purely embedded language can be readily mixed with code from the host language, for example, to retrieve the author name from a database.

The term self-extension can only apply to languages and not to language-development systems in general. Accordingly, we define:

Definition 3. *A language supports self-extension if the language can be extended by programs of the language itself while reusing the language's implementation unchanged.*

Self-extension has three essential advantages over regular language extension. First, to run or compile a program of a self-extended host language, the standard interpreter or compiler of the host language is reused. In contrast, systems that support regular language extensions often require compiler configurations that reflect the activated extensions, which may differ for different source files. Second, since the extended language is part of the host language, programmers can reuse standard libraries of the host language in code that applies a language extension. Third, since self-extensions are implemented in the self-extensible language itself, extensions can be used when writing further self-extensions. In particular, this enables the integration of meta-DSLs, that is, DSLs for implementing further DSLs (see Chapter 2).

We write $H \hookleftarrow E$ to denote the self-extension of a host language H with the embedded language E. As defined above, the implementation of E has to be an instance of H. The asymmetry of the language operator $\hookleftarrow$ reflects this dependency of the embedded language on the host language.

7.2.4 Extension composition

So far, we have identified three language-composition scenarios a language or language-development system may support: language extension, language unification, and self-extension. However, these properties only describe to which extent a system supports base-language composition with a single extension or language. Our terminology so far does not describe to which extent a system supports the composition of extensions, that is, whether different extensions can work together.

Let us first note that systems which support language unification also support unification of extensions: $L \uplus_g (E_1 \uplus_h E_2)$. On the other hand, for systems that only support language extension, we need to distinguish three cases: no support for extension composition, support for incremental extension, and support for extension unification. In a system that does not support any form of extension composition, two extensions

$B \lhd E_1$ and $B \lhd E_2$ cannot be used in combination at all. For example, this occurs in preprocessor-based systems. In contrast, in a system that supports incremental extension, an extended language $B \lhd E_1$ can in turn be extended to $(B \lhd E_1) \lhd E_2$. Here, extension E_2 may be specifically built to work on top of E_1. Incremental extension supports Steele's idea of growing a language [Ste99]. Finally, in a system that supports extension unification, two independent extensions can be composed and used together $B \lhd (E_1 \uplus_g E_2)$ by using some glue code g. Extension unification supports growing a language modularly.

A particularly interesting instance of extension unification is modularly defined language extensions that entirely avoid glue code $B \lhd (E_1 \uplus_\emptyset E_2)$ [KV12, SV09]. Such language definitions are restricted in expressiveness to guarantee their composability. This constitutes an interesting trade-off between the flexibility and the composability of language extensions.

Self-extension adheres to the same case distinction for extension composability as language extension: no extension composability, incremental extension, or extension unification. In addition, though, self-extensible languages support another interesting form of extension composition, namely self-application. Since implementations of extensions are programs of the host language itself, a host-language extension E_1 can be used in the implementation of another extension E_2, that is, $H \leftarrowtail E_2$ where E_2 is an instance of $H \leftarrowtail E_1$.

This discussion shows that language composition is not only important for the base language but also for extensions. Therefore, precise terminology is crucial to enable clear statements about the language-composition support of a system and to prevent confusion about whether a statement addresses base-language composability or extension composability. Furthermore, this discussion illustrates the utility of an algebraic notation for describing and reasoning about language composition.

7.3 Language components

Support for language composition is often not uniform for all components of a language definition because different low-level techniques and high-level considerations apply to different aspects of a language. Generally, a language consists of syntax and semantics. Accordingly, most language definitions stipulate the syntax and semantics of a language separately. However, for machine-processed languages and programming languages in particular, this picture is not entirely correct. In fact, the definition of many machine-processed languages consists of three artifacts: a context-free syntax, a collection of non-context-free validation procedures (the static semantics), and a definition of the language's behavior (the dynamic semantics). While the reason for separating context-free syntax and validation is a technical one—generic context-sensitive parser frameworks are inefficient—we cannot ignore the implications on language design and language

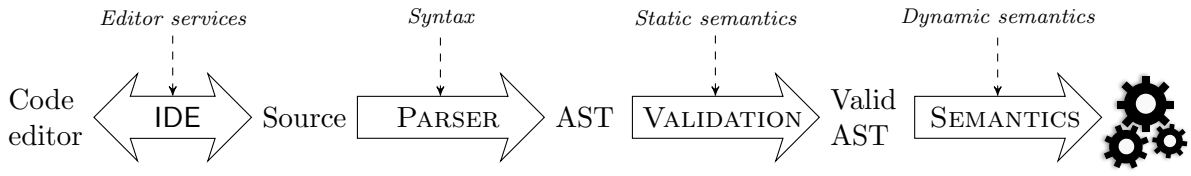


Figure 7.1: A typical language processing pipeline.

composition.

The relation between language-definition artifacts is depicted in Figure 7.1. First, a parser checks whether the input source code adheres to the given context-free grammar and either rejects the program with an error message or produces an abstract syntax tree. Subsequently, the language-validation procedure processes the resulting syntax tree and either accepts or rejects it. If the code is not valid, validation generates an error report. If the program is valid instead, validation may add information to the AST (for instance, resolving overloading in Java). Next, the language's (dynamic) semantics takes a syntax tree as input and produces the meaning of the corresponding program. The behavior of the dynamic semantics may be unspecified for programs which are rejected during parsing or validation.

In addition to these classical components of a language processing pipeline, we include integrated development environments (IDEs) as a fourth component into Figure 7.1 and the discussion in the present chapter. IDEs provide an editor with various editor services to the programmer. Editor services may include syntax coloring, code outline, code folding, code completion, reference resolving to jump to the definition of an identifier, or refactorings. More generally, this component includes all programming tools that a developer can use to write, navigate, and maintain programs. While IDE support is not directly part of a language definition, it is essential for the productivity of programmers. Furthermore, only few systems exist that support the composition of IDE support for different languages.

Our separation of languages into four components is general and covers virtually every programming language. For instance, the Java programming language declares a context-free syntax, a type checker, and a compiler that produces byte code [GJSB05]. Instead of using a general context-sensitive parser to parse Java's context-sensitive syntax directly, compilers parse the context-free syntax first before applying special-purpose validations such as type checking and the remainder of compilation. In addition, various IDEs for Java exist, for example, Eclipse or IntelliJ IDEA. Another example language is XML: XML's context-free syntax and XML validity can both be checked efficiently, whereas the application of a general-purpose context-sensitive parser will likely lead to inefficient XML processing. Finally, note that language components as outlined above similarly exist for DSLs such as SQL, VHDL, or DOT.

However, some languages combine two or more of the language components we identified. Prominently, dynamically typed languages such as Ruby or Smalltalk perform well-typedness validation as part of their dynamic semantics. Alternatively, type checking and parsing can be combined to resolve syntactic ambiguities by typing information [BVVV05]. LaTeX even applies parsing and validation as part of its dynamic semantics: it repeatedly parses, validates and executes the next command or macro until the complete source file is processed [EO10]. Finally, in Smalltalk, even the IDE is interpreted by the language's dynamic semantics and can be modified at run time [RGN10].

7.4 Existing technologies

We introduced new terminology for language composition in order to enable more precise descriptions of existing and future technologies. In this section, we exemplify the use of our terminology to classify existing language-development systems with respect to their support for language composition.

We reviewed existing language-development systems as described in the literature in light of our classification. Table 7.1 summarizes our findings. Each cell in the table shows how a system supports composition with respect to a specific language component, both regarding language extension or unification (first symbol) and regarding extension composition: incremental extension or extension unification (second symbol, in parentheses). The last column applies to all language components and records whether a system supports self-extension. We have been somewhat liberal in our judgment for extension unification and also acknowledged support to systems that only support unification for non-interacting language extensions.

Different technologies follow very different approaches to achieve language composability. One of the simplest and also most popular mechanisms is hand-written preprocessors [Spi01]. To extend a language, a programmer writes a preprocessor that translates the extended language into the base language. However, each extension requires its own preprocessor and preprocessors can only be composed sequentially, that is, run one after another. Consequently, preprocessors only support incremental extension but not extension unification.

AspectLisa [RMHP06], ableJ [VKBS07], and JastAddJ[EH07a] follow more sophisticated approaches and build on attribute grammars. Attribute grammars [EH04, VBGK10] enable the definition of new productions to extend the base syntax and new attributes to extend the base language validation and semantics. Since AspectLisa and ableJ allow language extensions to reuse and extend base-language attributes, they support language extension, where the base language does not have to be changed. In addition, AspectLisa applies aspect-oriented programming to add new attributes to productions of the base language. On the other hand, JastAddJ applies aspect-oriented programming and rejects information hiding to support overwriting attributes. Accordingly, JastAddJ supports

	Syntax	Validation	Semantics	IDE	Self-ext.
OpenJava [TCKI00]		$\lhd(\ )$	$\lhd(\uplus)$		yes
pure embedding [Hud98]		$\lhd(\uplus)$	$\lhd(\uplus)$		yes
MPS [VS10]		$\lhd(\uplus)$	$\lhd(\uplus)$	$\lhd(\uplus)$	yes
string embedding	$\lhd(\ )$		$\lhd(\ )$		yes
AspectLisa [RMHP06]	$\lhd(\ )$	$\lhd(\ )$	$\lhd(\uplus)$		no
Converge [Tra08]	$\lhd(\ )$	$\lhd(\ )$	$\lhd(\ )$		yes
preprocessors [Spi01]	$\lhd(\lhd)$	$\lhd(\lhd)$	$\lhd(\lhd)$		no
Racket [THSAC$^+$11]	$\lhd(\lhd)$	$\lhd(\uplus)$	$\lhd(\uplus)$		yes
JSE [BP01]	$\lhd(\uplus)$	$\lhd(\ )$	$\lhd(\uplus)$		yes
Helvetia [RGN10]	$\lhd(\uplus)$		$\lhd(\uplus)$	$\lhd(\uplus)$	yes
ableJ [VKBS07]	$\lhd(\uplus)$	$\lhd(\uplus)$	$\lhd(\uplus)$		no
Polyglot [NCM03]	$\lhd(\uplus)$	$\lhd(\uplus)$	$\lhd(\uplus)$		no
JastAddJ [EH07a]	$\lhd(\uplus)$	$\uplus(\uplus)$	$\uplus(\uplus)$	$\uplus(\uplus)$	no
Spoofax [KV10]	$\uplus(\uplus)$	$\lhd(\uplus)$	$\lhd(\uplus)$	$\lhd(\uplus)$	no
SugarJ and variants	$\uplus(\uplus)$	$\lhd(\uplus)$	$\lhd(\uplus)$	$\lhd(\uplus)$	yes

Table 7.1: Support for language composition in existing language-development systems: No composition (empty), extension but no extension composition $\lhd(\ )$, incremental extension $\lhd(\lhd)$, extension unification $\lhd(\uplus)$, language unification $\uplus(\uplus)$.

the composition of languages by unifying their respective implementations, that is, by only adding glue code and not changing previous implementations. The same applies to IDE support [SH11].

Polyglot [NCM03] is an extensible compiler that allows language extensions to integrate into various compiler phases. For example, a language extension can extend the parsing, type checking, and code generation phase of the compiler to support additional language constructs. Polyglot achieves language extensibility with method delegation, where compiler actions are delegated to extensions, which further delegate to yet other extensions. Polyglot does not support language unification since adapting the behavior of extensions is not supported.

Spoofax [KV10] follows an alternative approach to language composition based on SDF for syntax composition and Stratego for semantic composition. SDF [Vis97b] applies scannerless generalized LR parsing, which enables the unification of arbitrary context-free grammars. However, generalized parsing may result in a syntax tree that contains ambiguities. SDF supports the elimination of ambiguities on the basis of glue code, that is, without changing the original grammars. For semantic composition, Spoofax applies the Stratego term rewriting language [VBT98], which supports adding rules to handle an extended base language. Stratego does not support the adaption of an existing rule base,

though, which is necessary to unify languages.

Self-extensible languages. The following language-development systems are self-extensible languages, that is, the base language itself is used to implement language extensions or glue code. The extended base language can then be used in the implementation of further self-extensions. Notwithstanding this similarity, self-extensible languages come in various flavors.

String embedding and pure embedding are approaches available in any base language that supports strings and procedural abstraction, respectively. In string embedding, programmers use language extensions by writing specially-formatted strings of the base language, which the extension parses and evaluates at run time of the program. A typical example of a string-embedded language is the language of regular expressions. The main problem of string embedding is the lack of proper structural abstraction. Therefore, string embeddings fall back to lexical abstraction and lexical composition of program snippets, which is error-prone and forestalls static syntax analyses [EO10]. Furthermore, since IDEs require a structural representation of programs, string embedding comes without IDE support. Nevertheless, string embedding is widely applied in practice, for example, to issue SQL queries or generate XML documents [Feh11].

Pure embedding takes a more structural approach than string embedding and represents programs as API calls [Hud98]. In particular, a programmer can nest or sequentialize calls to such a special-purpose API. Moreover, API calls can readily be mixed with regular base language code as well as with calls to other special-purpose APIs. There is, however, one constraint that is often overlooked: Pure embeddings must share their data representations. For example, suppose an extension provides its own collection data type. This prevents reuse of functionality from the base language such as mapping or sorting as well as integration with other extensions that can only process standard collections. As pointed out by Mernik et al. [MHS05], pure embedding enables the reuse of IDE support of the base languages such as code completion for a special-purpose API. However, true domain-specific editor services such as SQL-specific code coloring is not in the focus of pure embedding.

Converge [Tra08], JSE [BP01], OpenJava [TCKI00], and Racket [Fla12, THSAC$^+$11] enable language extensions with macros and macro-like facilities. A macro is much like a normal function except it is run at compile time. Consequently, a macro does not receive or produce normal run-time data, but instead takes and produces compile-time data, that is, representations of programs. Converge, JSE, and Racket represent programs as syntax trees, whereas OpenJava represents programs as metaobjects. None of these systems support language unification since the meaning of a previously defined macro cannot be changed. However, some macro systems come with more advanced support for unifying independent language extensions. For example, Racket supports extension unification through local and partial macro expansion, which enables the collaboration of

independent macros [FCDF12].

SugarJ (Chapter 2) is similar to macro systems but supports more flexible syntax composition. Like Spoofax, SugarJ employs SDF [Vis97b] to support the unification of arbitrary context-free grammars, where additional glue code can coordinate between grammars to eliminate ambiguities. To specify the validation and semantics of extensions, SugarJ uses Stratego's support for composing partial pattern matches through equally-named rules. Since pattern matches can only be added, SugarJ does not support the unification of an extension's validation or semantics. Moreover, SugarJ provides IDE support for the base language and extensions (Chapter 3). IDE support is extensible because it aggregates information from all extensions (e.g., for code completion) or chooses the most specific editor service available (e.g., for syntax coloring), but unification of editor services is not supported.

Helvetia [RGN10] leverages Smalltalk's dynamic nature to enable extensibility of parsing, compilation, and IDE support. Helvetia extensions are implemented through annotated methods, which Helvetia organizes in a global rule set. Whenever two or more rules are active in the parser, compiler, or IDE, Helvetia throws an error. It is not possible to adapt existing extensions non-invasively.

The projectional language workbench MPS [VS10] rejects parsing and applies intentional programming instead. Essentially, MPS maintains a central program representation, which can be thought of as an AST, and displays projections of the AST to the programmer. To edit a program, a programmer sends edit directives to MPS, which applies the edits to the central AST and updates the projection. This way MPS provides IDE support and creates a user experience close to usual programming environments. Furthermore, MPS supports extensibility: The central program representation can be extended by new concepts, which can integrate into existing projections, validations, and code generation. As in the other systems, once defined, the behavior of an extension is fixed [Völ11].

Summary. We have shown how our terminology for language composition is useful to explain existing systems and distinguish between them meaningfully. In particular, our terminology enables the precise description of composition with the base language in contrast to composition of language extensions.

We are aware that our discussion of existing technologies is incomplete and many more systems deserve attention. In particular, we excluded any tools from this discussion that do not support semantic extensibility, because without semantics programs of an extended language cannot be executed. However, since the goal of this work is the clarification of language composition in general, we believe the omission of any particular system is negligible. Furthermore, we excluded semantic IDE services like debugging or testing from the present discussion. An investigation of the composability of such services remains future work.

One important conclusion of our study is the lack of wide-spread support for language

unification in existing systems. In our study, JastAddJ is the only tool that supports language unification for semantics. Language unification requires that a system supports the adaption of independently implemented languages, for example, by glue code. In JastAddJ, the flexible adaption by glue code is based on aspect-oriented programming. This suggests that technologies that favor flexibility over modularity in the sense of information hiding [OGKR11] should be more thoroughly investigated as a foundation for language-development systems.

7.5 Related studies

Other authors have described DSL-related patterns but with less focus on reusability of language implementations. Spinellis [Spi01] describes and classifies patterns for DSL design and implementation. Mernik et al. extend Spinellis' work and present an extensive survey [MHS05] that covers various aspects of DSL development methodologies: They identify different DSL development phases, discuss when DSL development is appropriate, and compare different implementation techniques for DSLs. Mernik et al. also survey language-development systems and mention the use of DSLs as metalanguages within such systems. Spinellis and Mernik et al. distinguish whether an existing language is restricted or extended with new elements. As explained in Section 7.2.1, we instead identify these scenarios and consider language restriction as an instance of language extension, targeting the validation system. In addition, we distinguish language unification, self-extension, and extension composability.

Hofer et al. [HORM08] distinguish hierarchical and peer language composition in the context of embedded DSLs. We can describe hierarchical language composition through $(H \lhd L_1) \lhd L_2$ and peer language composition through $H \lhd (L_1 \uplus_g L_2)$. Our notation and terminology thus covers these scenarios while supporting the description of further language-composition scenarios in a uniform way.

7.6 Chapter summary

The goal of this chapter is two-fold. First, we want to raise awareness on the many meanings of language composition and on the consequent ambiguity in discussions on language composition. For this ambiguity, we believe the lack of precise terminology deserves major blame. Therefore, our second goal is the classification of language composition and the introduction of precise terminology to describe language composition. We hope that the terminology introduced in this chapter can clarify future discussions and communication on language composition. It certainly helped us to better understand the composability of sugar libraries.

An interesting next step would be the development of a formal theory of language composition. In this chapter, we defined language-composition operators informally using

the notion of unchanged reuse. Furthermore, we refrained from specifying algebraic properties for language-composition operators. It would be interesting to study language composition based on a formal representation of languages, such as denotation semantics, modular structural operation semantics [Mos04], or algebraic specifications.

8 A Comparison of Approaches to Domain Abstraction

Domain abstraction has been the focus of research for quite some time now. Still, the interest in techniques and systems that support domain abstraction shows no sign of decline. In fact, domain abstraction has gained industrial relevance in the form of DSLs and model-driven development. Correspondingly, many approaches for realizing domain abstraction exist.

In the previous chapter, we *developed* a notion of language composition and used it to classify the language-composition support of various systems for domain abstraction. In contrast, in this chapter, we *survey* existing approaches with respect to *all* the design goals for flexible and principled domain abstraction we introduced in Chapter 1. Due to the vast number of existing approaches, we focus our survey on systems that are currently in use or have been developed in recent years. Table 8.1 gives an overview on the surveyed systems.

8.1 SugarJ

The central idea of SugarJ is to organize language extensions as libraries of the host language. We followed this path to support flexible domain abstraction with domain-specific syntax, semantics, analyses, and editor services as libraries. To achieve polymorphic domain abstraction, we extended SugarJ to JProMo, which decouples the definitions of syntax and semantics. Again, we use libraries to organize these artifacts and allow users to apply a transformation to a model as part of an import statement.

Regarding principled domain abstraction, our library focus has important advantages, such as the avoidance of global build scripts. In fact, SugarJ programs declare all their dependencies as library imports, which enables modular reasoning. For referential transparency, we assure communication integrity of transformations but fail to provide a fully hygienic transformation system. Thus, SugarJ transformations can perform accidental name capture. To counteract this danger, we typically generate references as fully qualified names that are not subject to name capture. However, this convention only works for top-level entities, because local variables are not qualified in either Java or Haskell.

SugarJ language extensions can reuse grammar productions and transformation rules from other libraries. However, our module system for grammars and transformations does not support fine-grained reuse: When imported, all productions and transformation rules

Approach	domain-specific syntax	domain-specific semantics	domain-specific analysis	domain-specific editor	polymorphic DSLs	modular reasoning	referential transparency	implementation reuse	declarativity of impl.	DSL composition	uniformity
SugarJ	●	●	●	●	●	●	◒	◒	●	●	●
Embedding											
string embedding	◒	●	○	○	●	●	●	○	○	○	●
pure embedding [Hud98]	○	●	◒	○	○	●	●	●	○	●	◒
poly. embedding [HORM08]	○	●	◒	○	●	●	●	●	○	●	◒
Internal extensibility											
Extensible syntax with lexical scope [CMA94]	●	◒	○	○	○	●	●	○	●	◒	○
OpenJava [TCKI00]	○	●	○	○	○	●	○	●	○	○	●
Helvetia [RGN10]	●	●	●	●	○	●	○	●	●	●	●
Katahdin [Sea07]	●	●	○	○	○	●	○	●	◒	●	○
Fortress [ACN+09]	●	●	○	○	○	●	●	○	●	●	○
Converge [Tra08]	●	●	●	○	○	●	●	●	●	○	●
Template Haskell [SP02]	●	●	○	○	○	●	◒	●	◒	○	●
Racket [Fla12]	◒	●	●	○	○	●	●	●	●	●	●
Honu [RF12]	◒	●	○	○	○	●	●	●	●	●	●
External extensibility											
Camlp4 [dR03]	●	●	○	○	○	○	○	●	●	○	●
Nemerle [SMO04]	◒	●	○	○	○	○	●	●	●	●	●
JSE [BP01]	◒	●	○	○	○	○	◒	●	●	●	●
Metaborg [BV04]	●	●	○	○	○	○	○	●	●	●	●
JastAdd [EH07b]	●	●	●	●	○	○	○	●	●	●	○
Silver [VBGK10]	●	●	●	○	○	○	○	●	●	●	●
Polyglot [NCM03]	●	●	●	○	○	○	○	●	○	○	●
Language workbenches											
Meta-Environment [vdBvDH+01]	●	●	●	◒	○	○	○	●	●	●	○
Spoofax [KV10]	●	●	●	●	○	○	○	○	●	●	○
MPS [Völ11]	◒	●	●	●	○	◒	○	●	●	●	○
Cedalion [LR11]	◒	●	●	●	○	○	○	●	●	●	●
Xtext [EV06]	●	●	●	●	●	○	○	●	●	○	○
Monticore [KRV10]	●	●	●	●	●	○	○	●	◒	◒	○
Epsilon [KRPGD12]	○	●	●	○	●	○	○	●	●	○	○

● addressed as goal ◒ addressed, but with restrictions ○ not regarded as goal

Table 8.1: Overview of systems that support domain abstraction.

of a library are brought into scope. In future work, we want to support more fine-grained reuse that allows to select and rename definitions for import.

SugarJ builds on SDF and Stratego for the implementation of language extensions. SDF and Stratego are declarative DSLs for parsing and transforming programs. One important consideration when selecting SDF and Stratego for SugarJ was their integrated support for composability. SDF grammars can be freely composed and glued together [Vis97b, vdBSVV02]. Stratego rules compose due to a try-catch execution model where multiple definitions of a rule may coexist, and all of them are tried until one succeeds or all fail [VBT98].

Finally, like regular libraries from nonextensible programming languages, SugarJ language extensions are self-applicable, which enables the application of domain abstraction in the implementation of further domain abstractions. Beyond regular libraries, SugarJ language extensions can also be used to abstract over the library concept altogether. We call a language extension that abstracts over the sugar-library concept a meta-DSL. In contrast, regular libraries cannot abstract over the library concept itself.

To structure our survey, we classify existing approaches into three categories: embedding, internal extensibility, and external extensibility. We discuss the systems of each category in turn.

8.2 Embedding

Embedding approaches reuse facilities of the host language to encode domain abstractions. We distinguish string embedding, pure embedding, and polymorphic embedding.

String embedding encodes domain-specific programs as strings of a host language. This requires the escaping of quotes and does not support static syntax checks, let alone more sophisticated semantic analyses. Instead, string-embedded programs are parsed at run time. Like for all other embedding techniques, the reuse of host-language facilities inhibits domain-specific editor services because domain-specific programs are indistinguishable from regular programs. String embedding supports polymorphic domain abstractions because domain-specific programs (host-language strings) are first-class and can be submitted to different semantics.

One advantage of embedding approaches is that no code generation is required. Therefore, embedding approaches retain the modular reasoning and referential transparency of the host language. However, string embeddings require a nondeclarative implementation that includes a run-time parser, analyzer, and interpreter. Since any host-language function can implement these artifacts, it is difficult to reuse or compose implementations of domain abstractions. String embedding supports uniformity since the interpretation of a string may result in another interpretation function, which can be applied at a lower metalevel.

Pure embedding represents domain abstraction through APIs of the host language

that encode the concepts of the domain [Hud98]. A domain-specific program then is a sequence of API calls. Accordingly, domain-specific programs have to follow the host-language syntax for function or method application, and cannot be written in a domain-specific syntax. The encoding as function application limits domains-specific analyses to the extent that can be encoded as type signatures of the host language's type system. A polymorphic interpretation of domain-specific programs is not possible in the pure-embedding approach.

Since all domain abstractions are implemented as library APIs, it is easy to reuse part of a domain abstraction in other domains. The implementation of a purely embedded domain abstraction is conducted using regular host-language constructs, which are not tailored to the implementation of domain abstractions. Pure embedding supports the composition of domain abstractions particularly well, because of the usage of libraries for scoping: A programmer can import multiple domain abstractions by importing multiple libraries and using the multiple APIs simultaneously. Pure embedding partially supports uniformity: While it is possible to use one library in the implementation of another library, it is not possible to use a library for declaring another API, because APIs typically are not first-class.

Polymorphic embedding extends the pure-embedding approach by separating a domain abstraction into a *language interface* and its implementations [HORM08]. In particular, a language interface can have multiple implementations, each of which represents a different semantics for the domain.

Domain-specific programs are parametric over the concrete semantics, that is, programs are written against the language interface. Before executing a program, a developer first has to select a concrete semantics and specialize the program to that semantics (through function application). This way, domain-specific programs are independent of concrete semantics. This makes it possible to start out with a simple interpreter semantics, and later change it to an optimized or pretty-printer semantics without modifying the program or the language interface.

Technically, polymorphic embedding uses Scala traits to represent language interfaces and their implementation. Since traits are not first-class in Scala, the uniformity of polymorphic embedding is restricted similar to pure embedding. Generally, polymorphic embedding retains all the advantages of pure embedding but adds polymorphic domain abstraction.

8.3 Internal extensibility

Systems that support internal extensibility provide metaprogramming facilities as an integral part. Typically, such systems are programming languages with metaprogramming facilities such as a macro system or a metaobject protocol. Since internal extensibility does not depend on external information, one of the characteristic features of internal

extensibility is modular reasoning: Developers can modularly reason about the set of available domain abstractions. All of the internally extensible systems we investigated support modular reasoning.

Cardelli et al. describe a language that features *extensible syntax with lexical scope* [CMA94]. The language supports flexible syntactic extensibility but is rather restricted in semantic expressiveness. Essentially, a syntactic extension can only paraphrase a host-language expression; no recursion or similar construct is supported in the declaration of domain semantics. This restriction enables Cardelli et al. to guarantee referential transparency because their transformation mechanism is hygienic. Cardelli et al. define a declarative, EBNF-like language for the declaration of grammars, where it is possible to express how an extension composes with the previous grammar. Moreover, they support the use of quasiquoted syntax for specifying the term that is generated on application of a production. Technically, Cardelli et al. generate an LL(1) parser from the syntactic extensions. Since LL(1) parsers support little lookahead and are not closed under composition, the composability of Cardelli et al.'s approach is limited.

OpenJava [TCKI00] applies a metaobject protocol [KDRB91] that enables reflection and modification of the structure of a Java class definition. An OpenJava class can declare a metaclass (also written in OpenJava), which is used to define compile-time transformations of the class. These transformations are regular OpenJava programs that exploit the metaobject protocol to inspect and modify the class structure. The instrumentation of an OpenJava class by a metaclass is explicit in the source code, which enables modular reasoning on the behavior of instrumented classes. Furthermore, metaclasses are regular OpenJava classes that are subject to reuse and instrumentation themselves. Therefore, OpenJava provides a uniform metaprogramming mechanism.

Helvetia applies the metaobject protocol of Smalltalk to provide a rich extensible language [RGN10]. Helvetia enables the programmer to influence the Smalltalk parser, code generator, and IDE, which allows for flexible integration of DSLs. Moreover, Helvetia supports dynamic domain-specific analyses [RDGN10]. Helvetia organizes DSL implementations in language boxes [RDN09]. Like libraries, language boxes encapsulate DSL implementations, but provide more fine-grained scoping than Java libraries: A Helvetia DSL can be scoped to a method, a class, a package, or the whole system. Moreover, users of DSL can activate a language box in the current scope.

Language boxes can be implemented using regular Smalltalk code, which enables uniformity and reuse. But Helvetia also provides a declarative parser-combinator DSL and supports quasiquoted syntax in transformations. The parser-combinator DSL features scannerless parsing [Vis97a] and ordered choice (similar to parsing expression grammars [For04]) to support the composition of different language boxes.

Katahdin is an interpreted programming language that supports extensible syntax through dynamic parser recompilation [Sea07]. The Katahdin interpreter detects the definition of new syntactic constructs and adapts the parser accordingly. Each syntactic extension defines an interpreter function that determines the meaning of the new construct.

Syntactic extensions can be organized in modules, whose import activates them in the current scope. Similar to SugarJ, this provides modular reasoning and allows for the composition of extensions. A syntactic extension is defined using an EBNF-like language, whereas the interpreter for syntactic extensions is written in regular, reusable Katahdin code.

The *Fortress* programming language features a macro system with user-defined macro call syntax [ACN+09]. Fortress uses parsing expression grammars [For04] to support an extensible core syntax. The semantics of syntactic extension is given as syntax transformation with quasiquoted syntax in a designated transformation language. Despite its syntactic flexibility, Fortress does not require a physical separation of macro definitions and macro call sites. Instead, Fortress applies a two-phase parsing approach that first recognizes all macro definitions while ignoring all other code. From the macro definitions, Fortress constructs a new grammar that is used to parse the main program and the quasiquoted syntax in transformations. As consequence, Fortress macros are organized similar to other top-level definitions and modular reasoning is supported. Furthermore, Fortress macros are hygienic and compose based on the ordered choice of parsing expression grammars.

Converge is a programming language with macro-like metaprogramming facilities [Tra08]. In particular, Converge features metaprogramming with quasiquotation in regular Converge functions that are run at compile time. In addition, Converge supports domain-specific syntax through DSL blocks. A DSL block `$<<expr>> indented-code` forwards the unparsed but indented code `indented-code` as a string to the user-designated function `expr`. The function `expr` implements the DSL by dynamically parsing the code and producing a syntax tree that replaces the DSL block. The DSL implementation can also perform domain-specific analyses on the parsed syntax tree and Converge provides facilities for reporting errors. Converge supports modular reasoning since DSL blocks explicitly refer to the DSL implementation. Moreover, Converge's quasiquotation mechanism retains referential transparency by hygienic macro expansion. The metaprogramming facilities of Converge are uniformly self-applicable. In fact, Converge supports the declarative specification of DSL blocks by self-applying the DSL-block concept to itself in order to introduce an EBNF-like grammar language.

The Haskell programming language features macro-like metaprogramming with Template Haskell [SP02]. Template Haskell supports domain-specific syntax in user-programs similar to Converge: Haskell programmers can use the quasiquotation construct `[:expr|code|]` to forward the unparsed code `code` as a string to the function `expr` [Mai07]. Like in Converge, `expr` parses the code and produces an abstract syntax tree that replaces the quasiquote. In contrast to Converge, Template Haskell only guarantees referential transparency when the DSL implementation uses quasiquotation for generating Haskell code. However, Sheard and Peyton Jones report that quasiquotation cannot express all the desired metaprogramming applications [SP02]. For such case, Template Haskell provides constructors for abstract syntax trees that can be used instead of quasiquotation,

but do not provide referential transparency. In fact, any Haskell program can be used as a compile-time function in Template Haskell. Template Haskell does not provide a declarative formalism for specifying domain-specific syntax. Instead, the parser of the DSL must be written in Haskell. However, Template Haskell supports uniformity, as long as the definition and the use of a domain abstraction are located in different files. Therefore, it should be possible to provide a declarative syntax formalism using Template Haskell itself.

Racket provides a macro system that organizes macros in libraries [Fla02]. Racket macros can specify new syntax, but always require a unique leading keyword, the macro name. This is limiting, for example, in the embedding of literal XML syntax. Racket macros are expanded at compile time and can conduct domain-specific static analyses on the macro arguments. Like SugarJ, Racket has been proposed as a host language for library-based language extensibility [THSAC+11, Fla12] that features composability and provides referentially transparent and hygienic macro expansion [CR91, KFFD86]. In contrast, SugarJ transformations are not hygienic. On the other hand, SugarJ employs non-local term rewriting instead of local macro expansion. This gives SugarJ more flexibility in code generation, but complicates referential transparency as discussed in Section 5.5.4. Racket uses the declarative **syntax-rules** construct to declare macros. A macro defines valid syntactic patterns (the new syntax) and code templates, which are instantiated on macro expansion. Templates can produce applications of other Racket functions and macros, which enables implementation reuse across macros definitions. Finally, Racket macros can expand into new macro definitions. This enables extension of the macro system through macros themselves.

In addition, Racket provides facilities for adapting its lexical syntax (using readers) and thus supports more flexible syntactic embeddings of DSLs [FBF09]. However, Racket lacks support for a declarative syntax formalism, and reader implementations do not compose well. The entries in Table 8.1 for Racket correspond to a usage of Racket without readers.

Honu is a programming language that extends the Racket macro system to support more flexible macro call syntax [RF12]. Similar to Nemerle, Honu macro calls require a unique macro name that can be followed by mixfix notation to describe the macro's argument list. The syntactic extension associated to a macro is declared by the macro's signature: Each macro argument is annotated by a user-definable syntax class such as numbers or regular expression. Macro arguments can be separated by lexical constructs such as commas or parentheses. This information is used to parse macro calls. Honu does not address domain-specific analyses so far. Honu retains all the principles of the Racket macro system.

8.4 External extensibility

Systems that support external extensibility add domain abstraction to a programming language through external tools. Such tools include preprocessors, transformation systems, and extensible compilers. Dual to internal extensibility, systems that support external extensibility rely on external configuration to activate or process domain abstractions. Therefore, these systems inhibit modular reasoning.

Camlp4 is an extensible preprocessor and pretty printer that is targeted especially at extending the syntax of the functional programming language OCaml. Camlp4 supports domain-specific syntax with an in-memory representation of the current grammar that is interpreted by a recursive descent parser. Backtracking can be enabled or disabled for individual rules. Syntax extensions are written in an EBNF-like grammar formalism and mutate the current grammar in order to install additional productions. To activate a Camlp4 syntax extension, a programmer needs to run the preprocessor with the corresponding extension definition on the source file. Thus, the configuration of syntax extensions is external to the source file, which inhibits modular reasoning. Camlp4 uses OCaml for specifying transformations but supports quasiquotation as well. Camlp4 transformations are not hygienic. Camlp4 is self-applicable and in fact, Camlp4 includes several extensions that are targeted at language extension authors, including the declarative grammar and transformation notations. Camlp4 can also be adapted for other languages than OCaml. However, language extensions themselves still have to be written in OCaml.

Nemerle [SMO04] is a programming language that uses an extensible compiler to support metaprogramming. Nemerle uses a macro-like mechanism to define compiler extensions that support a restricted form of syntactic extensibility, where each syntax extension must start with a unique token followed by mixfix notation. Nemerle compiler extensions are activated via a special compiler argument. This inhibits modular reasoning as the set of active macros cannot be determined from the user's source file. Nemerle supports hygienic code transformation. These transformations are regular, reusable Nemerle programs that can employ quasiquotation to declaratively describe syntax trees. Nemerle macros compose due to the requirement on unique first tokens in syntax extensions. Like other macro systems, Nemerle macros are uniformly applicable: Macros can be used to define other macros, which requires multiple applications of the compiler configured with different extensions.

The Java Syntactic Extender (JSE) also uses macros in external files to provide domain abstraction [BP01]. JSE targets the Java programming language and uses a preprocessor that expands macros to generate Java source files. When using a syntactic extension, the user must call the preprocessor by hand and configure it to eliminate the appropriate syntactic extension. Then the user can call `javac` on the resulting source files. The original JSE paper [BP01] describes a design for a hygienic code generation, but no implementation or other form of evaluation is available. To declare a syntactic extension with JSE, a

programmer provides a name for the macro, a mixfix argument list, and a generation template. The generation template is written in Java augmented with quasiquotation, and can be factored out into reusable Java methods. Since the implementation language and the object language both are Java, JSE features uniform self-application, which requires multiple calls to the preprocessor with different configurations.

MetaBorg is a language-agnostic preprocessor framework for syntactic extensibility based on SDF an Stratego [BV04]. MetaBorg uses SDF grammars to model and extend the object language's syntax with arbitrary context-free syntactic extensions. An extension-specific Stratego transformation desugars the user program into a program of the object language without extension. The execution model of MetaBorg is preprocessor-like, that is, users must manually configure and apply MetaBorg to their programs. Since MetaBorg is based on the flexible transformation engine Stratego, it cannot guarantee referential transparency. MetaBorg particularly promotes the use of language-agnostic quasiquotation in transformations, which enables declarative specifications of domain abstractions. As discussed for SugarJ above, SDF and Stratego are well-suited for the definition of composable language extensions. The extension mechanism of MetaBorg is self-applicable because MetaBorg is language-agnostic: The preprocessor can apply to SDF and Stratego programs as well.

JastAdd is a framework for extensible compilers based on aspect-oriented programming and attribute grammars [EH07b]. Compiler extensions are defined as aspects that are woven into the base compiler to extend it. The implementation of an extension can be given as Java code or as declarative reference attribute grammars [EH04], which are well-suited for extensible tree traversals and can be used to define domain-specific analyses. Moreover, JastAdd allows the use of attribute grammars to define domain-specific editor services [SH11]. For parsing, JastAdd uses an extensible grammar formalism that generates an LALR parser. JastAdd language extensions are activated from the command line by configuring and calling JastAdd on the source files. Withing the extension definitions, JastAdd supports reuse and refinement of attributes and equations from other extensions. This way JastAdd enables language unification as discussed in Chapter 7.

Silver is an attribute-grammar system that features extensibility [VBGK10]. Given a host-language implementation in Silver, other Silver modules can extend the syntax, semantics, and analysis of the host language. For extensible parsing, Silver employs context-aware scanning [WS07]. Silver extensions can build on attributes and equations defined in other modules, and attribute grammars provide a declarative means for the definition of such extensions. Silver supports uniformity and, in fact, is implemented in itself, where a small core system is extended with convenient features such as pattern matching on syntax trees or collection attributes.

Silver supports the composition of domain abstractions, but favors guaranteed composability over flexibility. In particular, the Silver developers investigate language extensions that are modular in the sense that their composition always succeeds [SV09, KV12]. This

has the benefit that users of multiple extensions never are exposed to composition errors. On the other hand, this guarantee places some restrictions on what kind of extensions are supported. For example, to guarantee syntactic composability, extension-specific syntax must start with a unique token.

Polyglot is an extensible compiler front-end for Java that supports customization of compiler phases [NCM03]. Extensions can declare domain-specific syntax through an extensible grammar formalism that generates an LALR parser. For semantics and static analysis, extensions can add, replace, reorder, and remove phases of the compiler, which gives extensions a lot of flexibility. Polyglot compiler phases are implemented in Java, heavily relying on abstract factories, delegation, and proxies to enable extensibility. This enables reuse between different extensions but does not provide a declarative mechanism for analysis or code generation. Moreover, Polyglot does not target the composition of extensions and it is not clear to which extend abstract factories, delegation, and proxies foster or inhibit composability. However, Polyglot supports uniform self-application because it uses Java both as object language and as implementation language of extensions.

8.5 Language workbenches

Language workbenches are tools that integrate traditional language engineering tools, such as parser generators and transformation systems, and tools to develop IDE support [Fow05b]. By combining these tools and by providing IDE support for these metaprogramming tasks, language workbenches enable developers to create new languages with IDE support.

Meta-environment is a language workbench based on the syntax and transformation engine ASF+SDF [Kli93, vdBvDH+01]. Developers can declare domain-specific syntax using SDF. The Algebraic Specification Formalism ASF allows developers to declare domain-specific semantics and analyses for their syntax through rewrite equations. The Meta-environment uses a generic syntax-directed editor that the system configures according to the language of the current file. In particular, the generic editor is used for language definitions and programs written in the defined languages. However, the editor is not configurable; it only highlights keywords and marks errors in the source code. ASF+SDF definitions are organized in modules that enable the reuse of productions and equations in the implementation of multiple domain abstractions. Furthermore, ASF+SDF is declarative and composable: When importing multiple modules, all productions and equations become available. ASF+SDF is not self-applicable, that is, only ASF+SDF and no other DSLs can be used to define languages. The Meta-environment project is continued by the Rascal metaprogramming language [KvdSV09, vdS11].

The *Spoofax* language workbench uses SDF and Stratego to define domain-specific syntax, semantics, and analyses, but also features a declarative language for the specification of domain-specific editor services [KV10]. From a simple editor-service declaration,

Spoofax generates an Eclipse plugin for the DSL that features syntax coloring, outlining, code completion, reference resolution, and more. Moreover, Spoofax supports the coevolution of a DSL and programs written in it, because Spoofax regenerates the domain-specific editor on-the-fly and no separate Eclipse instance is necessary. Spoofax editor services are declared in specific files in an Eclipse project, and no module system is available to foster reuse of definitions across projects. Still it is possible to compose DSLs by manually loading all relevant definitions into a single project. Spoofax supports the definition of domain-specific syntax, semantics, analyses, and editor services with domain-specific editor services for SDF, Stratego, and editor-service declarations. However, it is not possible to uniformly abstract over language definitions. The Spoofax language workbench was the basis for our uniformly extensible IDE described in Chapter 3.

The Meta Programming System (MPS) is a language workbench based on intentional programming [Völ11, VS10]. According to the intentional-programming paradigm, MPS avoids parsing and employs projectional editing instead. Therefore, instead of defining a grammar, DSL developers define editor actions that provide convenient and efficient ways to input programs. While this avoids potential syntactic ambiguities when composing languages, the composition of editor actions may also be conflicting. MPS organizes implementations of domain abstractions in modules. Dependencies between artifacts are specified within a property dialog for each artifact separately; dependencies are not part of the textual projection of a program, which limits the support for modular reasoning. Nevertheless, the module system supports reuse of implementation artifacts. MPS strictly separates metaprogramming and programming by providing fixed DSLs for the definition of editor actions, data schemas, code generation with quasiquotation, and others.

Cedalion is another language workbench based on intentional programming [LR11]. Cedalion provides constructs to declare, project, analyze, and evaluate domain abstractions. For the latter two, Cedalion builds on logic programming and represents domain abstractions as logical relations. The semantics of a domain abstractions then specifies the preconditions for a domain-specific relation to hold. Cedalion definitions are managed in a global namespace: All definitions in the current Eclipse workspace are available in a source file. This inhibits modular reasoning but allows predicates to be reused in the implementation of other predicates. Logic programming with syntactically flexible projections of predicates provides a rather declarative means for implementing domain abstractions. Furthermore, it enables the composition of domain abstractions, where each abstraction adds more constraints to the currently defined predicate. While this is a meaningful definition, it is not clear whether it provides a useful model for DSL composition. Finally, Cedalion has been bootstrapped to provide uniformity, which allows Cedalion programmers to extend the system itself.

Xtext is a popular model-driven language workbench that gives developers lots of flexibility [EV06, Xte12]. Xtext supports grammars, validators, code generators, and IDE providers for the implementation of all aspects of a domain abstraction. Moreover, Xtext code generators are configurable through workflows. A workflow loads domain-

specific programs and prescribes arbitrarily complex transformation schemes for these programs. Essentially, a workflow is a build script that applies to a whole project and inhibits modular reasoning. A direct consequence of this is the lack of separate compilation, which results in overly long compilation times for model-driven software projects [KMT12]. Xtext grammars, transformations, and workflows are organized in reusable libraries. Moreover, these artifacts are defined using declarative notations for the respective domain. Xtext focuses on standalone DSLs and domain abstractions that desugar to Java code. However, DSLs build with Xtext do neither compose with each other nor with the Java programming language. The only supported interaction is via the generated code. Moreover, Xtext does not support domain abstraction over the used metalanguages for grammars, transformations, workflows.

Monticore is a language workbench that particularly targets the composition of DSLs [KRV10]. To this end, Monticore provides a grammar formalism that features inheritance and embedding: Inheritance enables the incremental extension of a grammar by providing additional productions for a nonterminal. Reversely, embedding declares some nonterminals of a grammar as abstract, which enables clients of the grammar to specialize it by embedding a language into it. From a grammar, Monticore generates a parser for the concrete syntax and a Java encoding of the abstract syntax. For editor services, Monticore provides a declarative configuration language similar to Spoofax. For domain-specific analysis and code generation, Monticore uses regular Java programs and the visitor pattern. This does neither provide the same declarativity nor the same composability as Monticore's grammar formalism. However, Monticore employs the DSLTool framework, which provides an architecture for domain-specific analysis and code generation that enables polymorphic interpretation of domain-specific programs. Essentially, a semantics is represented as an object of the abstract class DSLRoot, which, similar to a build script, can apply different analyses and generators to inspect and transform a user program.

Epsilon [KRPGD12] is a modeling framework that provides a set of DSLs to analyze, compare, transform, and refactor domain-specific programs in the form of EMF [SBPM08] models. Epsilon does not support domain-specific syntax or editors directly, besides simple support Human-Usable Textual Notation (HUTN) [Obj04]. However, Epsilon can reuse frontend support for EMF models as provided by Xtext [Xte12] or EMFText [HJK+09]. For static analysis, Epsilon provides the declarative Epsilon Validation Language for the specification of invariants and other form of checks. For code generation, Epsilon provides separate model-to-model and model-to-text transformation languages. Both provide declarative constructs to transform a model, but neither ensures referential transparency. All Epsilon languages are organized as modules that can be reused in the implementation of different domain abstractions. Epsilon supports the application of a transformation to a model through a GUI action or through the definition of a workflow model. Both mechanisms inhibit modular reasoning since the transformation application is transparent to clients of the generated code.

8.6 Chapter summary

The list existing systems that support domain abstraction in one way or another is long. In this chapter, we provide an overview of systems that are currently in use or have been developed in recent years. We describe these systems according to their support for flexible and principled domain abstraction and provide an overview in Table 8.1.

Our survey has two goals beyond giving an overview. First, we want to put our own system for domain abstraction SugarJ into context of related work. As our comparison shows, SugarJ does not provide any new features, but unifies existing features in a unique way. This makes SugarJ very flexible, with support for domain-specific syntax, polymorphic semantics, domain-specific analysis, and domain-specific editor services, and quite principled, with support for modular reasoning, declarative implementations, DSL composition, and uniformity. While there are equally flexible systems (Xtext and Monticore) and more principled systems (Racket and Honu), SugarJ is the only system that combines high flexibility with strong principles.

The second goal of our survey is to evaluate our design goals for flexible and principled domain abstraction. In particular, our survey shows that our design goals for domain abstraction are relevant and sufficient to distinguish existing systems. The design goals are relevant since each is realized by a number of existing technologies. Moreover, the design goals are sufficient to characterize all systems we investigated in our survey. In fact, as Table 8.1 shows, no two systems have the same characteristics. Probably, our design goals are incomplete still because some desirable features for domain abstraction have not been identified yet. Future work will show what other features developers of domain abstractions require.

9 Conclusion and Future Work

Domain-specific languages promise to ease the development of software by raising the abstraction level to the level of domain constructs. This narrows the representational gap [Lar02] between the way programmers think about domain concepts and the way programmers encode the domain concepts in their programs. However, for DSLs to unfold their full potential, the employed domain abstractions must be flexible enough to support a complete domain-specific frontend (syntax, analysis, tooling), and principled enough to not interfere with best practices such as modular reasoning or code reuse. In this thesis, we identified eleven design goals for flexible and principled domain abstraction, developed a language that satisfies these goals, and evaluated it through numerous case studies.

We propose *extensible programming languages* for the definition of flexible and principled domain abstractions. In an extensible programming language, domain abstractions are encoded as language extensions, and programmers use the extended language to write domain-specific programs. Furthermore, we propose to represent *language extensions as libraries*, that is, as scoped, reusable, and composable components. A programmer can activate a language extension by importing the corresponding library into the current scope. A programmer can reuse the implementation of a language extension by importing the corresponding library into another extension definition. And a programmer can compose multiple language extensions by importing all corresponding libraries into a single scope.

Domain abstractions implemented as language extensions resemble internal DSLs, because the implementation reuses the existing abstraction mechanisms of the host language. As consequence, our approach provides the advantages of internal DSLs such as modular reasoning. However, extensibility is a powerful abstraction mechanism that enables internal DSLs as flexible as their external counterparts. In our approach, the flexibility of the supported domain abstractions directly depends on the flexibility of the extension mechanism: An extensible syntax enables domain-specific syntax, extensible static analysis enables domain-specific checking, and extensible tool support enables domain-specific editor services. Accordingly, a flexibly extensible programming language combines the advantages of internal and external DSLs.

We designed and implemented the flexibly extensible programming language SugarJ that organizes language extensions in libraries. SugarJ's extensibility mechanism is flexible and supports domain-specific syntax, polymorphic semantics, domain-specific static analysis, and domain-specific editor services. This enables domain abstractions as flexible as external DSLs. Despite this flexibility, SugarJ's extension mechanism is principled: It organizes extensions in libraries, which enables modular reasoning,

implementation reuse, composition, and uniformity like in internal DSLs. Moreover, SugarJ checks communication integrity and builds on SDF and Stratego to provide a declarative mechanism for the implementation of extensions.

On top of SugarJ, we developed an extensible IDE that programmers can coevolve with the language. Like SugarJ, our IDE organizes editor extensions in libraries: A programmer can activate an editor extension by importing the corresponding library alongside a language extension, or the programmer can package these two artifacts together into a single library. For each file, our IDE inspects the libraries in scope to determine the set of activated editor extensions. The IDE then presents the corresponding editor services to the user. Our IDE provides syntax highlighting for user-defined language extensions by default, but programmers can define more sophisticated editor services such as code completion or reference resolution. This way, our IDE can provide a user experience for extensible programming languages similar to IDEs of nonextensible programming languages.

In summary, our design and implementation of SugarJ and its IDE demonstrates that *extensible languages enable flexible and principled domain abstraction.* In addition, we evaluated the applicability of our system by conducting numerous case studies. We developed the following language extensions in SugarJ: closures for Java, entity data schemas, Java Server Pages with HTML and JavaScript, Latex and Bibtex, regular expressions, software-product-line development with \#ifdef, statemachines, a template engine, XML, and XML Schema. In all these case studies, we were able to integrate domain-specific syntax into SugarJ using a library-based language extensions. We were able to modularly reason on the set of active language extensions, compose language definitions to support multiple domains in a single file, and get assistance by domain-specific static analysis and domain-specific editor services. Moreover, some of our case studies make use of the uniform design of SugarJ and, in fact, apply to SugarJ's metalevel. For example, XML Schema is a meta-DSL that provides domain abstraction for the declaration of a static analysis, and our template engine provides an alternative implementation model for the declaration of program transformations. In general, our case studies explore and exploit all features of SugarJ. Appendix A includes detailed descriptions of all our case studies.

Based on our experiments with SugarJ, we found that library-based extensibility is not specific to Java but similar extensibility is applicable to other languages. Following this insight, we generalized the SugarJ compiler into a framework for library-based language extensibility that can be easily instantiated for new base languages. The only requirement on base languages is that they use a module system to organize code, where language extensions can only be activated by import statements that appear at the top-level of a source file. In practice, most programming languages are admissible according to this requirement. We instantiated our framework for Java, Haskell, Prolog, and F_ω to provide support for flexible and principled domain abstraction in each of these languages. Each extensible language built with our framework enjoys the same flexibility

and principles that the original SugarJ compiler and IDE provided. By generalizing the SugarJ compiler and IDE into a framework that supports different base languages, we effectively demonstrate that library-based language extensibility is a metaprogramming technique that generalizes to a wide range of programming languages.

Our work constitutes an important step toward a wider application of domain abstraction in practice for the following reasons. First, the flexibility of our approach provides a high gain for software developers and makes the application of domain abstractions attractive. Second, the principles we follow delimit the risk and cost of using domain abstraction, because we retain best practices, support reuse, and provide declarative implementation languages. Third, our approach facilitates domain composition and polymorphic domain abstraction to support complex software systems that address concerns from different domains and technical spaces. For these reasons, we believe that library-based language extensibility is well-suited for modeling practical systems, for which our case studies on the graph product line and the Java Pet Store give initial evidence.

Besides practical application, our work provides benefits for language designers, because the extensibility of SugarJ makes it an attractive platform for language-design experiments. A language designer can evaluate a design idea with SugarJ by first implementing the design as a language extension and then using the extended language to experiment with the design. This can be useful for experiments on the design of small language extensions, such as our extension of Java with closures, as well as on the design of whole languages, such as our statemachine DSL. SugarJ is well-suited for such experiments because it provides tool support for the extended language, has no stratification into metalevels that requires, for example, to start a new Eclipse instance, and it supports the evaluation of the interaction between different designs by composing the corresponding language extensions. At the time of writing, multiple research projects use SugarJ as a platform for design experimentation.

Future work. SugarJ addresses all and satisfies most of our design goals for flexible and principled domain abstraction. The design goals SugarJ only partially achieves are implementation reuse and referential transparency. We suggest addressing these goals more adequately in future work. For implementation reuse, future work could impose a module system on SDF and Stratego that provides namespace management and more fine-grained control over code reuse: partial import, qualified import, rename before import, import-as-extension versus import-as-library. For referential transparency, we suggest the investigation of hygienic program-transformation systems in general. A hygienic program-transformation systems must guarantee that variable resolution is invariant to the application of transformations. This requires the transformation system to know about the binders and scoping of the generated language. To this end, one interesting avenue of future work is to use the Name Binding Language [KKWV12], a

DSL for the declaration of binders and scoping, to generate transformations for systematic renaming. In the context of extensible programming languages, an interesting question for hygienic transformations is whether it is sufficient to declare name bindings of the base language, as supported by the Scheme macro system [SDF+09], or whether language extensions need to declare extended scoping rules, too. Moreover, the efficiency of the hygiene mechanism is important [DHB92] and constitutes an interesting challenge that precludes naive renaming strategies.

As described in this thesis, we realized our design for SugarJ as a compiler. However, our compiler retains a preprocessor character: It processes and reacts to import statements by changing the current grammar and desugaring transformation, but, in the end, out compiler emits plain base-language source code that we compile with an off-the-shelf compiler of the base language. This implementation strategy factors our technicalities of the base language and enabled us to focus on the novelties of SugarJ instead. It would be interesting to investigate the benefits of a tight integration of our extensibility mechanism and the base-language compiler. One immediate advantage would be the avoidance of pretty printing as a means for communicating generated code to the base-language compiler. Instead, a tight integration enables communication via a uniform program representation, which, in particular, can easily retain source-position information. Furthermore, a tight integration unifies the exception handling and abolishes the need for parsing the output of the base-language compiler to recognize errors. In addition, a tight integration can improve the expressiveness of language extensions. For example, a tight integration should enable type-dependent transformation that use the type-checking or type-inference engine of the base language to decide what code to generate. Finally, a tight integration may give rise to a process for outsourcing features implemented by the compiler into library-based language extensions, which simplifies the compiler and provides more flexibility to programmers. Reversely, the integration may give rise to a process for incorporating library-based extensions into the base-language compiler, which elevates our approach from extension prototyping to compiler development.

Another area of future work are extensible static analyses. SugarJ supports language extensions to define static analyses that, when in scope, are run prior to desugaring to validate the source code. Other works on extensible static analysis interleave desugaring with static checking [FS06] or fully desugar the code before analyzing it [THSAC+11]. Independent of the order of desugaring and analysis, the question remains: How can we guarantee the soundness of the extended type system? Specifically, we want to ensure that if the extended type system declares a program well-typed, then the *fully desugared program does not go wrong*. This guarantee renders analysis of generated code unnecessary because the extended analysis already rules out run-time type errors. In a system like SugarJ that performs analysis prior to desugaring, the soundness of the extended analysis constitutes a dramatic improvement in the quality of error messages, because errors never are reported in terms of generated code. We are currently working on a framework that guarantees the soundness of extended analyses given a sound base

analysis. In our framework, every extension of a static analysis must be accompanied by a proof that shows that the extension only accepts programs whose desugaring is accepted by the base analysis. This entails the soundness of the extended analysis. In fact, we plan to synthesize these proofs using a combination of symbolic execution and the base analysis itself. Early experiments with an extensible F_ω type system suggest the feasibility of our synthesis procedure.

In conjunction with our future work, we believe that flexible and principled domain abstraction, as presented in this thesis, elevate DSLs to their full potential as a scalable methodology for the implementation of complex software systems.

Appendix

A List of Case Studies

We summarize the 14 case studies that we conducted as evaluation of SugarJ, SugarHaskell, and JProMo. All case studies are open-source and available via `http://sugarj.org`.

A.1 Case studies with SugarJ

Closures

Version:	April 5, 2011
Developed by:	Sebastian Erdweg, Tillmann Rendel
Size:	171 lines of SugarJ code (9 files) and 192 lines of Java with closures code (3 files)
Description:	Closures (also known as lambda expressions or anonymous functions) introduce functions as first-class citizens into Java. We implemented closures as a sugar library, following the proposal of Gafter and von der Ahé [GvdA09] for integrating closures into the Java programming language. We used our closure embedding to implement a simple yet powerful list API for Java that features higher-order functions such as map, sortBy, or zip [EKR+11b].
Results:	This case study demonstrates that even sophisticated programming-language features can be implemented as syntactic sugar, and SugarJ is an implementation platform well-suited for the extension of the host language with new programming-language concepts.

Java Pet Store

Version:	July 11, 2012
Developed by:	Stefan Fehrenbach as part of his bachelor thesis [Feh11]
Size:	2896 lines of SugarJ code (37 files) and 4909 lines of reengineered Java code (55 files)
Description:	The Java Pet Store is a reference application for Java Enterprise Edition originally developed by Sun Microsystems [Sun02]. It implements a web store for trading pet animals. The implementation follows the model-view-controller design pattern and makes use of AJAX for dynamically updating sites.

We used the Java Pet Store to experiment with the practical adoption of syntactic sugar as provided by SugarJ. We integrated sugar libraries that provide domain abstraction for field access (similar to Java beans), XML, XML Schema, JPQL, and EBNF. We reengineered part of the implementation of the Java Pet Store to make use of these sugar libraries.

Results: This case study shows that domain abstraction in the form of sugar libraries can be adopted to practical applications. In particular, sugar libraries prevent syntactic errors that occur in the originally used string embedding of XML and JPQL, sugar libraries reduce boilerplate in the definition of field accessors, sugar libraries support additional domain-specific checks such as XML Schema validation, and sugar libraries are equipped with appropriate editor support to assist programmers writing domain-specific code.

Additionally, this case study let to the development of a novel methodology for the implementation of DSLs in existing legacy applications. With sugar libraries, the code base of a legacy application can be *incrementally* reengineered to improve maintainability: Sugar libraries can be added incrementally to support more domains, and sugar libraries can be adopted incrementally to lift more code to the domain abstractions. For the latter, it is essential that SugarJ promotes the use of syntactic sugar that has no semantic consequence. Therefore, the reengineering of one library does not influence code in other libraries. This enables the benefits of domain abstraction in large legacy applications.

Java Server Pages

Version: August 9, 2011

Developed by: Selman Halid Kahya during an internship at the University of Marburg, Sebastian Erdweg

Size: 3490 lines of SugarJ code (48 files) and
352 lines of HTML, JavaScript, and JSP code (6 files)

Description: Java Server Pages (JSP) is a DSL for describing dynamic web pages based on Java servlets. JSP combines HTML, JavaScript, and Java into a single language that supports web pages with client-side and server-side scripting. JSP uses HTML for the description of a web page's initial view. As usual, JavaScript can be embedded in the HTML document to enable client-side scripting. For server-side scripting, JSP defines integrates HTML and the Java programming language, where the Java code is syntactically embedded into a HTML document. Semantically, JSP compiles to a Java servlet that generates HTML documents at

runtime on the server. Since JSP compiles to Java, it should be possible to fully realize JSP as a sugar library. So far, we only the developed the frontend part of JSP: syntax and some editor services.

Results: This case studies shows that (i) language composition occurs in practice and (ii) SugarJ supports composition of modularly developed sugar libraries. Concretely, we defined the syntax of HTML, JavaScript, and Java in isolation and composed them to form the JSP syntax. This way, we are able to support the JSP. As future work, we want to define the compilation of JSP to a Java servlet as a desugaring in SugarJ. Furthermore, JSP supports a simple form of extensibility via tag libraries. We plan to support tag libraries as a meta-DSL for our JSP implementation.

Latex and Bibtex

Version: June 21, 2011

Developed by: Sebastian Erdweg, Lennart Kats, Tillmann Rendel

Size: 1588 lines of SugarJ code (14 files) and
3010 lines of embedded Latex and Bibtex code (7 files)

Description: Latex and Bibtex are DSLs for typesetting. We embedded them into SugarJ by defining sugar libraries for small, parsable subsets of each language. We further subdivided Latex into different language aspects and implemented each as a separate sugar library: basic Latex commands for structuring and typesetting of documents, mathematical formulas, code listings, bibliographical citations. The latter library integrates with the embedding of Bibtex. We used this Latex and Bibtex embedding to write our GPCE'11 paper [EKR+11a].

Results: We conducted this case study to evaluate the composability of editor services in our extensible IDE. In particular, this case demonstrates the effectiveness of our explicit-coordination scheme that we use to coordinate between a bibliography written in Bibtex and citations specified as part of a Latex document. Moreover, this case study shows how sugar libraries lend themselves for decomposing larger languages into smaller aspects that can be separately implemented and composed.

Regular expressions

Version: May 29, 2011

Developed by: Sebastian Erdweg

Size: 214 lines of SugarJ code (4 files)

Description: Regular expressions are a simple DSL that provides efficient matching of lexical patterns in strings. Most languages realize regular expressions as a string embedding, where a run-time parser and compiler handles regular expressions. In contrast, this case study promotes regular expressions as first-order language constructs that are parsed at compile time and desugar into the usual string encoding. In addition, we provide syntax coloring and code completion with explanations.

Results: The development of the regular-expression case study is straightforward and requires little effort. This case study shows that the effort in developing sugar libraries scales down to small language extensions. This indicates the feasibility of a syntactically extensible core language, where all advanced language constructs are realized through sugar libraries.

XML and XML Schema

Version: September 13, 2011

Developed by: Sebastian Erdweg

Size: 2792 lines of SugarJ code (19 files) and
596 lines of XML and XML Schema code (9 files)

Description: We implement a syntactic embedding of XML into Java. This embedding enables programmers to use literal XML syntax within a regular Java program, and to splice dynamic Java values into the XML document. Our embedding desugars to method calls of the SAX API, that is, an XML document is decomposed into events that describe the beginning and ending of an XML element.

On top of XML, we implement support static XML validation. To this end, we provide an embedding of XML Schema into SugarJ as a sugar library. XML Schema reuses XML syntax, but desugars into another sugar library that implements a validator for the given XML schema. When importing the library that defines an XML schema, the generated validator is activated to statically validate XML documents of the corresponding namespace.

Results: XML is a language that uses a syntax different from most programming languages. Therefore, XML is a good example of a DSL that cannot satisfactorily be implemented with pure embedding, because the host language does not support XML literals. In contrast, SugarJ supports the integration of arbitrary context-free languages.

The XML Schema case study illustrates two points. First, it shows how a sugar library can be used to implement a static analysis on

domain-specific programs. In our SugarJ, we define domain-specific analyses as program transformations in Stratego that transform the user program into a list of error locations and error messages. An import of the defining library activates a static analysis in the current module. Second, the XML Schema case study shows the power of uniform self-applicability: We can use domain abstraction to build XML Schema as a DSL for definition of domain-specific static analyses on XML. That is, we can provide domain abstraction on top of the abstraction mechanism itself. We call an abstraction such as XML Schema a meta-DSL as it is used to implement other DSLs.

A.2 Case studies with SugarHaskell

Arrows

Version:	June 18, 2012
Developed by:	Sebastian Erdweg
Size:	273 lines of SugarJ code (6 files) and 102 lines of Haskell with arrows code (2 files)
Description:	Arrows generalize monads to computations with multiple inputs and outputs. Since arrow combinators are difficult to use, Paterson propose a new notation for arrows [Pat01]. An extended version of this notation was integrated into Haskell by GHC as a compiler extension. In particular, the extended arrow notation features arrow-specific do notation, which requires a layout-sensitive parser.
Results:	We realized the extended arrow notation in SugarHaskell by writing a sugar library. In particular, this case study illustrates the following features of SugarHaskell. First, SugarHaskell extensions can be layout-sensitive, using our declarative layout constraints in productions of the extension grammar. Second, SugarHaskell allows developers to integrate customary layout-sensitive syntax for transformations. In particular, we used concrete arrow syntax for pattern matching in the desugaring of the arrow sugar library, and concrete Haskell syntax for code generation. In particular, when generating larger code fragments, layout-sensitive concrete syntax can reduce the accidental complexity by retaining the look-and-feel of the target language.

EBNF

Version:	June 3, 2012

Developed by:	Sebastian Erdweg
Size:	309 lines of SugarJ code (5 files) and 99 lines of Haskell with EBNF code (2 files)
Description:	Haskell traditionally supports the description of parser by parser combinators. While parser combinators are expressive and flexible, they do not provide the same declarativity as EBNF-based grammar formalism. Moreover, a grammar typically contains information on the abstract syntax as well as the concrete syntax of the described language. Parser combinators only address the latter aspect. The EBNF case study extends Haskell with syntax for the declaration of EBNF-based grammars. We desugar EBNF grammars into multiple artifacts: First, we generate a declaration of an algebraic data type that represents the abstract syntax oft he described language. Second, we generate a Haskell program that uses Parsec parser combinators [LM01] to represent the concrete syntax of the language. Our desugaring takes care of some technical issues related to parsing, such as whitespace and backtracking. To ease the use of the generated parsers, we also generate instance of the type class Read. Finally, we use SugarHaskell's self-applicability to generate another sugar library from a user's grammar, which enables programmers to use their concrete syntax in Haskell programs directly. Such user-language code fragments are parsed at compile time and translated into instances of the generated algebraic data type.
Results:	The EBNF case study illustrates the power of self-applicable extensible languages like SugarHaskell. A programmer can flexibly decide whether to parse a domain-specific expression at compile time or at run time. Also, this case study shows that it is possible, and in fact useful, to generate multiple artifacts from a single domain-specific program. We generate a data type, a object-language parser, and a metalanguage parser from an EBNF grammar. Moreover, we explored a design pattern that enables users of sugar libraries to decide which artifacts the desugaring should generate. This way, we enable users to select whether to only generate support for the abstract syntax, for the abstract and concrete syntax, or additionally a metaextension.

Idiom brackets

Version:	June 3, 2012
Developed by:	Sebastian Erdweg
Size:	35 lines of SugarJ code (1 file)

Description: Idiom brackets provide a simple syntactic abstraction on top of applicative programming with effects [MP08]. This case study implements idiom brackets as a sugar library for Haskell.

Results: This case study is straightforward and without surprises. It shows that the implementation effort of SugarHaskell extension scales down with the complexity of the extension. Accordingly, for idiom brackets, the sugar library is simple and easy to write.

A.3 Case studies with JProMo

Entity modeling

Version: May 11, 2012

Developed by: Sebastian Erdweg

Size: 802 lines of SugarJ code (20 files) and
87 lines of entity declarations (13 files)

Description: The modeling of data schemas as entities is a typical example used by MDD frameworks. Entities declare properties and functionality, and are independent of any particular execution platform. We realized an entity DSL as a metamodel library in JProMo. This library defines concrete and abstract syntax for entity declarations. We provide a separate transformation library that, when applied to an entity model, generates a class-based Java representation of the entity with getter and setter methods.

Furthermore, the entity case study explores self-application in an MDD setting. We provide a meta-DSL for the declaration of metamodels that separates the declaration of concrete syntax from the declaration of abstract syntax. Our meta-DSL enables the definition of a metamodel without any concrete syntax, which can already be used to program analyses or transformations for the metamodel, because these artifacts are independent of the concrete syntax. A user can add concrete syntax to a metamodel in a separate library, which corresponds to a model transformation that takes the metamodel as input and generates a regular JProMo metamodel with concrete and abstract syntax. In fact, this way a user can provide multiple concrete syntaxes for a single metamodel. When declaring a model instance of the metamodel, the programmer selects the concrete syntax by applying a corresponding transformation to the metamodel. Moreover, we provide a transformation that generates a default HUTN [Obj04] syntax for a user-defined metamodel.

Results: With this case study we explore the model-oriented programming paradigm. We exploit the separation of transformations and models to enable multiple semantics for the entity metamodel. Moreover, the case study shows that communication integrity does not prevent flexible domain abstraction. In fact, communication integrity provides a principled framework that directs the dependency managing of meta-DSLs. In this case study, communication integrity required us to import auxiliary libraries, which the generated code uses, in the transformation library. This way, the transformation is allowed to generated code that depends on the auxiliary library. For the user, this restriction provides a much clearer interface because dependencies are explicit in the original code.

#ifdef-based product-lines

Version: August 20, 2012

Developed by: Sebastian Erdweg, Jonas Pusch

Size: 1259 lines of SugarJ code (19 files) and
89 lines of Variability-aware Java code (6 files)

Description: Software product lines describe a set of related products by a single configurable code base. One way to implement a product line is to use conditional compilation with CPP #ifdefs. #ifdefs provide a form of syntactic abstraction to the developers that allows the inclusion or exclusion of certain fragments of code.

We implement language support for #ifdef-based product lines with libraries in model-oriented programming. For this, we provide a metamodel for variability-aware Java that supports the use of #ifdef statements at syntactically well-defined positions, such as classes, fields, methods, method parameters, statements, or expressions. This way, developers can declare libraries that encode a software product line. An application can consist of arbitrary many variable and non-variable libraries, which can be interconnected in both directions.

To configure a variable library, a developer specifies a feature configuration that selects some features and deselects others. We provide a simple DSL for the declaration of feature configurations as yet another library. In fact, feature configurations are a meta-DSL that compiles into a regular model transformation that takes a variable Java library as input and produces a regular Java library.

Results: This case study makes heavy use of polymorphic domain abstraction. In fact, the whole point of software product lines is to support multiple

products with different semantics through a single code base. Therefore, this case study demonstrates the flexibility enabled by polymorphic domain abstraction, and gives some indication for the relevance of this feature. Moreover, the #ifdef case study highlights our support for mixing models (variable Java programs) and code (regular Java programs). Since we organize models and code as libraries in the same technical space, they can freely depend on one another. Finally, this case study illustrates the usability of meta-DSLs, which we used to build domain abstractions for the declaration of feature models and feature configurations. This way, product-line developers do not need to concern with the complexity of model transformation. Instead, they can use a declarative formalism to select or deselect features.

Graph product line

Version:	August 18, 2012
Developed by:	Jonas Pusch, Sebastian Erdweg
Size:	1388 lines of SugarJ code (26 files)
Description:	We used above case study for #ifdef-based product lines to implement the standard graph product line [LHB01]. All feature models, feature configurations, and variable Java classes are expressed as JProMo libraries. Even the selection of a product, which initiates the product's generation, is declared within JProMo. Since a feature configuration is a domain abstraction that compiles into a regular model transformation, we can apply a configuration to a variable Java library as a transformation in import statements. Our product-line encoding can express the full graph product line. We provide multiple configurations that can be used to generate concrete graph libraries. Since in JProMo the configuration of a product line is declared within the language, a JProMo program can depend on multiple configurations of a variable Java library simultaneously.
Results:	This case study shows that model-oriented programming is expressive enough to encode whole programming paradigms such as #ifdef-based product lines. In particular, this case study demonstrates that our encoding supports typical product lines developed by others. However, our product-line encoding goes beyond what other frameworks can achieve, because we encode product lines as libraries of a larger application, and the feature configurations are part of that application, too. We plan to explore the applicability of model-oriented programming for advanced product-line engineering in future work.

Statemachines

Version:	August 20, 2012
Developed by:	Sebastian Erdweg
Size:	676 lines of SugarJ code (16 files) and 230 lines of statemachine code (10 files)
Description:	Statemachines are another typical example used by MDD frameworks. Again, we build language support statemachines with libraries in JProMo. However, simple finite statemachines are not expressive enough for modeling realistic protocols, because they cannot depend on external data carried by the events or managed as internal state in the machine itself. Therefore, we extend statemachines to data-dependent statemachines that have data as internal state and enable data-parameterized events. We develop data-dependent statemachines by reusing parts of the entity case study described above. In particular, we allow property declarations inside a statemachine for declaring internal data, and the transition function can query the internal and event-provided data using an expression language that is also part of the entity metamodel. We develop data-dependent statemachines by reusing the existing entity metamodel and transformation.
Results:	This case study demonstrates the support of model-oriented programming for composition across domains. We were able to reuse property declarations and the expression language from the entity metamodel unchanged, as well as parts of the transformation that translates an entity declaration into a Java program. Furthermore, we were able reuse the transformation from simple statemachines to Java, but we required some changes: We integrated extension points for the transformation of a state transition's premise and consequence. This allowed us to later add functionality for data-dependent premises and data-mutating consequences.

Template engine

Version:	August 16, 2012
Developed by:	Sebastian Erdweg
Size:	590 lines of SugarJ code (15 files) and 285 lines of code templates (3 files)
Description:	Most existing MDD frameworks employ a template engine for the generation of code. In contrast, JProMo uses the transformation language Stratego. In this case study, we realize a model-to-model template en-

gine as a meta-DSL in JProMo. The template engine enables developers to write concrete Java code, interspersed with Stratego expressions to inject model-dependent code fragments. We used the template engine in the implementation of the statemachine case study.

Results: This case study demonstrates the power of uniform self-applicability, which enables meta-DSLs that abstract over technicalities of the transformation system. Moreover, the resulting template engine is still user-extensible: If a feature is missing, developers can add it themselves via a library. To the best of our knowledge, model-oriented programming is the only system that provides such high level of flexibility to programmers.

Bibliography

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 187–197. ACM, 2002.

[ACN+09] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Proceedings of Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009. Available at `http://www.cs.cmu.edu/~aldrich/FOOL09/allen.pdf`.

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 71–80. IEEE, 1994.

[AHL08] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental SAT solver. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5125 of *LNCS*, pages 410–422. Springer, 2008.

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Object Technology*, 8(5):49–84, 2009.

[AYT09] Takahito Aoto, Junichi Yoshida, and Yoshihito Toyama. Proving confluence of term rewriting systems automatically. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, volume 5595 of *LNCS*, pages 93–102. Springer, 2009.

[BA99] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *Transactions on Programming Languages and Systems (TOPLAS)*, 21:813–847, 1999.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. *Communication of the ACM*, 6(1):1–17, 1963.

[BDV10] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing injection attacks with syntax embeddings. *Science of Computer Programming*, 75(7):473–495, 2010.

[Ben86] Jon Louis Bentley. Little languages. *Communication of the ACM*, 29(8):711–721, 1986.

[Béz05] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 143–153. IEEE, 1998.

[BP01] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 31–42. ACM, 2001.

[Bra04] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. Available at `http://bracha.org/pluggableTypesPosition.pdf`.

[Bro87] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[BS02] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 31–40. ACM, 2002.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383. ACM, 2004.

[BVVV05] Martin Bravenboer, Rob Vermaas, Jurgen J. Vinju, and Eelco 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.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 266–277. ACM, 1997.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Cha06] Matt Chapman. Extending JDT to support Java-like languages. Invited Talk at EclipseCon'06, 2006.

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Functional Programming*, 19(5):509–543, 2009.

[CMA94] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report 121, DEC SRC, 1994.

[CR91] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 155–162. ACM, 1991.

[DFS09] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of Haskell Symposium*, pages 93–104. ACM, 2009.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[Dij68] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communication of the ACM*, 11(3):147–148, 1968.

[Dmi04] Sergey Dmitriev. Language oriented programming: The next programming paradigm. Available at `http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf`, 2004.

[DMN67] O. J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 common base language. Technical Report S-22, Norwegian Computing Center, 1967.

[dR03] Daniel de Rauglaudre. Camlp4 reference manual. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`, accessed Oct. 01 2012., 2003.

[EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012. to appear.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.

[EH07a] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–18. ACM, 2007.

[EH07b] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[EKR+11a] Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2011.

[EKR+11b] Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Library-based model-driven software development with SugarJ. In *Companion to Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 17–18. ACM, 2011.

[EO10] Sebastian Erdweg and Klaus Ostermann. Featherweight TeX and parser correctness. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 6563 of *LNCS*, pages 397–416. Springer, 2010.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus 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.

[ERKO12] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 244–263. Springer, 2012.

[ERRO12] Sebastian Erdweg, Felix Rieger, Tillmann Rendel, and Klaus Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of Haskell Symposium*, pages 149–160. ACM, 2012.

[EV06] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[FBF09] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 109–120. ACM, 2009.

[FBLS12] Andrew Forward, Omar Bahy Badreddin, Timothy C. Lethbridge, and Julian Solano. Model-driven rapid prototyping with Umple. *Software Practice and Experience*, 42(7):781–797, 2012.

[FCDF12] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together - compile-time bindings, partial expansion, and definition contexts. *Functional Programming*, 22(2):181–216, 2012.

[FCF+02] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for scheme. *Functional Programming*, 12(2):159–182, 2002.

[Feh11] Stefan Fehrenbach. Retrofitting language-oriented design with SugarJ. Bachelor's Thesis, University of Marburg, November 2011.

[FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: An introduction to programming and computing.* MIT Press, 2001.

[FL08] Andrew Forward and Timothy C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals. In *Proceedings of Workshop on Models in Software Engineering (MiSE)*, pages 27–32. ACM, 2008.

[Fla02] Matthew Flatt. Composable and compilable macros: You want it when? In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 72–83. ACM, 2002.

[Fla12] Matthew Flatt. Creating languages in racket. *Communication of the ACM*, 55(1):48–56, 2012.

[For02] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 36–47. ACM, 2002.

[For04] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 111–122. ACM, 2004.

[Fow05a] M. Fowler. PostIntelliJ. Available at `http://martinfowler.com/bliki/PostIntelliJ.html`, 2005.

[Fow05b] Martin Fowler. Language workbenches: The killer-app for domain specific languages? Available at `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.

[FR07] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of Workshop on Future of Software Engineering (FOSE)*, pages 37–54. ACM, 2007.

[FS06] David Fisher and Olin Shivers. Static analysis for syntax objects. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 111–121. ACM, 2006.

[GHC12] GHC Team. The glorious Glasgow Haskell Compilation System user's guide, version 7.4.1, 2012.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, (3rd Edition)*. Addison-Wesley, 2005.

[GvdA09] Neal Gafter and Peter von der Ahé. Closures for Java. Available at `http://javac.info/closures-v06a.html`, 2009.

[HHPW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Proceedings of Conference on History of Programming Languages (HOPL)*, pages 1–55. ACM, 2007.

[HHS03] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 3–13. ACM, 2003.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Proceedings of European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, volume 5562 of *LNCS*, pages 114–129. Springer, 2009.

[HJSW09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between modelling and Java. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 5969 of *LNCS*, pages 374–383. Springer, 2009.

[HKGV10] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: A case study in transformation modularity. *Software and System Modeling*, 9(3):375–402, 2010.

[HO10] Christian Hofer and Klaus Ostermann. Modular domain-specific language components in Scala. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 83–92. ACM, 2010.

[HORM08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 137–148. ACM, 2008.

[HPvD09] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-specific languages in practice: A user study on the success factors. In *Proceedings of Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5795 of *LNCS*, pages 423–437. Springer, 2009.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142. IEEE, 1998.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[HW09] Daqing Hou and Yuejiao Wang. Analyzing the evolution of user-visible features: A case study with Eclipse. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 479–482. IEEE, 2009.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: A DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 249–254. ACM, 2006.

[JMW10] Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 417–430. ACM, 2010.

[KATS12] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *Transactions on Software Engineering Methodology (TOSEM)*, 21(3):14:1–14:39, 2012.

[KBJV06] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based DSL frameworks. In *Companion to Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 602–616. ACM, 2006.

[KdJNNV09] Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco 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.

[KDRB91] G. Kiczales, J. Des Rivieres, and D.G. Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[Ken02] Stuart Kent. Model driven engineering. In *Proceedings of Conference on Integrated Formal Methods (IFM)*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of Conference on LISP and Functional Programming (LFP)*, pages 151–161. ACM, 1986.

[KKWV12] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Proceedings of Conference on Software Language Engineering (SLE)*. Springer, 2012. to appear.

[Kli93] Paul Klint. A meta-environment for generating programming environments. *Transactions on Software Engineering Methodology (TOSEM)*, 2(2):176–201, 1993.

[KM71] J. Katzenelson and E. Milgrom. A short presentation of the main features of AEPL - An extensible programming language. In *Proceedings of International Symposium on Extensible Languages*, pages 23–25. ACM, 1971.

[KMC12] Tomaz Kosar, Marjan Mernik, and Jeffrey C. Carver. Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments. *Empirical Software Engineering*, 17(3):276–304, 2012.

[KMT12] Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson. An exploratory study of forces and frictions affecting large-scale model-driven development. In *Proceedings of Conference on Model Driven Engineering Languages*

and Systems (MoDELS), volume 7590 of *LNCS*, pages 352–367. Springer, 2012.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[KO10] Karl Klose and Klaus Ostermann. Modular logic metaprogramming. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 484–503. ACM, 2010.

[KOM+10] Tomaz Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Crepinsek, Daniela Carneiro da Cruz, and Pedro Rangel Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, 2010.

[Kri06] Shriram Krishnamurthi. Educational pearl: Automata via macros. *Functional Programming*, 16(3):253–267, 2006.

[KRPGD12] Dimitrios S. Kolovos, Louis Rose, Richard Paige, and Antonio García-Domínguez. The Epsilon book, 2012. Available at `http://www.eclipse.org/epsilon/doc/book/`, accessed Nov. 13, 2012.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular development of textual domain specific languages. In *Proceedings of Conference on Technology of Object-oriented Languages and Systems (TOOLS)*, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: A framework for compositional development of domain specific languages. *Software Tools for Technology Transfer*, 12(5):353–372, 2010.

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: Tool framework for feature-oriented software development. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE, 2009.

[KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

[KV12] Ted Kaminski and Eric Van Wyk. Modular well-definedness analysis for attribute grammars. In *Proceedings of Conference on Software Language Engineering (SLE)*. Springer, 2012. to appear.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain-specific language for source code analysis and manipulation. In *Proceedings of Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177, 2009.

[KVW10] Lennart C. L. Kats, Eelco Visser, and Guido 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.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communication of the ACM*, 9(3):157–166, 1966.

[Lar02] C. Larman. *Applying UML and patterns: An introduction to object-oriented analysis and design and the unified process*. Prentice Hall, second edition, 2002.

[Lay85] Paul J. Layzell. The history of macro processors in programming language extensibility. *The Computer Journal*, 28(1):29–33, 1985.

[Lea66] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9:790–793, 1966.

[LHB01] Roberto Lopez-Herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of Conference on Generative and Component-Based Software Engineering (GCSE)*, volume 2186 of *LNCS*, pages 10–24. Springer, 2001.

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001.

[LR11] David H. Lorenz and Boaz Rosenan. Cedalion: A language for language oriented programming. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 733–752. ACM, 2011.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *Transactions on Software Engineering (TSE)*, 21(9):717–734, 1995.

[LZ74] Barbara Liskov and Stephen N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[Mai07] Geoffrey Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of Haskell Workshop*, pages 73–82. ACM, 2007.

[Mar10] Simon Marlow (editor). Haskell 2010 language report. Available at `http://www.haskell.org/onlinereport/haskell2010`, 2010.

[MB90] Tony Mason and Doug Brown. *Lex & yacc*. O'Reilly, 1990.

[McB04] Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 130–170. Springer, 2004.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communication of the ACM*, 3(4):184–195, 1960.

[McI60] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Communication of the ACM*, 3(4):214–220, 1960.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, 2005.

[MMP10] Ole Lehrmann Madsen and Birger Møller-Pedersen. A unified approach to modeling and programming. In *Proceedings of Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 6394 of *LNCS*, pages 1–15. Springer, 2010.

[MO06] Sean McDirmid and Martin Odersky. The Scala plugin for Eclipse. In *Proceedings of Workshop on Eclipse Technology eXchange (ETX)*, 2006. Published online `http://atlanmod.emn.fr/www/papers/eTX2006/`.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Logic and Algebraic Programming*, 60-61:195–228, 2004.

[MP08] Conor McBride and Ross Paterson. Applicative programming with effects. *Functional Programming*, 18(1):1–13, 2008.

[MQR95] Mark Moriconi, Xiaolei Qian, and Robert A. Riemenschneider. Correct architecture refinement. *Transactions on Software Engineering (TSE)*, 21(4):356–372, 1995.

[MS06] Anders Møller and Michael I. Schwartzbach. *An Introduction to XML and Web Technologies*. Addison-Wesley, 2006.

[MTR05] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of Conference on Compiler Construction (CC)*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.

[Obj04] Object Management Group. Human-usable textual notation (HUTN) specification 1.0. Available at `http://www.omg.org/spec/HUTN`, 2004.

[Ode10] Martin Odersky. The Scala language specification, version 2.9. Available at `http://www.scala-lang.org/docu/files/ScalaReference.pdf`., 2010.

[OGKR11] Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 6813 of *LNCS*, pages 155–178. Springer, 2011.

[Oli09] Bruno C. Oliveira. Modular visitor components. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*, pages 269–293. Springer, 2009.

[Pat01] Ross Paterson. A new notation for arrows. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 229–240. ACM, 2001.

[PF11] Terence Parr and Kathleen 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.

[PP04] Ross Paterson and Simon Peyton Jones. Type and translation rules for arrow notation in GHC, 2004.

[PQ95] Terence Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[PRBA10] Luis Pedro, Matteo Risoldi, Didier Buchs, and Vasco Amaral. Developing domain-specific modeling languages by metamodel semantic enrichment

and composition: A case study. In *Proceedings of Workshop on Domain-Specific Modeling (DSM)*, pages 16:1–16:6. ACM, 2010.

[Pri05] Steffen Priebe. Preprocessing Eden with Template Haskell. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 357–372. Springer, 2005.

[RCM04] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *Transactions on Software Engineering (TSE)*, 30(12):889–903, 2004.

[RDGN10] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Domain-specific program checking. In *Proceedings of Conference on Technology of Object-oriented Languages and Systems (TOOLS)*, volume 6141 of *LNCS*, pages 213–232. Springer, 2010.

[RDN09] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language boxes: Bending the host language with modular language changes. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 5969 of *LNCS*, pages 274–293. Springer, 2009.

[RF12] Jon Rafkind and Matthew Flatt. Honu: Syntactic extension for algebraic notation through enforestation. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 122–131. ACM, 2012.

[RGN10] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding languages without breaking tools. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *LNCS*, pages 380–404. Springer, 2010.

[Rie12] Felix Rieger. A language-independent framework for syntactic extensibility. Bachelor's Thesis, University of Marburg, June 2012.

[RMHP06] Damijan Rebernak, Marjan Mernik, Pedro Rangel Henriques, and Maria João Varanda Pereira. AspectLISA: An aspect-oriented compiler construction system based on attribute grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):37–53, 2006.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[Sch07] Sylvain Schmitz. Conservative ambiguity detection in context-free grammars. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, volume 4596 of *LNCS*, pages 692–703. Springer, 2007.

[SDF+09] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Functional Programming*, 19(Supplement S1):1–301, 2009.

[Sea07] Chris Seaton. A programming language where the syntax and semantics are mutable at runtime. Master's thesis, University of Bristol, 2007.

[SH11] Emma Söderberg and Görel Hedin. Building semantic editors using JastAdd: Tool demonstration. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, pages 1–6. ACM, 2011.

[Shu93] John N. Shutt. Recursive adaptable grammars. Master's thesis, Worcester Polytechnic Institute, 1993.

[SMO04] K. Skalski, M. Moskal, , and P. Olszta. Meta-programming in nemerle. `http://nemerle.org/metaprogramming.pdf`, accessed Oct. 01 2012., 2004.

[SP02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of Haskell Workshop*, pages 1–16. ACM, 2002.

[Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Systems and Software*, 56(1):91–99, 2001.

[Ste99] Guy L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Sun02] Sun Microsystems. Java Pet Store, 2002. Available at `http://www.oracle.com/technetwork/java/index-136650.html`, accessed Nov. 14, 2012.

[SV09] August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210. ACM, 2009.

[TCKI00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In *Proceedings of Workshop on Reflection and Software Engineering*, volume 1826 of *LNCS*, pages 117–133. Springer, 2000.

[The12] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2012.

[THSAC+11] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 132–141. ACM, 2011.

[Tom87] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2):31–46, 1987.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):1–40, 2008.

[VBGK10] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.

[VBT98] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.

[vdBSVV02] Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco 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.

[vdBvDH+01] Mark van den Brand, A. van Deursen, J. Heering, HA De Jong, et al. The ASF+SDF Meta-Environment: A component-based language development environment. In *Proceedings of Conference on Compiler Construction (CC)*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[vDK98] Arie van Deursen and Paul Klint. Little languages: Little maintenance? *Software Maintenance*, 10(2):75–92, 1998.

[vdS11] Tijs van der Storm. The Rascal language workbench. Submitted to Language Workbench Competition 2011, available at `http://oai.cwi.nl/oai/asset/18531/18531D.pdf`., 2011.

[Vis97a] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[Vis97b] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[Vis02] Eelco Visser. Meta-programming with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.

[VKBS07] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute grammar-based language extensions for Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer, 2007.

[Völ10] Markus Völter. Embedded software development with projectional language workbenches. In *Proceedings of Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 6395 of *LNCS*, pages 32–46. Springer, 2010.

[Völ11] Markus Völter. Language and IDE modularization, extension and composition with MPS. In *Pre-proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 395–431, 2011.

[VS10] Markus Völter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. `http://voelter.de/data/pub/VoelterSolomatov_SLE2010_LanguageModularizationAndCompositionLWBs.pdf`, 2010.

[W3C99] W3C HTML Working Group. HTML 4.01 specification. Available at `http://www.w3.org/TR/html4/`, 1999.

[W3C04] W3C XML Schema Working Group. XML Schema part 0: Primer second edition. Available at `http://www.w3.org/TR/xmlschema-0`, 2004.

[W3C08] W3C XML Working Group. Extensible markup language (XML) 1.0 (fifth edition). Available at `http://www.w3.org/TR/xml`, 2008.

[War95] M. P. Ward. Language-oriented programming. *Software – Concepts and Tools*, 15:147–161, 1995.

[WC93] Daniel Weise and Roger F. Crew. Programmable syntax macros. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 156–165. ACM, 1993.

[Weg70] Ben Wegbreit. Studies in extensible programming languages. Technical Report ESD-TR-70-297, Harvard University, Cambridge, Massachusetts, 1970.

[WHG+09] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. Improving domain-specific language reuse with software product line techniques. *IEEE Software*, 26(4):47–53, 2009.

[WS07] Eric Van Wyk and August Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 63–72. ACM, 2007.

[Xte12] Xtext 2.3 documentation. `http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf`, 2012.