Managing Interlingual References – a Type-Generic Approach

Dissertation
zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von
Sebastian Menge

Dortmund
2011

Tag der mündlichen Prüfung: 11. Juli 2011
Dekan: Prof. Dr. Gabriele Kern-Isberner
Gutachter: Prof. Dr. Ernst-Erich Doberkat
Prof. Dr. Jan Jürjens

Preface

> Es gibt nichts Praktischeres als eine gute Theorie.
>
> *(Kurt Lewin)*

This work is the result of several years of research in the field of "software technology". The field ranges from foundational research in the realm of specification of software and related processes using abstract formal methods, to the very practical and pragmatic analysis of modern software engineering. Because researchers want to focus on single specific problems, there is a trend to concentrate on formal methods in research – otherwise the complexity of the modern software engineering context might distract too much from the core problems. On the other hand, the young research assistant (at least me) wants to develop something "useful", something that solves a problem in practical software engineering. In this field of tension this work evolved.

The general goal of my research in this context was to *apply* formal methods such that they solve real-world problems. This is a challenge, because works that develop formal methods *have to abstract* from many constraints and peculiarities that do exist when developing software. On the other hand, works that have a purely practical goal, often solve their problems *only* under certain constraints and a specific context. As a consequence, applying formal methods means to understand both the problem on the practical side and suitable formal methods deep enough to *transfer* the abstract solution to a specific context.

For this thesis, I found my playing field in Haskell, a practical functional programming language that stems from academia and has many formal ingredients. Using Haskell (and its underlying formal paradigms), this thesis develops a framework for *static analysis* of software that is developed using *multiple languages*, e.g. Java and XML.

In real-life projects there are often references between source-code files: For example, XML-files might reference certain Java classes, methods and fields. These references have to be managed somehow.

The problem I solved in this work is to find an extensible approach that allows to manage such references between files of different languages. The extensibility by languages was the core challenge, especially in the strictly typed world of Haskell. Based on advanced theoretical concepts, the approach of *datatype generic programming* was implemented to write programs that can deal with multiple languages generically. So finally, I could take this (existing) approach and apply the theory to develop a prototypical solution for the sake of a useful static analysis solution.

I would like to thank a number of people that supported me during this work. First

of all I thank Prof. Dr. E.-E. Doberkat for all the patience and his good questions that often directed me on my way. I also thank the secondary reviewer, Prof. Dr. J. Jürjens for his kind engagement, Prof. Dr. P. Padawitz for valuable comments and taking over the chair of the committee and Dr. H. Falk for his good feedback and taking part in the committee. The colleagues from our research group were always helpful: thanks to Christoph, Ingo and Jan and the members of the "Haskell-Stammtisch" for fruitful discussions and thanks to Stefan and Doris for encouraging words when they were needed (and omitting them when they were not wanted). I also thank Pascal Hof and Tristan Skudlik who helped me with the implementation. Pascal implemented parts of the graphical frontend (cf. Figure 5.1) of the prototype while Tristan tried to provide a suitable Java-Parser. I have to point out that neither of them took part in the actual research work of this thesis as I am not concerned with graphical user interfaces or implementing parsers.

Last but not least I thank my family, especially Annette, for encouraging me to go on when I nearly lost the confidence to ever make it.

Before going on to the main part of the thesis, I supply a small reading guide: The chapters follow generally the same structure, namely introduction, elaboration, conclusion and bibliographic notes. I assume a certain knowledge in formal specification of software structures, especially using algebraic specifications and a basic knowledge of Haskell (though I will employ advanced concepts in the later chapters). I introduce them as deep as necessary and feasible. For a deeper discussion of such concepts, the reader is delegated to the existing literature. Listings, especially Haskell code, are given inline or in a paragraph (similar to the literate programming style often found in Haskell research papers). I always try to explain the listings in detail in the surrounding text, so I hope all important parts can be understood without understanding the source code in deep detail. Finally, some figures make use of colors and although everything is explained in the text a color-printout will make things clearer.

So here we go ...

Contents

List of Figures

1. Introduction

1.1. Motivation

1.1.1. Context

Traditionally, a project is defined as a temporary undertaking with a clear goal and under certain resource constraints. In software engineering, a *software project* also comprises the software to be produced. Colloquially, the software under development is called *the project* as reflected by the nomenclature of integrated development environments or revision control tools.

We focus on the *data* that constitutes a software project, which is defined, created and manipulated by software engineers. This involves not only the implementation of source code, but also the development of requirement and design documents, user documentation, run-time configurations, test cases, graphics and the like. While a part of the data resides in local files and is managed via revision control systems, other parts may be maintained centrally on web-servers or in databases.

Internally, each kind of data is structured. Source code follows the syntax of the employed programming language, design models have a graphical syntax, documentation is typically written in some markup language, run-time configuration is often given as structured key/value pairs and databases have a database schema.

Externally, the data that constitutes a software project is also structured, namely according to some *project layout*. This means local files are organized in a fixed directory structure and remote resources are organized such that they are easy to remember and to access.

The complexity of these structures makes up a big part of the complexity of software engineering in general.

1.1.2. Interlingual References

Each kind of data serves a special purpose, and is structured to serve that purpose easily. In a sense the whole software (project) is decomposed to smaller specialized parts that can be expressed more concisely: there are special languages for different layers like databases, business processes and user interfaces. While this is fine when concentrating on a specific layer, accessing one layer from another often leads to an undesirable mixture of languages.

For example, in common template engines for the web, pure presentation oriented HTML templates typically have variables which ought to be filled by a Java program. The data, especially language dependent text is stored in special dictionary-like files.

Configuration of such an application might be given in XML or also in key/value dictionaries. The core program, which might be written in Java, then reads all the files from the file system, fills the templates with localized text, and renders the page.

An artificial example is show in Listings 1.1 to 1.5 on page 3. We arranged the example such that the three layers for data, program code and presentation are visually recognizable. The configuration layer is typically orthogonal to the other layers.

We will not explain the code in detail, but focus on the references introduced by the string literals in the different layers and their languages. The references between the code samples in the listings are highlighted by red, number-labelled arrows. The keys `tpl` and `lng` in the configuration on the left (Listing 1.1) denote the file names of the template and dictionary that should be used. In the Java program code (Listing 1.3), these keys are referenced as simple string literals (cf. reference 1) . In the template (Listing 1.2) we have introduced variables `pass, user, secret` and `name`. These are also referenced as string literals in the Java code (cf. reference 2). Finally, we have two language dictionaries (Listings 1.4 and 1.5) whose keys are again referenced as string literals in the Java code (cf. reference 3). Of course, there are also references between the keys in the dictionaries (cf. reference 4).

Thus we have a number of different kinds of entities, like file names, configuration keys, language keys, and variables spread over the whole application. The single layers are quite concise, but we must access the different layers by use of plain strings. This example is very simple: we can extend it easily from simple string literals to identifiers, for example, the configuration could name the main class to invoke, or a number of test-methods.

We conclude the example with the observation that we have no obvious option to check the consistency of these different entities which span over the whole project, since this would involve the analysis of code given in many different languages.

The visualization of the references in the listings gives a first impression of the additional information that has to be managed and how it spans over layers given in different languages. The example also shows that the number of references in such a small example is already quite large.

Using a single language we have references too. Variables and functions have to be defined and used consistently throughout a program. We could have drawn references for all the occurrences of the names `t` and `d` in Listing 1.3, but since the correct use of such symbols is managed by the compiler, we do not care about it here. The management of this kind of references is studied extensively in the field of compiler construction.

In contrast, in an interlingual setting (as common today) there is little support to check consistency *across the different languages*. In the example, we had three languages: Java files, augmented HTML-templates and key-value dictionaries. If it is worth the effort, one could develop a dedicated tool to check the consistency for these three languages and this context of reoccurring variables and look-up-keys.

But in general there are many languages specialized for different tasks: database query languages, markup languages for presentation, configurations for each and every tool and employed framework. While these languages serve their special purpose well, the integration of so many languages is a real problem, sometimes referred to as *language*

```
! a configuration that
! spans many lines to
! show that it is
! orthogonal to the
! other layers.



! the template file
tpl = login.tpl


! the language file
lng = de.dic
```

Listing 1.1: config.properties

```
<html>
...
$pass$: $secret$
$user$: $name$
...
</html>
```

Listing 1.2: login.tpl

```
...
// create a  properties object from
// the file ‘‘config.properties’’
Properties p =
  createConfig("config.properties");

// create a template object from
// the file given in the configuration
Template   t =
  new Template(p.getProperty("tpl"));

// create a dictionary object from
// the file given in the configuration
Dictionary d =
  new Dictionary(p.getProperty("lng"));

// replace variables in the template
// with values from the dictionary
t.replace("pass",d.get("pass"));
t.replace("secret",user.getSecret());
t.replace("user",d.get("user"));
t.replace("name",user.getName());

// finally render the page
t.render();
...
```

Listing 1.3: example.java

```
pass     Password
user     User
```

Listing 1.4: en.dic

```
pass     Passwort
user     Benutzer
```

Listing 1.5: de.dic

cacophony [Fow07]. Developing dedicated supporting tools for each application context and combination of languages from scratch is thus not feasible. We want a solution that can be *reused* in many contexts.

How can we subsume this additional complexity that is imposed on software projects? This additional structure is neither internal because it deals with multiple languages nor is it external, since the references are between entities that appear inside the files.

We call this additional structure *interlingual references* and we claim, that these references have to be managed well in a software project just like the *intralingual references* that are analyzed and managed in compilers. If the interlingual references get inconsistent, the software might not compile, produce wrong results or even crash at runtime.

The management is especially important for large projects: First, there are many languages involved and the sheer number of files can be quite big, so there is potential for many different implicit interlingual references between these files. Second, developers might try to keep these references in mind, but large projects last longer, the team changes, and developers might forget things over time or leave the team taking their knowledge with them.

On the other hand, the effects of mismanagement are obvious: more complexity means longer development times and more errors – a typical maintenance problem. Therefore, it is not astonishing that there are approaches and solutions to the management of interlingual references for some widespread cases.

1.1.3. Approaches

The technical project leader is responsible for the inherent complexity of a software project. By choosing the right programming languages, libraries, frameworks and tools and striving for clean and lean interfaces, one tries to keep the complexity low. In the terminology of above, one is responsible for the overall structure of a project. This includes internal structure (choosing the right languages), external structure (defining the project's spatial layout) and implicit references. To manage the latter we can identify different approaches:

Most important is the choice of libraries and frameworks that minimize the usage of implicit references on their own. A good example for this is the introduction of Java's Annotation mechanism that allows to configure and adapt Java-Applications *in place* instead of splitting off the configuration into separate files, which often introduces many implicit references between program code and configuration. Libraries and frameworks that use this features introduce fewer references into a project.

There are also tool-based solutions. Instead of introducing and handling references manually, a tool might generate references automatically. One example is the plugin registry in the popular integrated development environment Eclipse[IBMC06]. Each plugin has an XML-configuration that maps the plugin's Java classes to so-called extension points of the Eclipse platform. To edit this complex configuration, Eclipse provides a structure editor for the underlying XML configuration. Thus one can choose classes and extension points from auto-generated lists. Following the tool-based approach, developers do not have to deal directly with the references. They can use higher-level software

engineering tools to manage them.

The third approach to manage references is rule-based: Developers often agree on a house style of informal rules and conventions. On the one hand there are operational rules: E.g. *Whenever the name of a database column changes, you have to change all related SQL-statements in the program code.* On the other hand there are simple naming schemes: e.g. names of constants are often written in capital letters. Another example might be naming schemes like the popular Hungarian Notation[Sim99] to encode additional information about the type into the name of a variable which is sometimes useful in dynamically typed languages. Given such a scheme, the understanding of names in the program code will be easier for developers. Following the rule-based approach, developers have to remember or write down all the rules and schemes, there is typically no tool support.

When done properly, the first approach minimizes the problem, but it does not solve it. The tool-based and the rule-based approaches are also not optimal. The tool-based approach is too strict, because it forces developers to manage the projects as prescribed by the tool vendors. It also makes it hard to make custom changes to the software that violate the tool's structure or to even abandon such a tool. Further, the tools themselves have to be configured and integrated, which can introduce new complexities. In contrast, the rule-based approach is too loose, because it is very easy to break the rules, especially in large projects with many developers. There is no automation that checks whether rules are violated.

In this thesis we are developing a new approach to the management of implicit references which is a blend of the tool- and the rule-based approach: It is as expressive as the informal rules and conventions and at the same time tool-driven to allow for automation.

Further we want to work out Brook's *essential complexity* [Bro87] of this ubiquitous software engineering problem: While it is obvious to decompose the overall complexity of a software project into different parts expressed in specialized languages, it is hard to manage the interlingual references that somehow reintegrate the parts to the complete program later on. In a way, all solutions presented above are accidental in Brook's sense as they work only in a specific context: They help when developing a GUI for a Java program, extracting Java classes from a SQL-database schema, writing down rules and conventions for project x or programming language y. But they do not help with the reintegration problem in itself. In contrast, we are seeking for a solution that can be *reused* for many languages and in many contexts.

We envision a solution in a similar spirit as popular software engineering tools which tackle essential problems like revision control (`rcs, cvs, svn` and descendants) or automated builds of software (`make` and descendants). These tools are widely independent of the accidental contexts of a specific project, such as operating systems, programming languages and libraries. Our solution shall enable developers to create, edit and check these references. For now, the creation has to occur manually, but once a sustainable representation of references is available, an automated analysis could generate the references. The checking for consistency of references is easy to automate as we will see shortly.

We want to make the implicit references as described in Section 1.1.2 *explicit* and

we want to exploit this explicitly in a reusable way to prevent the typical inconsistency problems of large projects in the long term.

1.2. Problem Description

1.2.1. Introduction

To manage interlingual references, we need to find a suitable specification language. Using this language we have to

1. define interlingual references,
2. specify language- and context-dependent consistency and
3. adapt references when the software evolves.

The language and the specification should be accessible for implementation. We have to be able to formulate data structures and algorithms that work *across* the different languages that are used in a project. This means that we need suitable data structures to represent all the data that constitutes a project and we want these data structures to be formulated in a uniform way.

Using this specification language, we want to analyze a software project *statically* and mostly *syntactically*. We are not interested in dynamic aspects such as control or data flow issues of a program at compile- or run-time. The problems we investigate belong to the area of static semantics, but since we will focus on *interlingual* aspects, we are not yet in the position to make elaborated analyses as common in the case for one single language.

1.2.2. Problems

Defining interlingual references

We need a suitable representation for the project data we work with. For now we will simply think of *abstract syntax trees* (ASTs), i.e. parse trees that abstract from concrete tokens like parentheses or semicolons. Each inner node of an AST reflects a syntactic construct in the source code, such as an expression or a statement.

Our first problem is to record the implicit references between two arbitrary subtrees, such that they can be used for further analysis. We can imagine a *reference* as a link between two nodes of abstract syntax trees of *possibly different* languages.

As an example, consider Figure 1.1, which depicts excerpts from the ASTs for the examples of Listings 1.2 and 1.3 on page 3. The red, dotted arrows represent two interlingual references. The first one is the already explained reference between the variable “name” and the string literal “name”. The second is an example for a reference between a template variable and a (nested) method call. Note that it is not clear under what circumstances the latter reference is consistent.

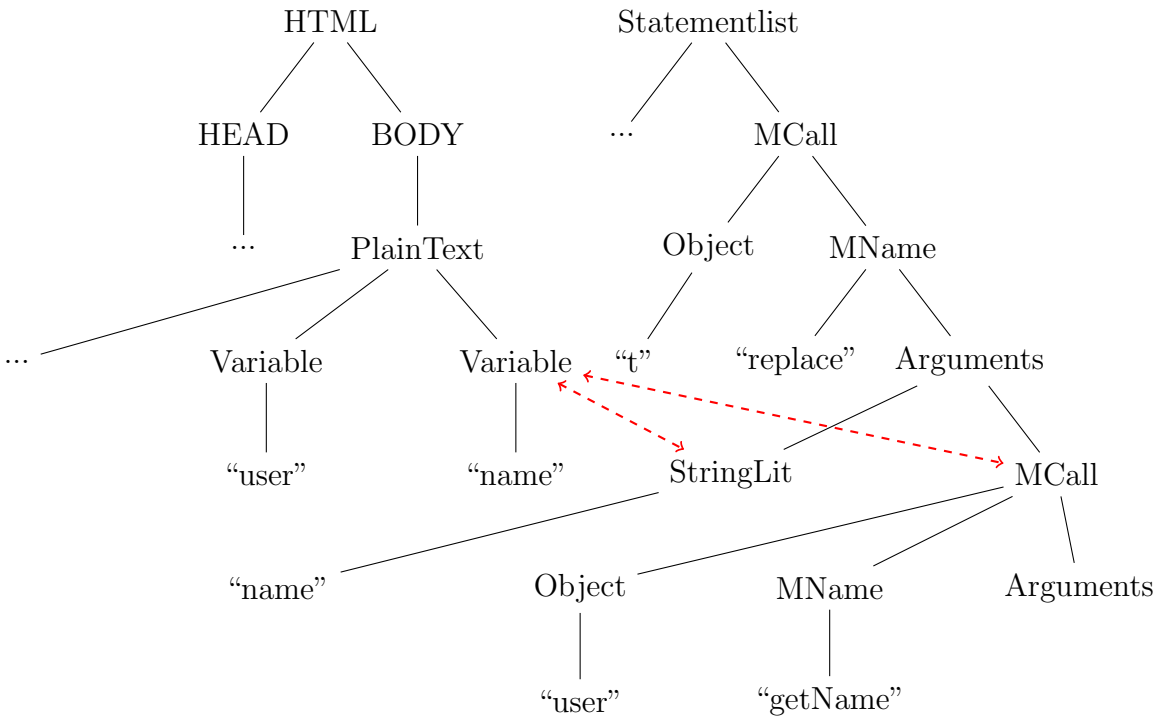


Figure 1.1.: References between ASTs

Given two abstract syntax trees t_1 and t_2 in different languages, we need a way to address nodes. We can use the tree's structure and take paths from the root to the respective node as the addresses. Then we can represent a specific node in t_1 as $a@t_1$. Here @ denotes an infix function that maps an address a and a tree t_1 to the respective subtree. Note that such an address might point to *null*, if there is no node at that address.

A pair of two nodes addressed in this way can then represent an arbitrary reference:

$$\langle a@t_1, b@t_2 \rangle$$

Defining Consistency

Our second problem is to specify *how* two subtrees might be related.

In the simplest form both subtrees represent string constants, e.g. variable names or string literals. Then the most basic relation would be some kind of equality: both nodes represent the *same* string. But notice, that the subtrees for the *variable* `name` in the HTML-AST and the *string literal* `name` in the Java scope are different.

Continuing with the example, we might want to define that an HTML variable `name` and a Java method call `getName()` are consistent in respect to the common substring `name`. Then we would need something more than simple equality: We need a function that takes an HTML *variable* and a Java *method call*, analyzes them by investigating the common substrings of the used strings, and returns true or false.

This approach can be extended to arbitrary subtrees: The developers have to supply functions that take two subtrees and decide whether they are consistent or not. We call such functions *consistency functions*. The definition of these functions depends on the languages and the concrete context of the project. Here, the semantics come into play: By the definition of specific consistency functions, we specify what consistency for two nodes of possibly different languages means. Notice, that this meaning might change from project to project: A string literal can mean a lot of things in the scope of a specific project.

To represent a reference with a consistency function, we simply extend the pair of two addresses as introduced above by a consistency function f to obtain a triple $\langle a@t_1, b@t_2, f\rangle$. Then we can (automatically) check whether two subtrees given by addresses are consistent by applying f to the respective subtrees.

Adapting references over transformations

Source code evolves during development. The changes made to source code, i.e. insertion of new code, renaming of existing identifiers and deletion of code, can make consistent references inconsistent: We distinguish different cases:

1. Rename or change of code which is directly addressed by a reference. The existing consistency function can be readily applied to the addressed subtrees, but it might turn out, that the reference is now *inconsistent* after the change.

2. Deletion of code.

   a) If the deleted code does not affect the path to the addressed subtree, the reference can still be checked.

   b) If the deleted code affects the path, the address points to a wrong subtree now, thus the reference is *invalid*.

3. Insertion of code. Analogous to deletion: If the path to the address is changed, the reference becomes invalid, otherwise it can be checked for consistency as before.

When the code evolves, the addresses have to be adapted: For example, if three nodes on a path are deleted, the path has to be shortened by the three respective positions. Then the address is valid again and references containing this address can be checked as before. To specify a function that adapts addresses, we have to investigate the transformation the code undergoes closely.

Thus the third problem, adapting addresses, breaks down into two parts: Firstly, a description of transformation functions that are applied to syntax trees when the source code changes. And secondly, given a term t, an adaption function d that takes an address a and a transformation function $x : t \to t'$ to a new address a' such that:

$$a@t = a'@x(t)$$

1.2.3. A Type-Generic Specification Language

Until now, we have talked loosely about abstract syntax trees. When it comes to a formal description or even an implementation, one can regard each node of an AST as a function or an operator that is constructed from the node's children. Then, each node has a *type* and each tree represents a typed *term*. This point of view will give us the formal, algebraic underpinning of this work. At the same time, it yields the fundamental problem:

We need to specify functions, like the addressing function @ or the consistency functions, that have to work for multiple types but depend on the types' structure. Because these functions have to change their behavior based on the structure of the type itself, they must be able to inspect this structure. Such functions are called *datatype generic* (or shorter *generic* if there is no danger to confuse the concept with parametric polymorphism as used in Java Generics).

While parametric polymorphism regards the parameter type as a black box, in datatype generic programming, we can change behaviour of our programs based on the structure of the given datatypes. To put it simple, we can write programs (i.e. functions) that are parametric in a whole language, classical examples are functions for equality and serialization of arbitrarily typed values.

There is a branch of research in the field of programming languages and theory, which investigates implementations and uses of this idea. We will pick out an approach, which is an implementation of ideas that date back to a paper by Wadler from 1990 [Wad90b].

These techniques will allow us to specify solutions to the problems that are not *accidental*. The *generic* definition of the addressing function will work for any type or language specified in a specific manner. Thus *generic programming* in this sense allows us to tackle *essential* software engineering problems — i.e. regardless of the languages under consideration. Thus, the investigation of this technique is also worthy in itself for software engineers, because it may be used in other occasions in software engineering, for example type-safe differencing of source code[LLL09] (i.e. between terms or syntax trees rather than strings) or generic parsing techniques to annotate syntax trees with additional information like source code positions[VSMaJ10].

To put it in one sentence, we want to develop a *type-generic* framework for the specification and management of interlingual references.

1.3. Bibliographic Notes

The problems we described in this chapter are motivated from the point of view of a software engineer, i.e. from a practical perspective.

In the wider software-engineering context, the management of inconsistencies is treated either on a higher level or concentrates on one kind of software artifacts. For example, in [NER01] Nuseibeh et. Al. address inconsistencies in an informal way and propose a process-oriented framework for managing inconsistency. Interestingly, they also have the notion of a repository of consistency checking rules – quite similar to our approach

in chapter 4. In [SE03] Easterbrook and Sabetzadeh model inconsistent views on a graph-based model as fuzzy sets and show how incomplete and inconsistent viewpoints can be merged. The approach is directly based on category theoretic concepts and thus quite abstract. Further, there are two special sections in the *IEEE Transactions on Software Engineering* [GN98, GN99] that discuss the management of inconsistencies from different perspectives. Though there are many interesting approaches to the problem in these articles, they are either quite abstract (e.g. discussing inconsistencies on the level of requirements, which is hard to formalize or implement) or they concentrate on single aspects like weaving "development goals" into the software development process or dealing with conflicts in policy-based distributed systems.

In contrast, we seek for a general and implementable solution and want to stay independent of accidental issues. This approach is following the well-known works of Brooks[Bro87], in which he separates between essential and accidental complexity of software development. In the press, one silver-bullet after another is praised, but if one looks closely at it, these are often solutions to very specialized, i.e. accidental, problems. For example, there are a lot of object-oriented frameworks that simplify complex tasks. But the *integration* of such a class-library into an existing context has a complexity of its own. And the complexity of this integration is seldom taken into account.

This complexity becomes visible when there are multiple languages involved to describe data, programs and configurations. Bentley coined the term *Little Languages* [Ben86] to describe the ubiquitous small languages known from the UNIX environment, e.g. to do text processing, sorting and the like. While the pipes and filters architecture employed in the UNIX world is powerful, the data passed around is generally untyped.

The development of compilers is well-known[ALSU06, App97] for single languages. A core data structure of compilers is the abstract syntax tree from which other structures are derived for analysis, optimization and finally code generation. Crew showed in 1997 that such syntax trees can also serve as a basis to analyze C-programs in other contexts than compilers by presenting a PROLOG-based domain specific language to query ASTs[Cre97]. His interpreter allows to formulate various custom queries against C-source code. Interestingly, he mentions techniques like pattern-matching and higher-order functions which will be of importance in the next chapter, when we present Haskell. He also remarks, that his language is *adaptable to many other kinds of structures* but he does not show how that might work. He rather relies on a specific AST-library for C/C++. While Crew's approach is a good example of a flexible source-analysis tool and might serve as a basis to our problems, it is also a good example for yet another accidental approach rather than a solution to an essential problem, in that case querying complex syntax trees.

Independent of our concrete problem, Fowler presents ideas about integrated development environments that do not deal with plain source code, but *richer data structures* to allow for more flexible ways of programming or even generating code in multilingual contexts[Fow07].

The popular development environment *Eclipse*[IBMC06] takes up similar ideas and represents a workbench that can be heavily customized by plugins similar to the ECMA reference model for development environments[UA91]. The workbench itself delivers

abstract interfaces to file-systems, common actions like compiling, debugging and testing and to various external tools. Due to its openness, Eclipse is the de facto standard in any context in which dedicated development environments like VisualStudio (Microsoft) or XCode (Apple) are not feasible. In a sense it is the essential solution to the problem of a unified interface to software development that can be adapted to the accidental requirements of specific contexts by the use of plugins.

1.4. Outline of this Thesis

In the first part of this thesis we will introduce foundations. First we will review and evaluate three approaches to specify typed terms instead of abstract syntax trees. A central aspect is the specification of the types, namely the signature of a language. Then we refine our findings and look subsequently at technical foundations, that means at ways to implement the theoretical ideas presented before. This will already introduce ways to specify types systematically in a manner that can be used to write programs that are generic in the types they are applied to.

In the second part, we will tackle the main task, specifying and implementing a type-generic framework for references. We will then present a prototypical implementation consisting of a library and a visual editor to manage interlingual references. Finally we will conduct a case study that analyzes a template engine for the web similar to our introductory example.

1.5. Conclusion

Coming from the very concrete observation that *interlingual* references often stay *implicit* due to the lack of tool support, we found that it is hard to find a formal basis that can be used to build such tools for the management of these interlingual references. Once we have a formal basis, i.e. a type-generic specification language, we will be able to define, check and adapt interlingual references.

Part I.

Basics

2. An Overview of Theoretical Foundations

2.1. Introduction

We need a suitable way to formalize what we have called *kinds of data*, *languages* or *types* in the preceding chapter.

In many areas of software engineering one is concerned with the structuring of data, be it the customer's problem domain or the intermediate data that software engineers create and consume when developing their solutions. This means that a good understanding (or theory) of data structures is a very important issue in general. In our context, we consider a software project as structured data, not as executable programs: Our goal is to ease the development of software, not to make arguments about executable programs at run-time. The structured data we work with may be source code, but we can also take diagrams, configuration files or databases into account. The structure of the different kinds of data can be derived from their *syntax*. Only syntactically well-formed data can be processed in the software engineering process. For example, compilers and other tools (e.g. XML-Validators) check the validity of a single kind of data. But to check the consistency of a whole project, especially across different kinds of data, we need to specify the structure of different kinds of data *formally*.

To describe the structure of one kind of data, there are different formalisms available. Grammars consist of a set of production rules and focus on the derivation of valid words of the described language. Grammars for programming languages are typically notated in some form of the Backus-Naur-Form. In the case of markup languages, especially XML, special formats like the DTD or XML Schema exists to describe valid documents. But there are also other descriptions of the structure of data: In the object oriented paradigm, valid object structures that encode data are defined using a class design consisting of classes and their interrelations (associations and inheritance). The common theme is, that we have two levels: The data itself (source code, documents, object structures) and the description or *specification* of the data's structure (BNF grammars, XML schemata, class designs).

But things get tricky, when we want to describe the structure of such specifications generally. This introduces a second level of abstraction. The question is: How should we specify specifications?

This is quite important in our case, since we have to deal with many specifications of the different kinds of data. A good way to specify specifications allows us to

1. investigate specifications themselves systematically and

2. relate different specifications with each other.

The latter is the key to our notion of interlingual references: To describe this kind of references generally, we have to inspect the structure of the language specifications themselves, thus we need a suitable way to specify such specifications.

To manage these two levels of abstraction, namely the specification of data and the specification of such specifications, we need a rigid formal underpinning that abstracts aways from everything unnecessary and is still amenable for implementation. This finally leads us to an *algebraic* viewpoint on data and their specifications.

In this chapter, we will present different formalisms to specify the structure of data and finally choose one of them for the coming chapters. The choice will include theoretical and pragmatic arguments: On the one hand, we need a formal, theoretical basis. On the other hand we want to develop a useful solution and give a working proof of concept. Because we want to stress similarities between the formalisms, it will be helpful to know at least one of them. We do not assume that the reader knows all of them in depth, thus we will only scratch the very surface of each of these topics.

Algebraic specification takes a structural point of view to describe structured data: data may consist of multiple sorts and concrete values are represented as many-sorted terms. Thus a datatype is considered as an algebraic structure. We give a very brief introduction in Section 2.2.

The theory of algebraic specifications and their algebras uses concepts stemming from universal algebra. In Section 2.3, we use **Category Theory** as a lingua franca to express and introduce such concepts. These concepts typically involve the structure between different algebras or classes of algebras with certain universal properties. Thus, category theory is a good candidate to specify algebraic specifications formally.

As we already mentioned, we want to develop a proof of concept, namely a tool that helps in managing the inconsistency problems of software projects. This means we need a programming language that implements the concepts of algebraic specifications and category theory and is yet practical enough for day-to-day use. In Section 2.4 we briefly introduce **Haskell** as our language of choice and stress the similarities to algebraic specifications.

2.2. Algebraic Specification

2.2.1. Signatures

We start with some basic definitions following the presentation of [Wir90]. A many-sorted *signature* $\Sigma = (S, Op)$ consists of a set of sort symbols $S = \{S_1, ..., S_n\}$ and a set of operation symbols $Op = \{o_1, ..., o_m\}$. The set Op is equipped with a mapping $type : Op \to S^* \times S$ where S^* is the set of all strings over S. This means that $type(o)$ of an $o \in Op$ returns a pair consisting of a string of sorts as domains and one sort $S_i \in S$ as range. Thus we have two mappings $d : Op \to S^*$ and $r : Op \to S$. When $d(o) = \varepsilon$, the operation symbol denotes a constant of the range sort.

One standard example is to specify lists of elements of some base sort E for data elements: We specify one sort L for lists of various length and two operations, *empty* : $\varepsilon \to L$ which constructs the empty list and *append* : $L \times E \to L$ which constructs a new list given a list and one element of sort E.

A structure preserving map between signatures, $m : (S, Op) \to (S', Op')$ is given by a pair of mappings $m = (m_S, m_{Op})$ with $m_S : S \to S'$ and $m_{Op} : Op \to Op'$ such that m is compatible with the types. i.e. For any $o \in Op$:

$$type(m_{Op}(o)) = (m_S^*(d(o)), m_S(r(o)))$$

where $m_S^* : S^* \to S'^*$ lifts m_S to strings over S.

2.2.2. Algebras

A signature defines the basic structure of a datatype, but it does not tell anything about the sorts or the operations themselves. Once we assign a set to each sort symbol and a function to each operation symbol, we obtain the notion of a Σ-algebra. A Σ-algebra consists of carrier sets A_i for each sort in S and functions $f_k : A_{S_1} \times A_{S_2} \times ... \times A_{S_k} \to A_{S_r}$ for each operation symbol in Op. Structure preserving maps between Σ-algebras are defined accordingly: A Σ-homomorphism $h : A \to B$ is a family of maps $\{h_S : A_S \to B_S\}$ such that for each operation symbol it holds that

$$h_S(f^A(a_1, a_2, ..., a_n)) = f^B(h_{S_1}(a_1), h_{S_2}(a_2), ...h_{S_n}(a_n))$$

It is clear, that there may be many algebras or "models" for one signature depending on the chosen carrier sets and functions. By fixing an algebra, one can say what is *meant* by a signature. One approach, called the "initial" algebra approach, is to construct the so called *ground term algebra* and use this as the standard semantics of a signature.

2.2.3. Terms

A signature defines the structure of a datatype, thus we can use it to form syntactically correct expressions. Then the operations of the signature can be thought of as "value constructors". We will come back to this in section 2.4. A signature $\Sigma = (S, Op)$ yields the set T_{Σ_s} of all terms of sort s. It consists of all constant operation symbols with range s as atomic expressions and all composed expressions $f(t_1, t_2, ..., t_n)$ where f is an operation symbol with range s and the t_i denote terms of sort s_i in $T_{\Sigma_{s_i}}$.

We can use these sets as carrier sets to form a special Σ-algebra T_{Op}, called the ground term algebra. Given an arbitrary Σ-algebra A, we can define a function $eval_A : T_{Op} \to A$ which assigns to each term a value from A. It turns out [Wir90, 2.2.3], that $eval_A$ is a Σ-homomorphism and that we can define it uniquely for each algebra A. This is the concept of "initiality". A Σ-algebra is called initial if there exists a unique homomorphism from it to every other Σ-algebra.

The basic idea here is, that one can construct a model for the signature, just from the signature it self.

2.2.4. Discussion

These are the very first notions of the theory of algebraic specifications. For our purposes this should suffice and we only mention the more detailed concepts that arise, when the admissible algebras are restricted by imposing formulas or axioms that must hold for all algebras, or when signatures or specifications are extended or parametrised.

We can now relate the vague concepts from chapter 1 to this terminology. The languages or kinds of data we spoke of, can be formalized as signatures in the algebraic sense. For example we can imagine the signature of a programming language which consists of many sorts for identifiers, statements, declarations and the like. The operation symbols would take values of these sorts to construct new values, quite similar to the production rules of the underlying grammar. Similarly we can interpret XML-schemata (i.e. the allowed tags and attributes) as signatures. In this case, even clearer, the sorts would relate to certain tags and attributes, and the operations would define how to construct tags from smaller building blocks.

Now that we have a formalization of the structure of a language, it is clear that specific programs or XML-trees can be formalized as evaluations of ground terms. The specific algebra, e.g. the encoding to bytes or other computer-manageable structures, may be arbitrary, since we know that there is a unique morphism from the initial model to the concrete representation. Thus, to each well-structured chunk of data there is a corresponding ground term and its representation is unique (up to isomorphism).

In this framework signatures and their models can be studied thoroughly. But the notions of algebra and homomorphism suggest that the approach builds upon more general concepts from universal algebra or category theory.

2.3. Category Theory

2.3.1. Introduction

While category theory is a very abstract branch of mathematics, we use it as a language to express certain concepts that are so general that we find it easier to explain the general idea and interpret it in our context, than the other way around. In category theory one takes a black-box point-of-view: One considers *categories* of mathematical *objects* and their interrelations, called *morphisms* or *arrows*. A *category* is a five-tuple $(O, M, dom, trg, \circ)$ consisting of a collection O of objects, a collection M of morphisms, two total functions $dom, trg : M \to O$ and a composition operation $\circ : M \times M \to M$. Further the following laws must hold:

1. Each object o has an identity morphism id_o.

2. The composition operation $\circ$ is associative.

3. The identity morphisms are neutral to composition.

For example, signatures and signature morphisms form the category Sig of all signatures. Σ-Algebras and Σ-homomorphisms form the category of all algebras with signature Σ.

2.3.2. Constructions

A *product* of two objects A and B is an object $A \times B$ together with two projection morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, such that for any C and arrows $f : C \to A$ and $g : C \to B$, there is a unique arrow $m : C \to A \times B$ such that $f = \pi_1 \circ m$ and $g = \pi_2 \circ m$.

Intuitively, a product is a specific pair of objects together with morphisms that allow to recover the first and second component from the pair-object.

Dually, a *sum* of two objects A and B is an object $A + B$ together with two injection arrows $i_1 : A \to A + B$ and $i_2 : B \to A + B$ such that for any object C with $f : A \to C$ and $g : B \to C$ there is a unique arrow $m : A + B \to C$ such that $m \circ i_1 = f$ and $m \circ i_2 = g$.

Intuitively, a sum is an object, that embeds two other objects and allows to recognize (along the injection arrows) what part is considered.

A sum object can be thought of as the disjoint union of two objects. For example in the category Sig we might combine two signatures $\Sigma_1 = (S_1, Op_1)$ and $\Sigma_2 = (S_2, Op_2)$ to $\Sigma_1 + \Sigma_2$ which is obviously the same as $(S_1 + S_2, Op_1 + Op_2)$ in this way.

2.3.3. Functors

In most applications, objects are mathematical structures, like algebras, and morphisms are corresponding structure preserving maps. Transferring concepts from one mathematical domain (category) to another makes the notion of functor important:

A functor $F : C \to D$ is a structure preserving map between two *categories* C and D. It consists of two mappings: F_O maps objects to objects and F_M maps morphisms to morphisms. To preserve structure, F_M must respect identities and composition:

1. $F_M(f \circ g) = F_M(f) \circ F_M(g)$
2. $F_M(id_O) = id_{F_O(O)}$

To ease notation and reading, we will omit the subscripts of the mappings when it is clear from context (arguments or return-type) whether F_M or F_O is meant.

When domain and range of a functor are the same category, one speaks of an *endofunctor*. We give a some examples of endofunctors, which will be of further importance in section 3.2.

Constant and Identity

The identity functor $I : C \to C$ maps each object to itself, and each arrow to itself as well. The constant functor K_X maps each object of C to the object X and each arrow of C to id_X.

Sum and Product

Given two endofunctors F, G on a category which has finite products and sums for all objects. We can define the sum-functor as follows:

$$(F + G)(X) := F(X) + G(X)$$

Then $(F+G)(m)$ for an $m : X \to Y$ has to map from $(F(X)+G(X))$ to $(F(Y)+G(Y))$ which can be represented as either $i_1 \circ F(X)$ or $i_2 \circ G(X)$:

$$(F + G)(m)(X) := \begin{cases} F(m)(X) & \text{if } (F + G)(X) = i_1 \circ F(X) \\ G(m)(X) & \text{if } (F + G)(X) = i_2 \circ G(X) \end{cases}$$

where i_1 and i_2 are the injections into $(F + G)(X)$.

Similarly for products:

$$(F \times G)(X) := F(X) \times G(X)$$

Then $(F \times G)(m)$ for an $m : X \to Y$ has to map from $(F(X) \times G(X))$ to $(F(Y) \times G(Y))$:

$$(F \times G)(m)(X) := (F \times G)(m(X))$$

2.3.4. Initial and Terminal Objects

Given a category C, an object I is called initial, if there is a unique morphism from I to every object of C. When we apply this to the category of Σ-algebras, we obtain exactly the definition of Section 2.2: The ground term algebra is the initial object in the category of Σ-algebras. Here the black-box point-of-view makes life a bit easier: we do not have to define terms to make the concept of initial semantics of Σ-algebras clear. On the other hand, the internals of the initial object are not obvious to imagine for a given category.

Dually, an object T is called terminal, if there is a unique morphism *from* any other object of the category to T.

2.3.5. F-Algebras

When talking about the category of Σ-algebras, we would also like to represent signatures categorically. We do so in terms of endofunctors $F : C \to C$:

When we use the category of sets as the base category, we can use a constant functor K_A to represent the set A as a type without any structure.

Lists with elements of a set E, denoted as $List_E$, are constructed with an operation $cons : K_E \times List_E \to List_E$. An empty list is a constant $empty : List_E$. These operations in the sense of algebraic specifications (see the example in Section 2.2 where a list is specified algebraically) can be combined to a sum-functor

$$List_E := (T + (K_E \times List_E)),$$

a recursively defined endofunctor, which maps to either the empty list represented as the constant functor T (the terminal object), or to the product of an E-value and an existing list. Further, we can express recursive positions in the functor by the identity functor I. Then the signature for lists can be represented by the following functor:

$$F = (T + (K_A \times I))$$

This presentation is independent of representation: Operations like *empty* or *cons* have no name anymore, but every parameter can be recovered using the implicitly available projection and injection morphisms of the involved sums and products.

We see that different signatures can be composed from these base functors. The choice of constructors can be modeled as a sum, a constructor with multiple arguments (like *cons*) is modeled as a product over the functors that in turn model the signatures of the parameter types. Recursive positions are represented by the identity functor, unstructured types (e.g. sets) are modeled as constant functors. With these ingredients, we can model a wide range of signatures as (polynomial and recursive) endofunctors.

In analogy to Σ-algebras, an F-algebra (A, α) is then a morphism $\alpha : F(A) \to A$. This morphism maps structured values of $F(A)$ (e.g. terms) to the carrier A, (e.g. a disjoint sum of carrier sets A_i in the many-sorted case).

Lambeks Lemma[Lam89, Sec.1] says, that endofunctors of complete categories (i.e. a class of categories with a certain universal property) have a least fixed point, namely the initial object of the category. When we compute the fixed point for an endofunctor that models a signature, we get the initial model for the datatype (i.e. the ground terms) for free.

2.3.6. Discussion

At this point, we can define datatypes with categorical means. Especially the last subsection gives a very succinct presentation of datatypes and their interrelations. Using the so-called sums-of-products-view, it is very easy to *compose* complex datatypes from these basic building blocks.

In Section 3.2 in the following chapter, we will show how this view on data structures can be *implemented and used* for functions that are generic in the datatype.

Referring back to section 1.2.3, we now have a *systematic* way to *specify* the datatypes we need in a unified way. Instead of studying complex and fine-grained algebraic specifications, it suffices to understand some basic building blocks like sums and products of types / functors and the fixed point construction of initial models. Following the argumentation of the introduction to this chapter (cf. Section 2.1), the signature functors provide us with a suitable formalization of specification of specifications. In the following we will see how functors are implemented in functional languages.

2.4. Haskell

The implementation of algebraic types dates back to the works of Burstall in the seventies (cf. [HHJW07, Sec. 5]). The definition of functions by pattern matching on the constructors of the types was the logical consequence. Today, the most common and popular general purpose functional programming language is Haskell.

The language is defined in natural language in the Haskell Report[PJ03]. The first sentence of the introduction lists the main features: *Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers.*

While a general introduction to the language can be found in [Bir98] for example, we will now focus on the definition and use of algebraic datatypes in Haskell.

2.4.1. Algebraic Specifications in Haskell

As we saw in section 2.2, we need to define the signature, consisting of sorts and operations. In Haskell this is done with the "data" declaration:

```
data Nat = Zero | Succ Nat
```

Here we introduce one sort for natural numbers together with two *constructors* `Zero` and `Succ`, the first being a constant and the second being an operation that maps one natural number to another. Following the idea of initial semantics, one can use terms built from these constructors to represent natural numbers.

By using pattern matching we can write functions that work on `Nat` values: For each *shape* of a `Nat` value, i.e. each constructor, we define the outcome:

```
fromNat :: Nat → Int
fromNat Zero = 0
fromNat (Succ n) = 1+(fromNat n)
```

Because we defined the function for every possible constructor, it is defined for all values by induction. We will make use of this technique later on for arbitrary signatures.

The other basic technique is to define *polymorphic datatypes* that is, structured types whose "elements" are algebraic datatypes themselves:

```
data List a = Empty | Cons a (List a)
```

Given a type `a`, we specify lists of elements of type `a` by introducing a new polymorphic datatype `List`. Values of type `List a` can be constructed by two constructors, namely a constant `Empty` and an operation which takes an element to append and an existing list to form a new list.

We see that this implements basic techniques of algebraic specification: Haskell comes along with a number of base types like integers or characters. With the ability of defining custom (polymorphic) datatypes, many more practical datatypes are available,

and many of these come predefined in libraries. For example, strings are *not* defined as a base type, but as a list of characters.

Because it is relatively easy to specify custom datatypes and functions on the types and evaluate the functions for specific data, Haskell is sometimes called an *executable specification language*.

2.4.2. Category Hask and Polymorphic Datatypes as Functors

Haskell builds up upon the typed lambda calculus. It is known, that the typed lambda calculus is equivalent to so-called cartesian closed categories (see [BW95, Chapter 6] for example). This means, we can roughly think of Haskell's types as objects and of the functions as morphisms. This category is commonly called *Hask*.

Then one can regard a polymorphic datatype like `List` as a mapping from types to types. Using optional features of the GHC compiler, we can rewrite the definition of the datatypes for `Nat` and `List` with types for the constructors, and so-called *kinds* for the datatypes:

```
data Nat :: ∗ where
  Zero :: Nat
  Succ :: Nat → Nat

data List :: ∗ → ∗ where
  Empty :: List a
  Cons :: a → List a → List a
```

This makes the types of the value constructors explicit, and shows the *kind* of the declared types: `Nat` is a simple type of kind `*`, whereas `List` is a *type constructor* that maps a simple type, namely type variable `a` to another simple type namely `List a`, hence the kind `List :: * -> *`

When such type constructors map types to types, i.e. objects to objects in category Hask, how about the morphisms of Hask? The higher order function `map` is well known: it takes a function which operates on elements of type `a` and lifts it to a function on type `List a`:

```
map :: (a → a) → (List a → List a)
map f Empty = Empty
map f (Cons a as) = Cons (f a) (map f as)
```

Using the feature of pattern matching we define for every possible case what `map f` should return for a list of arbitrary shape.

In summary, we have the basic ingredients of an endofunctor on Hask: We identified the object part F_O of that functor as the type constructor `List :: Hask →Hask`, and the morphism part F_M as the function `map`.

This central idea will lead to interesting consequences, since many concepts from category theory can now be applied to the category Hask. In our case, we will investigate the generic specification and analysis of datatypes with categorical means.

2.4.3. Discussion

We have shown that the definition of algebraic types works in Haskell. We have also shown how Haskell's types and functions can be interpreted as objects and morphism of a category. We have not shown how this works formally as there are many subtle issues (cf. [BW95, Nog07]). The main point is, that Haskell delivers an implementation that roughly resembles many abstract category theoretic ideas, at least those that are relevant for our purposes.

2.5. Bibliographic Notes

The theory of algebraic specifications was an early attempt to the formal treatment of data structures. It emerged in the late seventies with the works of Goguen, Thatcher, Guttag and Zille[GTWW75]. Extensive overviews and in depth treatments can be found in [Wir90] and [EM85].

Category theory as a language to express interrelations between different mathematical fields delivers a unified framework in which also many topics of theoretical computer science can be expressed[BW95]. Especially the specification of datatypes using concepts from universal algebra benefited a lot from these approach. For example the duality between state-based systems and data types was made explicit by the duality between co-algebras and algebras [Rut96].

The first mainstream programming language that made some of these concepts *usable* for the average programmer was Haskell. As a successor to a variety of functional languages like ML[MTM97], Hope[BMS80] and Miranda[BW88] it was designed to serve as a test-bed for various experimental language features[HHJW07]. We choose it as a basis for the upcoming implementation due to the benefits of the built-in algebraic specification approach.

Another manifestation of Haskell's adjacency to category theory is the implementation of the input output system by the use of *monads*. This was a consequences of the rigorous separation between pure, functional computations and computations that allow for side effects. For input and output, Wadler introduced the use of the monads[Wad90a]. While this category theoretic construction was introduced to *model* computations with side-effects by the works of Moggi[Mog91], Wadler *implemented* this concept in Haskell. The use of monads is today ubiquitous in Haskell programs, especially when it comes to parsing, IO, or other state-based computations.

2.6. Conclusion

In this chapter we have introduced the basics of three different specification methods: Algebraic specification as introduced in 2.2 is a technique based on set theory but the same concepts can be expressed with categories with the asset and drawback that individual elements are not mentioned. Category theory (section 2.3) is so general that it is sometimes hard to figure out what the concepts mean in a field at hand, but it still

gives us a commonly known language to describe the needed concepts. Finally, section 2.4 introduced Haskell as a programming language that features some of the theoretical concepts from algebraic specification and category theory. We showed how simple datatypes can be specified and we opened the door for the *systematic* specification and utilization of types as introduced in subsection 2.3.5.

While we have seen how to encode endofunctors in Haskell, it is not clear how this relates to the built-in formalism of data declarations that we use to define algebraic types. Once we have a way to convert between the *generic* functorial representation (Section 2.3.5) of a type and the built-in way using plain data declarations (Section 2.4.1), we will be able to write functions that work for any type that is described using such a functor. This is the topic of the following chapter.

3. Datatype-Generic Programming in Haskell

3.1. Introduction

We assume that the syntax of the languages we want to use is given as the signature of an algebraic datatype, i.e. in Haskell as a set of Haskell data declarations. We further assume that lexers and parsers are available that create Haskell-terms from their representation in concrete syntax. In fact, such parsers are available for a number of main stream languages in the Haskell libraries. Since language specifications are typically publicly available nowadays, interested parties develop such parsers based on the language specification.

One of our problems is to define the addressing function @ as introduced in the problem statement in section 1.2.2 such that it works for many languages but is still type-safe. We will look at this for a number of very simple languages, namely lists, trees and simple expressions.

First we introduce lists and trees (so called rose trees, i.e. each node has a label and a list of children)

```
data []    a = [] | a : [a]

data Tree a = Node a [Tree]
```

To address an element of a list we can specify the index in the list.

To address a node in a tree, we can specify the list of indexes in the respective list of children of a node:

```
listindex :: [a] → Int → a
listindex    (x:_)  1   =  x
listindex    (_:xs) n   =  listindex xs (n-1)
```

For a tree of Strings we could implement addresses as paths:

```
treeindex :: Tree a   →   [Int]   → Tree a
treeindex    n            []     =  n
treeindex    (Node _ cs) (i:is) =  treeindex (listindex cs i) is
```

How could we write a function to address subterms of an aritrary expression?

For recursively defined types it is obvious to write recursive functions that behave differently for the different constructors. The visible evidence for this is typically the use of pattern-matching to define such functions. In fact, this is related to the principle of structural induction over terms.

But for recurring functionality in an interlingual context, such as equality, pretty printing, parsing and also addressing, this is a problem, because we have no exact knowledge about the types and their constructors, which would be needed to define such functions. We can only be sure that we are given a signature consisting of multiple sorts and constructors with different arities.

Another example for recurring but type-specific functionality is the higher-order function `map`: It is obvious how to map a function over polymorphic types such as lists or a trees. But how can we map a function over an arbitrary term, i.e. over all arguments of an arbitrary constructor? When mapping over trees and lists, we know the type of elements. It is fixed albeit polymorphic. On the other hand, mapping over arbitrary terms means that the function that we want to map over a term has to work on possibly multiple types and additionally change behavior depending on the constructor at hand.

We introduce a toy-language for arithmetic integer expressions with variables and let-bindings that we will use as a running example in this and the following chapters:

```
data Expr   =  Const  Int
            |  Add    Expr  Expr
            |  Mul    Expr  Expr
            |  EVar   Char
            |  Let    Decl  Expr

data Decl   =  Char := Expr
            |  Seq    Decl  Decl
            |  None
```

Listing 3.1: Expressions and Declarations

The Haskell data declarations are given in Listing 3.1. An expression can be an integer constant, an addition or a multiplication which take expressions as arguments. An expression can also be a variable specified by one character. In a let expression one or more variables are declared to be used in an expression. Variables are declared by the assignment of an expression to a character. A declaration can also be a sequence from two other declarations. To form lists (or trees) of declarations, a constant `None` is introduced for termination.

This language exhibits a number of features: We have constructors with two, one and zero arguments, the arguments might recurse on the specified types and we have *two* types which mutually recurse on each other: We may nest let expressions and declarations arbitrarily.

An example term showing most of the features might be:

```
Let
  (Seq ('x' := Const 5) ('y' := Const 7))
  (Add (EVar 'x') (EVar 'y'))
```

This binds variable 'x' to 5 and variable 'y' to 7 in an expression that adds both variables.

Most constructors have specific names and are written using standard prefix notation. The only exception is the assignment of an expression to a character, which is notated

in infix form using `:=` as the constructor name.

How could we address specific expressions or declarations inside a term, how could we map a function of type `Expr` $\rightarrow$`Expr` over all arguments of an arbitrary constructor? To do so we have to *represent* the constructors, their arguments and the types *in Haskell*, i.e. we want Haskell values that represent these concepts.

The definition of such functions, which make sense for many types of different structure, is called *datatype generic programming*, or simply *generic programming*, when there is no danger to confuse the concept with generics in the language Java, which is just a form of parametric polymorphism. The main goal is to derive the behavior of these functions automatically from the *structure* of the given type. Typical examples for such functions are equality, pretty printing or traversal of terms which all work in the same way in principle, but depend on the types' structure. Because we need to take the structure of a type into account, we need a more sophisticated approach than parametric polymorphism where the type parameter is only a black box.

To investigate and use the structure of a type programmatically, one needs some form of introspection and reflection for types, which is difficult to implement in statically typed languages, where all the type checking is done by the compiler in the first place. How can we represent a type structure (and then write functions that work for arbitrary types) without knowing the types before but still have the compiler to do the standard type inference?

We will give answers in this chapter: First we show how to *define* a one-sorted, recursive algebraic datatype generically and how to *adapt* existing datatypes to this representation. Then we show how to *write generic functions* for datatypes given in this fashion. Finally we show how to *extend this to many-sorted datatypes with mutual recursion*.

3.2. Definition of Datatypes using Functors

In this section we will take the ideas of the preceding chapter on. To do generic programming we need a more elaborated way to specify types than the standard "data declarations". While these declarations can be seen as the signatures of algebraic datatypes, we have seen in section 2.3.5 that we can can specify arbitrary datatypes using functors. Because the idea of functors is readily available in Haskell, we can investigate the structure of such a functor by pattern-matching and we have the type-introspection we need.

In the following we will see an implementation of the ideas of section 2.3.5 in Haskell. For one-sorted algebras these ideas were already published in 1990 by Wadler [Wad90b] and the first implementation appeared in the language *PolyP*[JJ97].

3.2.1. Preliminaries

We recall the basic concepts from Sections 2.3 and 2.4:

A polymorphic type like `List a` can be seen as some kind of function that takes a type, namely `a`, to another type, `List a`. Then `List` is called the type constructor. Universes of types are called *kinds* in Haskell and the set of all plain Haskell types is denoted by $*$. Now we can say that `List` is of kind $* \to *$, written as `List :: *→ *`. We can further define how to *lift* a function that operates on a type of the domain-kind $*$ of the type constructor `List` to a function that operates on type `List a` (which is also a type of kind $*$, when `a` is bound to a type). This is of course the standard function `map :: (a →b) →(List a →List b)` which is normally written and defined as `map :: (a →b) →List a →List b` due to the right-associativity of $\to$.

We already saw that this allows us to regard Haskell types and functions as a category as introduced in Section 2.3. It gives us the opportunity to *implement* (or rediscover) a lot of category-theoretic ideas in Haskell. For example sums of objects appear as the polymorphic datatype `data Either a b =Left a |Right b`. Thus `Left` and `Right` are just the injection morphisms from Section 2.3.2.

The idea of endofunctors is also directly available in Haskell: The type class `Functor` prescribes that an instance for a given type constructor `f` of kind $* \to *$ has to implement the function `fmap :: (a →b) →f a →f b`.

But this also works the other way around: Given a polymorphic datatype and a function that lifts functions from the base type to the target type, we always have the basic ingredients for an endofunctor on Hask: We map types to types using a polymorphic datatype `F :: *→ *` (like List, Tree, Maybe, IO) and we map functions to functions using `fmap :: (a →b) →(F a →F b)`. It remains to show for the polymorphic type that the functor laws are valid. Then one can think of any such polymorphic datatype as a functor.

This in turn means that we can compose functors: E.g. it is immediately clear how to deal with a tree of lists of maybe-values.

In the following we will show how the theoretical way to specify data types as F-Algebras as introduced in Section 2.3.5 can be implemented in Haskell. The key idea is, that polymorphic datatypes can be seen as functors, or better the other way around: any datatype can be specified as a functor.

3.2.2. Pattern Functors

Given an endofunctor on category Hask `F :: *→ *` we can represent a fixed point of that functor as a datatype of a higher kind: `Fix :: (*→ *) → *`. The type constructor `Fix` takes a functor and returns a type. In Haskell this reads as:

```
data Fix f = In {out :: f (Fix f)}
```

The type constructor `Fix` constructs the sought fixed point from a given functor `f` (as a type variable, it has to be written in lowercase in Haskell). The only value constructor has the signature `In :: f (Fix f) →Fix f`. So to find the fixed point `Fix f`, the functor `f` is applied to the type `Fix f` recursively during type inference. The only constructor `In :: f a →a` is the implementation of the morphism α for the *F-Algebra* $(Fix\ F, \alpha)$

from section 2.3.5. The function `out :: f (Fix f) →Fix f` is just a selector function to get the inverse of `In`.

Using the fixed point representation of a type obtained by applying Fix to a corresponding functor, we can represent recursive types generically. Consider a subset of the type introduced in Listing 3.1 on page 28 in the introduction of this chapter. For now, we only represent integer expressions consisting of one single type `Expr` and three constructors:

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

We first encode the plain recursive type `Expr` into a type constructor for a polymorphic type, the so called *pattern functor*:

```
data PFExpr r = ConstF Int | AddF r r | MulF r r
```

The type variable `r` of the pattern functor takes the place of the type we recurse on, thus the fixed point `Fix PFExpr` will be a type that is isomorphic to the type `Expr`. We will encode functions that witness the isomorphy between the fixed point and the original type later in Section 3.3.

The trick is now that we can express any such pattern functor as *composition* of a fixed set of smaller functors in a unified way.

Based on the introductions in Sections 2.3.5 and 2.4, we encode a constructor with no recursive calls to the base type (a constant functor) as Ka, a constructor with one recursive call to the base type as I (the identity functor), the choice between two constructors by the sum of the functorial representation of these two constructors written as an infix type constructor `:+:`. Finally we write the combination of two fields of a constructor (which are again constructors represented as functors) as the product of them written as infix `:*:`. We also require that the type constructor `:*:` binds stronger than the type constructor `:+:`.

```
data K a       r = K a
data I         r = I r
data (f :+: g) r = L (f r) | R (f r)
data (f :*: g) r = f r :*: g r
```

These data declarations are the base-functors that are applied to some type `r`. Because functors can be composed, we can define a type synonym for functor `PFExpr` from above as

```
type PFExpr' = K Int :+: (I :*: I) :+: (I :*: I)
```

which has still the same shape and is of kind `* -> *`, but abstracts from the accidental constructor names. We can see that an expression can be formed from a constant that gets an `Int` as argument, or from one of two binary operations which take both expressions (represented by the identity functors) as arguments. The constructors can be distinguished by their position in the outer sum, the fields of the constructors can be distinguished by their position in the inner products. This is why this way of specifying types is sometimes called the sum of products view. With this approach, the structure

of the type is visible without any (accidental) overhead and we can access the parts using the constructors of the base functors (i.e. injection morphisms for sums). We will see how this works in the following section.

3.3. Adapting given Types

To ease the presentation, we first define a *type family*[1] PF for such pattern functors, and make the pattern functor for a given type `a` an instance of it at index `a`:

```
type family PF a :: ∗ → ∗
type instance PF Expr = PFExpr'
```

This means that the family PF consists of pattern functors of kind $* \to *$ and the pattern functor for index `Expr` is just the pattern functor `PFExpr'` we developed in the preceding section. Thus we reuse the actual type as an index in this type family. This is why the approach is sometimes called *type-indexed*.

Any member `PF a` of the type family `PF` now *generically* represents a type as a functor whose fixed point should be isomorphic to the actual type `a`.

To witness the isomorphy between the index and the functorial representation of the type, we need to provide functions that translate values back and forth. These functions also serve as the *adaption* between the standard form of values and types and the functorial view because they allow us to convert a value *from* its standard form, treat it generically, and translate it back from the generic form *to* the standard form.

Using the type family of pattern functors we now define a *type class* which defines the signatures for these functions.

```
class Regular a where
  from         :: a → (PF a) a
  to           :: (PF a) a → a
```

The function `from` takes a value of type `a`. The type of the result is obtained by one application of the pattern functor at index `a`, to type `a`. This means that the conversion is *shallow*. It only converts the outermost constructor. Eventual recursive arguments in the domain are still of type `a`. The name of the type class reflects the structure of the representable types. At a recursive position, we can only use the one type we are defining. With regular types we can not represent many-sorted types.

With the type class `Regular`, we can now give the adaptor code that allows to convert values of type `Expr` to values of type `(PF Expr) Expr`:

[1] type families are an extension to Haskell that allows to overload *types* in the same way that type classes allow to overload functions.

```
instance Regular Expr where
  from :: Expr → (PF Expr) Expr
  from (Const i)   = L (K i)
  from (Add e1 e2) = R (L (I e1 :∗: I e2))
  from (Mul e1 e2) = R (R (I e1 :∗: I e2))

  to :: (PF Expr) Expr → Expr
  to (L (K i))                = Const i
  to (R (L (I e1 :∗: I e2))) = Add e1 e2
  to (R (R (I e1 :∗: I e2))) = Mul e1 e2
```

Note again, that this conversion is not recursive but shallow. The function `from` unfolds only one constructor to the corresponding structural representation. The children `e1` and `e2` of the constructor remain in their original form.

In fact, the functions `to` and `from` are our very first examples of generic functions, because we can apply them to any type for which we have defined a pattern functor. If we look closely at it, we can see that these functions are *type indexed*. For each type we would call another function obtained by the overloading of type class `Regular`. We can also see, that the functions make heavy use of pattern-matching: By structural induction, they will work for any Expr-value.

The functor and the witness functions can be derived automatically from a given data declaration using Template Haskell[SPJ02, NRH+08].

3.4. Definition of Generic Functions

In this section we will develop a handful of generic functions based on the functorial view on datatypes described in the preceding section.

The first example is to convert an arbitrary term to a tree of strings in which each node represents the name of the corresponding constructor – thus an untyped form of an AST. For that we first need a way to inspect a constructor to get its name.

The second example is the definition of `fmap` for arbitrary terms, i.e. to apply a function to all direct children of a constructor.

The definition of `fmap` will then be used for the deep traversal of a term, i.e. to apply a function to *all* descendants of a constructor, not only the immediate one.

3.4.1. Building Rose Trees from Terms

Using a representation of constructor names in the generic setting, we can now write our first generic function, namely a function that takes a term and returns a tree of constructor names. The datatype for trees is commonly called rose tree or multiway tree. Each node has a string label and a possibly empty list of children.

The definition of generic functions follows the principle of structural induction. For each possible *shape* of a term we define what to do. Because we have only a fixed set of

base functors, this means we have to specify what to do for constants, identities, sums and products.

Because we need to recurse into deeper levels of a term, we need a common interface to the functor types. This interface is given by the definition of a type class. Then each of the functors is put into the type class, and by structural induction, the function will work for any regular datatype defined in the way explained above.

Given such a type class, one usually defines an interface to standard values by composing the generic function with `from`. The interface to our toTree function looks like this:

```
toTree :: (Regular a, ToTreeF (PF a)) => a → [Tree String]
toTree =  (toTreeF toTree) . from
```

The function takes a "normal" value of type `a` and returns a list of trees of strings by delegating recursively to the function `toTreeF` defined in the type class `ToTreeF`. Note that

1. we require type `a` to be in type class Regular to call `from` on it,
2. the corresponding pattern functor `(PF a)` has to be in the type class `ToTreeF` and
3. we have to pass the interface function `toTree` to the type-class function `toTreeF` because at recursive positions we have only values of type `a` rather than the structural type `(PF a) a`.

The type class `ToTreeF` is used to make sure that the instances can apply the function `toTreeF` recursively to eventual parameters. In some sense it will take the place of the induction hypothesis.

```
class ToTreeF f where
  toTreeF :: (a → [Tree String]) → f a → [Tree String]
```

We look at the signature from right to left: We produce a List of Trees of Strings out of some structural value of type `f a`. We also have to carry around the interface function (which works on plain type `a`).

Now we make each of our set of basic functors an instance of `ToTreeF` and thus also compositions of these:

Constants

We begin with constants:

```
instance (Show a) =>  ToTreeF (K a) where
  toTreeF _ (K a) = [Node (show a) []]
```

For constants, i.e. leaves in the tree we don't need to descent any further, so we only have to require that the constant's value can be converted to a String, i.e. is member of the type class Show.

Sum and Product

```
instance (ToTreeF f, ToTreeF g) => ToTreeF (f :+: g) where
  toTreeF t (L x) = toTreeF t x
  toTreeF t (R y) = toTreeF t y

instance (ToTreeF f, ToTreeF g) => ToTreeF (f :*: g) where
  toTreeF t (x :*: y) = toTreeF t x ++ toTreeF t y
```

A sum models choice, so a value can either be a "left" or a "right" value. In both cases we recurse on the inner structured value. A product models the combination of fields of a constructor, thus we concatenate the results for both arguments to obtain a list. Here we can see how the induction hypothesis works: we require the types of the nested terms (i.e. functorial types) to be in the type class `ToTreeF` so we can apply `toTreeF` to them recursively.

Identity / Recursive Positions

The instance for the identity functor is as follows:

```
instance ToTreeF I where
  toTreeF t (I x) = t x
```

For recursive positions in the term, we have to apply the interface function `toTree` which is passed to this function as `t`. This function works on the *unstructured, plain* value `x`, rather than on a structured value of type `(PF a) a`.

Tagged Constructors

To access the *name* of a constructor, we need to attach additional information to the pattern functor and the resulting generic terms. Therefore we introduce another basic functor, namely one that attaches a *constructor representation* to the structural representation of a value. This new base functor takes the following form:

```
data C c f r =  C f r
```

Structured values `f r` are wrapped into the value constructor `C` on the right hand side. On the left hand side, i.e. the functor/type level, the data declaration is "tagged" with an additional type `c`. This type can be used represent additional information about the wrapped value.

Thus we need to model a *type* for each constructor and way to get a constant string value that stands for this type. The tagging types `c` themselves are only needed on the type level, thus we specify them as "empty" datatypes and provide constant functions that map from these constant types to constant values. This allows us to get back from the type level representation (tagging types `c`) to the value level representation (simple strings) of the constructors. The empty types are defined as:

```
data C_Const
data C_Add
data C_Mul
```

The constant functions are are declared via a type class which may be reused for other types:

```
class Constructor c where
  name :: t c f r → String

instance Constructor C_Const_ where name = const "Const"
instance Constructor C_Add_   where name = const "Add"
instance Constructor C_Mul_   where name = const "Mul"
```

The domain type of the function `name` is quite involved but it matches the type of the constructor-wrapper `data C c f r =C f r` well. A structured value that is tagged with an application of `C` has type `C c f r` where `f r` is the type of the value to wrap, an `c` is the type of the tag. Given a value of that type, the function `name` will return the value level representation of the constructor by investigating the type of `c`.

With this trick, we can now redefine the pattern functor for `Expr` accordingly:

```
type instance PF Expr =
     (C C_Const_ (K Int))
 :+: (C C_Add_ (I :*: I))
 :+: (C C_Mul_ (I :*: I))
```

Then the constructor name of a structured value

```
name (  C (K 4) :: C C_Const_ (K Int) ((PF Expr) Expr)  )
```

evaluates to the string `"Const"`.

We access the constructor name of an arbitrary constant Int-value `(K 4)` here. We wrapped the value into the `C`-type, and supplied the constant type `C_Const_` for the constructor name on the right hand side. We have to make the type explicit to help the type-inferrer, since it cannot know what constructor type this value of `C` is tagged with.

Now we can present the instance of the generic function ToTree for the basic functor `C`:

```
instance (ToTreeF f, Constructor c) => ToTreeF (C c f) where
  toTreeF f c@(C x) = [Node (name c) (toTreeF f x)]
```

In case we meet an explicitly modeled constructor of type `C c f`, we construct a singleton list with one node whose label is the constructor's name and whose children are obtained recursively via `toTreeF`.

Summary

Using library functions to draw trees, we can now plot the tree of an *arbitrary* term, when available in a structured, functorial representation using the generic function `toTree`:

For example, the IO-action

```
putStr ( drawForest ( toTree ( Add (Const 4) (Const 5) )))
```

wil output the following lines to the terminal:

```
Add
|
+- Const
|  |
|  `- 4
|
`- Const
   |
   `- 5
```

3.4.2. Generic `fmap`

The second and more general example is to define generic fmap, i.e. a higher-order function that applies a function argument to all arguments of a constructor.

The type class to serve as "induction hypothesis" is the already mentioned Functor type class:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

For regular datatypes we interpret it as follows: Given a function of type `(a → b)` and a functorial representation of a value `x :: a`, we have to define what should happen to `f x` to obtain a result of type `f b`.

If we do that for all our basic building blocks we can be sure, by composition of functors, that we can apply a function via `fmap` to any value of a pattern functor composed of these building blocks:

Constants

Given a constant, the function shouldn't do anything.

```
instance Functor (K a) where
  fmap f (K a) = K a
```

Sum and Product

A sum models (binary) choice of constructors. Thus when a sum value is examined, it can either be a left *or* a right part. If it is the "left" part of a sum, we return a left part of a sum with the function mapped over the argument, analogously for the right part:

```
instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (R x) = L (fmap f x)
  fmap f (L y) = R (fmap f y)
```

A product models the arguments of a constructor, so we have no choice but must process all of them. This is done by mapping f over both x *and* y, and combining the result using the infix `:*:` constructor for products:

```
instance (functor f, Functor g) => Functor (f :*: g) where
  fmap f (x :*: y) = (fmap f x) :*: (fmap f y)
```

Identity / Recursive Positions

To apply a function to the identity of some value x, we return the identity of the function applied to x:

```
instance Functor I where
  fmap f (I x) = I (f x)
```

Tagging Constructor

Since the Constructor-functor doesn't do anything to the value but attaching a string label to its type, mapping over the constructor value means mapping over its argument `f r`.

```
instance Functor f => Functor (C c f) where
  fmap f (C r) = C (fmap f r)
```

3.4.3. Generic Traversal of Terms

Our third example of a generic function is the generic traversal of arbitrary terms. In the preceding section, we introduced the generic function `fmap` which applies a function f to all *immediate* subtrees of a structured value. For a deep traversal of a term we compose the functions we have seen already:

```
compos :: (Regular a, Functor (PF a)) => (a → a) → a → a
compos f = to . (fmap f) . from
```

Thus we transform a value to its structured form, apply f to all subterms via fmap, and reconvert the result to its standard form.

As an example we increment all constants of a term by 5:

```
inc :: Expr → Expr
inc (Const i) = Const (5+i)
inc x = x

rewrite f :: (Regular a) => (a → a) → a → a
rewrite f = f . compos (rewrite f)
```

We first define a standard function on expressions to increment a constant, then we define generic rewrite for values of pattern functors, which makes a recursive traversal of the term and applies f to all nodes.

3.4.4. Summary

Now we have seen how to define and use generic datatypes using functors and the fixed point combinator. We defined a number of generic functions as finger exercises. Function `toTree` translates a typed term to an untyped tree of strings, function `fmap` makes use of the functor idea and is the basis for generic traversal of typed terms using `compos`.

The main drawback of the approach up to now is, that it works only for single recursive types. In the next section we will extend this approach to more (and mutual recursive) datatypes as it is common for real-life programming languages.

3.5. Mutual Recursive Datatypes

This section extends the generic programming approach to *families of mutually recursive datatypes*. While the toy-language of Section 3.2 consisted of one recursive datatype, real languages like Java have typically far more than 10 types to represent their syntactic elements like expressions, statements, class declarations, field declarations, identifiers, etc.

This means, that we cannot represent such a language with one functor that recurses over one type – we have to represent the whole family.

In [YHLJ09] the authors present an approach to generic programming with fixed points for mutually recursive datatypes. We sketch their approach by presenting the main technical basis, GADTs, and their use to restrict Haskell's base kind * to a family of types. This representation can then be used to define a fixed point combinator for functors that represent the recursive structure of such families.

3.5.1. Generalized Algebraic Datatypes

As a preliminary, we shortly present how to use Generalized Algebraic Datatypes, or GADTs for short, to represent families of datatypes.

When multiple datatypes are involved, the generic functions need to know about the type at which they should operate. This is typically done by giving an explicit type representation as an additional argument to the function. The type representations

for a family of types can be implemented by using GADTs consisting of constants to represent the types in the family.

GADTs are an extension to Haskell's data declarations such that the constructors may evaluate to different types [SPJSV09, PWW04]. For GADTs it is common that the kind signature of the type constructor is given explicitly, and that a type signature for each value constructor is given (one per line). The resulting type of the constructors is then parametrized with types as given by the kind signature. For example:

```
data AST :: * → * where
  ExprRep  ::  AST Expr
  DeclRep  ::  AST Decl
```

This defines an GADT for the types of Listing 3.1 on page 28, namely `Expr` and `Decl`. The family-GADT `AST` has two constants of the corresponding type. Then one can write a function like the following which works on any member of the family.

```
f :: (Show a) => (AST a) → a → String
f ExprRep a = (show a) ++ ": Expr"
f DeclRep a = (show a) ++ ": Decl"
```

This function simply changes behavior depending on the given type representation. In general any GADT – commonly called 'phi' – of kind `(* → *)` will do, so the generic library we use uses this to represent type families or members thereof.

3.5.2. Fixed points for Families of Mutual Recursive Datatypes

Given this representation for families, how can we adapt the fixed point idea from Section 3.2 to the case of a *family of mutual recursive types*? If we form the functors for the types of Listing 3.1, we get:

```
data ExprF e d   =  Const  Int
                 |  Add    e e
                 |  Mul    e e
                 |  EVar   Char
                 |  Let    d e

data DeclF e d   =  Char :=  e
                 |  Seq    d d
                 |  None
```

We can see that we can recurse on two types, namely type `e` for expressions and type `d` for declarations.

Thus, a corresponding fixed point operator would get two functors of kind `(* → *) → *` as its argument and would have to produce one of the two types. While the kind of the original fixed point combinator `Fix` (c.f. 3.2) was

$$(* \to *) \to *,$$

we would now need *two* combinators that each produce one of the two types:

$$Fix_{2,0} :: (* \to * \to *) \to (* \to * \to *) \to *$$

$$Fix_{2,1} :: (* \to * \to *) \to (* \to * \to *) \to *.$$

This can now be generalized to families which consist of n types: We would need n fixed point combinators and each of these would get n functors as arguments which would take again n arguments for the possible recursive positions. Thus, the fixed point operators will change according to the arity of the functors. The problem is then, that we need according building blocks (`I, K, :+: , :*:`) for each arity to represent the types generically. Because this does not scale to languages which have ten to twenty mutual recursive types, we need a uniform way to represent these fixed point operators for varying arities.

The main contribution of Löh et Al. in [YHLJ09] is then this argumentation (and its implementation):

If we had tuples on the kind-level in Haskell, we could write the kind of Fix_n (for n mutual recursive types) as

$$((*^n \to *)^n \to *.$$

Because of the correspondence of exponentiation and function spaces, we could replace kind x^n by $n \to x$, if we had numbers on the kind level. Then we could rewrite the kind to

$$n \to (n \to ((n \to *) \to *)) \to *.$$

Now we could reorder the arguments and finally arrive at a fixed point operator for $n \to *$:

$$Fix_n :: ((n \to *) \to (n \to *)) \to (n \to *)$$

So how can we express this *numeric kind* n in Haskell? There is little choice: In Haskell we have only one base kind, namely $*$. But instead of encoding numbers on the type-level, (which would be possible), we take the simpler approach to restrict kind $*$ to the n different types in the type family by using a family GADT as introduced before.

The constants `ExprRep` and `DeclRep` have different types, namely `AST Expr` and `AST Decl` respectively, but we know that both are member of the GADT `AST`. We can now restrict the allowed types for kind n using quantification over $*$ but allowing only for members of the family `AST`. In a function signature, we can write $\forall$`ix :: *. AST ix`, to restrict the allowed types for index `ix` to only those types that are member of our family `AST`.

Finally we want a better way of presenting functors with these n arguments. Instead of parametrizing over each individual type as we did in the beginning of this Section, we pass the GADT and an index to each functor:

```
data ExprF' (r :: * → *) (ix :: *) =  Const Int
                                   |  Add   (r Expr) (r Expr)
                                   |  Mul   (r Expr) (r Expr)
                                   |  EVar  Char
                                   |  Let   (r Decl) (r Expr)

data DeclF' (r :: * → *) (ix :: *) =  Char := (r Expr)
                                   |  Seq     (r Decl) (r Decl)
                                   |  None
```

Here `r` is a type variable for the family-GADT and `ix` the index therein.

Since these functors now have the same arguments, we can now combine them to *one* pattern functor for the whole family – by selecting the type we want to work on, we can choose which of the individual pattern functors we want to use and how to parametrize it.

```
data ASTPF (r :: * → *) (ix :: *) where
  ExprF :: ExprF' r Expr → ASTPF r Expr
  DeclF :: DeclF' r Decl → ASTPF r Decl
```

Turning back to the fixed point combinator, we can now pass such a type-indexed functor to the adapted fixed point combinator of the kind derived above:

```
HFix :: ((* → *) → (* → *)) → (* → *)
HFix (f :: (*→*)→(*→*)) (ix :: *) =
  HIn (f (HFix f) ix)
```

Thus finally, we can express the generic representation of the types of our mutually recursive family of datatypes `AST` using using simple type synonyms:

```
type Expr' = HFix ASTPF Expr
type Decl' = HFix ASTPF Decl
```

The definition of the building blocks for sums and products follows the same pattern as in Section 3.2 and the definition of generic functions is also done analogously, thus we skip it here and refer to [YHLJ09] for more details.

3.6. Conclusion

In this chapter we have introduced flexible ways to generic programming which are entirely type-safe and simply implement the theoretical viewpoint of terms as values of a functorial representation of recursive datatypes in Haskell as introduced in chapter 2.3.5.

The class of datatypes that can be represented by using the simple fixed point combinator from Section 3.2 is called *regular*. In the real world, most languages make heavy use of different datatypes, e.g. for statements, expressions and the like. Furthermore,

these types are typically mutual recursive. The ideas from Section 3.5 are presented in detail in[YHLJ09]. We will build up on that approach in the following chapter.

While the theoretical concepts as introduced in Chapter 2 are well known, suitable implementations are hard to find. We now made our way from plain functors in Haskell and came via regular datatypes to mutually recursive datatypes. This finally allows to represent real-life languages like Java in this fashion.

3.7. Bibliographic Notes

As mentioned in Section 2.5, the underlying theory of the specification of datatypes using F-algebras dates back to the ADJ-Group[GTWW75]. The first approaches to use this in actual programming were made by Hagino[Hag87] and popularized in the functional programming community by Malcolm[Mal90]. The application to types in functional languages is due to Wadlers "recursive types for free" [Wad90b]. The first successful and often cited implementation was *PolyP*, done by Jeuring and Jansson at Chalmers in 1997 [JJ97]. While PolyP was a language with a dedicated compiler, there were also efforts to include the approach into the main functional programming language, namely Haskell. In [Löh04], Löh presents Generic Haskell as a variant of Haskell that has datatype generic programming built in. Later, the group presents the approach we introduced in Section 3.5 which uses advanced type-level options for an implementation of the concepts that finally allows to represent real-world languages. Finally, Yakushew's PhD-Thesis "Towards getting Generic Programming ready for Prime Time"[Yak09] shows that the concepts can now be *used* to solve real-world problems, e.g. for type-safe diff[LLL09] or generic representations of source selections in parser technologies [VSMaJ10].

A good introduction to the field of Generic Programming in general can be found in the lecture notes by Backhouse et al. [BJJM99].

Other approaches to Generic Programming are more pragmatic and usable but not type-safe: SYB[LP03, LP04, LP05] uses the function `unsafeCoerce :: a →b` to define the basic type cast operation (which is a bit spooky in a statically typed setting). This function (as delivered by the GHC module `Unsafe.Coerce`) allows for example, to convert a value of any type to a value of any other type, including function types. Finally, Template Haskell [SPJ02] is also unsafe and much too powerful because it can even generate incorrect Haskell code.

More overviews over the various approaches and libraries are given in the survey papers [HJL07] and [RJJ+08].

Part II.

Inconsistency Management

4. Generic Inconsistency Management

4.1. Introduction

This chapter develops a solution to the problems described in Section 1.2.2. The previous chapter presented how to specify types and write functions generically. This gives us a type-generic specification language. We know that we have to work with terms of some algebraic datatype, that we can specify these types generically and that we can write generic functions that work for a large class of such types. Now we have to investigate the core problems as introduced in Section 1.2.2, namely the definition of *references* and *consistency* and the *adaption of addresses* over source transformations.

4.2. Paths and Zippers

First, we present a conceptual way to represent the needed addresses that point into terms and a way to implement them using a programming pattern, namely *the zipper*. We show the standard instantiation to one fixed type, then we present a library based on the generic programming paradigm from Section 3.5.

Paths

The central element of a reference as introduced in Section 1.2.2 is the address of a specific position in a term. The only commonality of terms of different types is that they can all be considered as syntax trees. To point at specific subtrees one usually employs paths from the root of the whole tree to the root node of the subtree to address. Figure 4.1 shows the syntax tree of the term $a^2 + 2ab + b^2$ together with the path to the second summand $2ab$.

Such paths can be encoded uniquely as lists of integers in which each element i denotes the $(i-1)$th parameter of a constructor. The list for the path of Figure 4.1 would be `[1,0]`.

Given such paths, how can we look up the subterm residing at that address? In Haskell we could write something like:

```
lookup :: Expr → [Int] → Expr
lookup init path = foldl get init path where
  get :: Expr → Int → Expr
  get (Add e1 e2) 0 = e1
  get (Add e1 e2) 1 = e2
  get (Mul e1 e2) 0 = e1
```

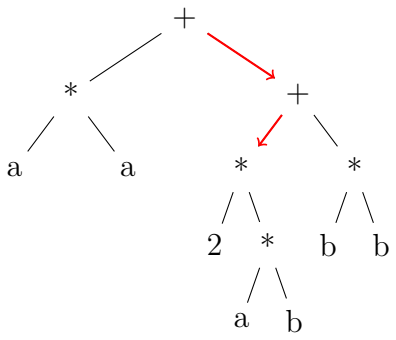


Figure 4.1.: Syntax tree of a term together with a path

```
get (Mul e1 e2) 1 = e2
...
```

This function folds an initial expression `init` from the left, by calling `get` recursively to get the ith parameter of each respective constructor.

But this approach has drawbacks: It would only work for the single type `Expr` and we would have to enumerate all constructors and write a case for each parameter. This means that the function *depends on the structure of the type*. Though the concept is clear: we do a recursive lookup of the i-th child for each constructor, there is no way to write `lookup` in a way that works for arbitrary terms, since we have to name the (type-specific) constructors themselves in the definition of the function. Note that standard mechanisms for reuse would not help here: Parametric polymorphism has only type variables, we cannot look into the structure of such a variable type i.e. the constructors and the number and type of their arguments. Type-classes (ad hoc polymorphism) does not help either, as it only allows to define a common interface to a number of types, thus it can not be used to model the structure of types. Further, if the language has multiple types, the result type of the function `get` would have to change as the types of the parameters of the constructors do.

So this very first naive approach is inconvenient in terms of reuse and we have to look for a better one. The structure of types is reflected in the pattern-functor of generic programming. Therefore we have to look for a programming pattern for the addressing and lookup of arbitrary subterms that can be used in a generic setting.

We will now introduce the well-known pattern for the lookup of subterms by navigation, namely *The Zipper*[Hue97], subsequently we will show how this pattern can be encoded to a *type generic data structure* that is available as a library.

The Zipper

The zipper pattern describes a way of navigating in a treelike data structure (a term) to perform *local* operations on the current focus of the zipper's location. The main motivation for the pattern is efficiency: One splits off the navigation to a subterm from the operations one wants to apply there. In cases of many subsequent operations on deeply nested subterms, the descent has to occur only once. The classic example for this

is a structure editor, which was in fact the context in which Huet used the zipper in the first place.

We choose the zipper not because of efficiency considerations but for design reasons: Since it is formulated as a pattern that can be applied to many algebraic datatypes, it is not surprising that there is an implementation based on the principles of generic programming as introduced in Chapter 3.

First we explain the main idea: Navigating in a term means following the parameters of the constructors while keeping track of focus and context. When we descend into a term, we want to remember the current context, namely the constructor we came from, and the parameter we want to go to. The constructor we came from is modeled as a *context frame*, which can be imagined as a constructor with a hole. This is encoded as a specific datatype with one constructor for each parameter that could be the hole. To navigate back, we can simply *plug in* the current focus into the imaginary hole of the topmost context frame. A navigation path is then a stack of such context frames and together with the current focus it makes up the state of the zipper. Technically, navigation is then the manipulation of the topmost context frame of the stack and the current focus.

As an example we show a very simple zipper for simple expressions. We reuse the Expression datatype from Section 3.2. The datatypes for the zipper are as follows:

```
data Expr = Const Int
          | Add Expr Expr
          | Mul Expr Expr

data Ctx = CConst Int
         | CAddL Expr | CAddR Expr
         | CMulL Expr | CMulR Expr

data Stack = Empty | Push Ctx Stack

data Location = Loc Expr Stack
```

The type for context frames is `Ctx`. For each constructor of the base type `Expr`, there are corresponding constructors for the context frames. If an `Expr`-constructor has multiple recursive calls, there is one `Ctx`-constructor for each of them. The type for stacks of context frames `Stack` is self-explaining. The zipper's state, called `Location` is constructed from the expression at the current focus and the stack of context frames.

The interface to this zipper is given by the following set of functions. We omit the implementation for brevity and show only the signatures.

```
enter :: Expr → Location

down, up, right, left  :: Location → Maybe Location

leave :: Location → Expr
```

The function `enter` injects an expression into a location together with an empty stack.

The navigation functions change the state of the zipper and because they can fail, the result has the type `Maybe Location`. Moving down always directs to the leftmost child of a constructor. The function `up` pops the topmost element off the stack and combines it with the current focus to yield the new focus. It plugs in the focus into the hole of the topmost context frame. The function `leave` is a recursive call on `up` that finally returns the focus itself when the stack is empty. Leaving a zipper can be imagined as *zipping* the stack with the values that result from plugging each current focus into the hole of the corresponding top element of the stack. Hence the name of the pattern.

With all the datatypes and functions at hand, we present an example for the usage of this expression zipper:

```
e :: Expr
e = (Add (Mul (Const 1) (Const 2)) (Const 4))

nav :: Location → Maybe Location
nav = down >=> down

focus :: Expr → (Location → Maybe Location) → Expr
focus e n = focus' where
  Loc focus' stack =  fromJust $ n $ (enter e)

result = focus e nav
```

The definition of function `focus` can be read back to front: To look up the current focus of a zipper, we enter an expression, apply the navigation `n`, deal with the resulting Maybe-value and extract the focus by pattern matching on the resulting Location. For reference, the standard functions `$` and >=> are explained in B.1 and B.5, function `fromJust` in B.4.

Generic Zipper

We now turn to a generic zipper that works for any type that is given generically (cf. 3.5). An implementation is available as a library[YHLJ09, Sec. 5] and ready to use. In the following, we will coarsely explain the types and interface functions for this generic zipper.

The idea is as follows: The family of types that makes up our language is given as an index GADT `phi` and we have the conversion functions `from` and `to`. The problem is, that each node in a term can have one of the multiple types of the family. Whenever we have to assume a specific type, we have to supply a witness argument by using the GADT.

For each context frame we need to remember two types. The type it represents, and the type of its hole. When the frames are stacked, these types have to fit, otherwise one could plug an ill-typed value into such a hole. Thus each stack also represents a value of some type at its bottom, and has a hole of some type at the top.

Context stacks are defined accordingly:

```
data Ctxs :: (∗ → ∗) → ∗ → (∗ → ∗) → ∗ → ∗ where
```

```
Empty :: Ctxs phi a r a
Push :: phi ix
     → Ctx (PF phi) b r ix
     → Ctxs phi ix r a
     → Ctxs phi b r a
```

Stacks are also parametrized over the index-GADT. The constructor `Empty` is a constant, notable is the resulting type: The stack is constrained for a family phi, and top and bottom of the empty stack have the same type. The constructor `Push` takes a witness argument to constrain the type of the hole of the new top-element. Then it takes a context for the pattern functor of family `phi`. This context represents a constructor of type `ix` with a hole of type `b`, and pushes it on a stack which represents a value of type `a` with a hole of type `ix` on top. The result is a stack which still represents a value of type `a` but now with a new hole of type `b` on top.

Now, we need to understand the state of the zipper called `Loc`.

```
data Loc :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Loc :: phi ix → r ix → Ctxs phi ix r a → Loc phi r a
```

We can read this as follows: To construct a generic location, we take a witness of type `phi ix`, a functorial representation of an `ix`-value for the focus (here `r` denotes the recursive position of the value as introduced in Section 3.5), and a stack of contexts. The stack is parametrized over the same family of types `phi`, has a hole of type `ix` in which a functor `r` sat and represents a value of type `a` when the value is left. The result is a location of type `Loc phi r a`, which means that the location represents a value of a type `a` in family `phi` and is currently focused at the recursive position `r`. Note that we cannot recognize the type of the hole `ix` from the type of the resulting location.

Put simply, we constrain the constructor using a witness of type 'phi ix' (i.e. an index in the family GADT), and pair a value with a stack, just as we did in the simple case before. Due to the underlying machinery and the fact that multiple types are involved, we have to take care of all the types to make sure that no ill-typed values are plugged into the holes of the contexts.

The navigation functions look quite similar to the ones we presented before, only with more complicated types due to the type parameters[1].

```
enter :: phi ix   → ix → Loc phi I0 ix
down  :: Loc phi I0 ix → Maybe (Loc phi I0 ix)
up    :: Loc phi I0 ix → Maybe (Loc phi I0 ix)
right :: Loc phi r  ix → Maybe (Loc phi r ix)
left  :: Loc phi r  ix → Maybe (Loc phi r ix)
leave :: Loc phi I0 ix → ix
```

Despite the type variables for the family and the recursive positions, the interface is analogous to the simple case as shown in Section 4.2

[1] here `I0` is the variant of the identity functor that is used to emphasize the usage at kind $*_\phi \to *$ instead of kind $* \to *$. It is defined as `newtype I0 a =I0 unI0 :: a`. For details, we refer to [YHLJ09]

4.3. References

From these preliminaries, we will now derive a way to address subterms in arbitrary terms. This can then be used to specify generic references between terms of different languages.

We start with the combination of the conceptual idea of address-paths as lists of integers and the navigation-operations available through the simple zipper to model references between different, but fixed and regular types. Then we generalize this idea to the case for two families of mutual recursive types.

Simple References

With a little Haskell-juggling of the standard `map` function, folding from the left with `foldl` (cf. Section B.3) and monadic composition using the operator >=> for the Maybe monad (cf. Section B.5), we can simply transform the encoding of paths as integer lists to a composition of navigation-operations for the zipper.

```
tonav :: [Int] → (Location → Maybe Location)
tonav path = foldl (>=>) return (navs path) where
  navs :: [Int] → [Location → Maybe Location])
  navs p = map tonav' p
  tonav' :: Int → (Location → Maybe Location)
  tonav' 0 = down
  tonav' i = tonav' (i-1) >=> right
```

The idea here is, that each number i in an address-path `path` means that we have to descend once, and move right i times in terms of the zipper. This is implemented by the helper function `tonav'`. The list of composed navigations `navs` yielded by mapping `tonav'` over `path` is then folded from the left, with `return` being the neutral element and `>=>` being the composition operator for functions in the Maybe monad.

To ease presentation we additionally introduce a dedicated type-synonym for address-paths and rewrite `tonav` to an infix function `@@` that manages the use of the zipper. This finally resembles the presentation from Section 1.2.2 in the introduction:

```
type Path = [Int]

(@@) :: Path → Expr → Expr
p @@ e = case (tonav p $ enter e) of
  Just (Loc focus stack) → focus
  Nothing → error "invalid path"
```

The following session of the Haskell interpreter `ghci` shows how this function can be used.

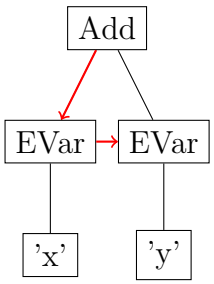


Figure 4.2.: Path `[1]` in expression `e`

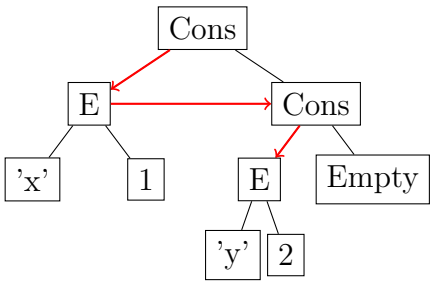


Figure 4.3.: Path `[1,0]` in map `m`

```
ghci> [0] @@ (Add (Const 5) (Const 7))
Const 5
ghci> [1,0] @@ (Add (Const 5) (Const 7))
*** Exception: invalid path
```

The first application of `@@` looks up the first child of the outermost constructor, which is `Const 5`. The second application shows what happens, when an invalid path is given to `@@`. The second child `Const 7` has no more children, thus the application fails since one of the resulting navigations yields `Nothing`.

Now, that we are able to lookup subterms in the case for a single, regular type, we first introduce a second grammar and then show how pairs of addresses can be encoded.

```
data Entry = E Key Value
data Map = Empty | Cons Entry Map

type Key = Char
type Value = Int
```

The new language models simple association lists that assign numbers to characters. In our running example of let-expressions, we used characters as names for variables. Our second grammar also uses characters, and we could use it to define an assignment for variables in expressions for test data. Then expressions could be evaluated for a number of different assignments:

```
e :: Expr
e = Add (EVar 'x') (EVar 'y')

m :: Map
m = Cons (E 'x' 1) $ Cons (E 'y' 2) $ Empty

test :: Expr → Map → Bool
test e m = ...
```

The languages themselves are not aware of each other: Only the context (e.g. interpreting characters as variables) specifies how the languages relate. Of course, problems will occur, when someone decides to rename the variables in expression `e`. That is why we would like to make the references between the various occurrences of `xs` and `ys` ex-

plicit: The path to the variable `y` in `e` is `[1]`, the path to the entry `y` in `m` is `[1,0]`. Figures 4.2 and 4.3 show the resulting zipper navigations of these paths as thick, red arrows.

To make this reference explicit we could define a *nested pair*:

```
(([1],e),([1,0],m)) :: ((Path,Expr),(Path,Map))
```

In the simple case, this pair defines an explicit reference between expression `e` and map `m`. Using the zipper, the referenced subterms could be looked up easily and checked for consistency. Note that we would need an specific zipper for each language, which does not scale for many languages. Note also, that the type `((Path,Expr),(Path,Map))` is once more accidental as it depends directly from the types for expressions and maps.

Generic References

After explaining the basic principles, we now extend the ideas to arbitrary types. The interesting question is: What is the type of the nested pairs for arbitrary languages? While the paths can mainly stay as they are, we will need a generic type for the terms.

As before, we start with the encoding of address-paths to zipper-navigations, now for the generic case. First, we encapsulate the navigation-operations of the generic zipper to a shorter type-synonym `Nav` that is parametrized over a family of types and an index that represents the type of the term under consideration.

```
type Nav phi ix= (Loc phi I0 ix → Maybe (Loc phi I0 ix))
```

Then we can simply adapt the function that converts address-paths to navigations to the generic setting:

```
-- convert paths to navigations
tonav :: Path → Nav phi ix
tonav path = foldl (>=>) return (navs path) where
  navs :: Path → [Nav phi ix]
  navs p = map tonav' p

  tonav' :: Int → Nav phi ix
  tonav' 0 = down
  tonav' i = tonav' (i-1) >=> right
```

The implementation stays exactly the same, only the types change slightly.

To lookup the current focus of a zipper, we use a new function `focus` that extracts the current focus. Because we cannot recognize its type from the type of the location (it could be any member of the family), we have to supply a witness argument again. Using this, we can rewrite the infix function `@@` to the generic case:

```
-- explicit lookup
(@@) :: forall phi ix1 ix2. (Fam phi,
                             Zipper phi (PF phi),
                             EqS phi)
     => (phi ix2,Path)
     -> (phi ix1,ix1)
     -> ix2
(q,path) @@ (p,e) = case (tonav path $ enter p e) of
  (Just l :: (Maybe (Loc phi I0 ix1))) -> focus q l
  Nothing -> error "invalid path"
```

We have made all the types explicit by giving the type representations of type `phi ix` as additional arguments to the function because there is no way to infer them automatically. But since we know that only members of the type family `phi` are possible, we can quantify over these as shown in Section 3.5. Using this approach, we can simply navigate to the subterm indicated by the path, and extract the focus that resides there. In case the navigation fails, we stop the evaluation with an error.

Because this combination of value, path and the associated type representations is hard to read, we introduce a new type, that models an address. It consists of four components: the path and the main value, each together with a type representation as witness. In turn, this can be used for a more convenient look up function.

```
data Address phi s t = Address
  { typerep :: phi s    -- type representation for the value
  , value   ::     s    -- the main value
  , pathrep :: phi t    -- type representation for the subterm
  , path    :: Path     -- the path
  }

lookup :: (Fam phi, Zipper phi (PF phi), EqS phi)
       => Address phi s t -> t
lookup a = (pathrep a, path a) @@ (typerep a, value a)
```

Now we can define a type for the nested pairs from the simple case. The type for generic references consists of two addresses which are parametrized over two distinct families, and all together four type representations for the two values and the two paths.

```
data Reference phi1 s1 t1 phi2 s2 t2 = Reference
  { fst :: Address phi1 s1 t1
  , snd :: Address phi2 s2 t2
  }
```

To summarize the section, we can now write down syntactic references between arbitrary algebraic datatypes explicitly.

4.4. Defining Consistency

As outlined in the introduction, we want to check an explicit reference for consistency. What does consistency mean? In general this is impossible to say for two arbitrary languages, because a specific element of one language might be interpreted differently in others.

The main use case is string literals and identifiers: There are many different possible interpretations for a Java string literal, e.g. it might represent a file-path, a specific table or column in a relational database or a URL. Its interpretation depends highly on the specific context. Thus, we have to leave the decision whether two arbitrary subterms of terms of different languages are consistent *to the user*, i.e. the developer. She has to decide under what conditions two subterms are considered to be consistent in the context of the project.

Technically, this means that one has to provide a function that takes those subterms, compares them somehow, and delivers a result.

Because we have specified the types of the subterms in the references explicitly, the type for the consistency function is obvious. We extend the type for references by a Boolean function for these types:

```
data Reference phi1 s1 t1 phi2 s2 t2 = Reference
  { fst :: Address phi1 s1 t1
  , snd :: Address phi2 s2 t2
  , cmp :: t1 → t2 → Bool
  }
```

The function `cmp` takes two values which are the endpoints of address-paths and produces (for simplicity) a Boolean. The types of these endpoints are known, because we had to give type representations to the addresses in the first place.

Now we can specify a generic function `check` that takes a reference and applies the type-specific consistency function `cmp` to the values of the reference. The function `check` is polymorphic in two families, so we require that the families `phi1` and `phi2` are member of the Zipper type class.

```
check :: (Fam phi1, Zipper phi1 (PF phi1),
          ,Fam phi2, Zipper phi2 (PF phi2))
      => Reference phi1 s1 t1 phi2 s2 t2 → Bool
check ref = (cmp ref) (lookup $ fst ref)  (lookup $ snd ref)
```

We simply look up the first and the second address of the reference, and apply the (type-specific) consistency-function to the resulting values. Note, that we defined the types for references, addresses and check-function *once and for all* because they are parametric in the languages given by the families `phi1` and `phi2`.

To illustrate this, we continue with the example from above. To compare an expression with an entry, we first define the (context-specific) consistency functions.

```
-- simple compare functions
cee :: Expr → Entry → Bool
cee (EVar v) (E k _) = v == k
cee _ _ = False

cem :: Expr → Map → Bool
cem (EVar c) m = DM.member c $ DM.fromList $ toList m
cem (Add e1 e2) m = cem e1 m && cem e2 m
cem (Mul e1 e2) m = cem e1 m && cem e2 m
cem (Const i) _ = True
```

The function `cee` compares expressions and entries, the function `cem` compares expressions with maps. The first one yields *True* only when a variable is compared with an entry and both use the same character. The second one checks recursively, whether all variables of an expression occur in the map.

Then we formulate the addresses and references explicitly and finally call the check function:

```
e1 = Add (EVar ('x')) (EVar ('y'))
m1 = Cons (E 'x' 1) $ Cons (E 'y' 2) $ Empty

a1 :: Address AST Expr Expr
a1 = Address TheExpr e1 TheExpr [1]

a2 :: Address MAP Map Entry
a2 = Address TheMap m1 TheEntry [1,0]

a3 :: Address AST Expr Expr
a3 = Address TheExpr e1 TheExpr []

a4 :: Address MAP Map Map
a4 = Address TheMap m1 TheMap []

ref1 = Reference a1 a2 cee
ref2 = Reference a3 a4 cem

result1 :: Bool
result1 = check ref1

result2 :: Bool
result2 = check ref2
```

The first reference relates the character `'y'` that denotes a variable in `e1` with the character `'y'` that denotes a key in `m1` using the consistency function `cee`. The second reference relates the whole expression with the whole map to check whether all variables are defined using the consistency function `cem`.

Here, `TheExpr,TheMap` and `TheEntry` are the type representations from the family-

GADTs `MAP` and `AST`:

```
data MAP :: * → * where
  TheEntry  ::  MAP Entry
  TheMap  ::  MAP Map

data AST :: * → * where
  TheExpr  ::  AST Expr
  TheDecl  ::  AST Decl
```

4.5. Intermediate Summary

Because the preceding sections solve our core problem of specifying and validating references in a generic way, we give a short intermediate summary.

Firstly, our approach assumes that parsers are available that transform source code to its abstract syntax, which is then available for analysis as a Haskell term. The specification of the abstract syntax itself comes typically along with the parsers as a couple of (mutually recursive) Haskell datatypes. The result of the parser is typed with these types.

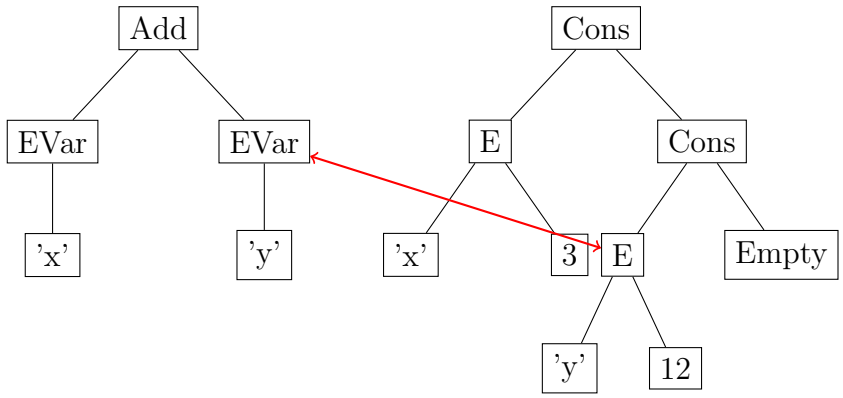
Following the approach presented in Sections 3.2 and 3.5, we can also assume that pattern functors and corresponding adaptors from the given datatypes to their generic representations are available. They can easily be generated by Template-Haskell routines of the underlying generic programming library.

In the preceding sections, we first defined datatypes to make addresses and references explicit in a generic way, based on the multirec library for generic programming by Löh et al. [YHLJ09]. Using the generic zipper presented in that work, we are able to generically lookup subterms specified by addresses given as paths in the syntax tree. Then we showed how to define specific consistency functions, depending on our accidental context. And finally, we defined a function to check explicit, generic references with customizable consistency functions *once and for all*.

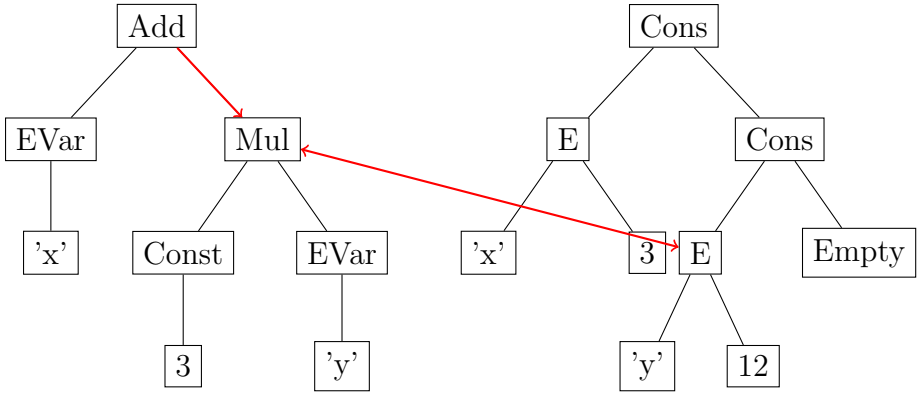
4.6. Adapting along Transformations

Until now, we have considered references between fixed terms. The crux is now that the terms change, as the project evolves, thus the references might become invalid.

The core problem is, that after a structural change of a term, paths pointing to specific subterms might point to another subterm afterwards. This means that the consistency functions we introduced before are not applicable anymore. Instead, we have to change or *adapt the given paths* according to the transformation of the term. Technically this means, we are looking for a function of type `Path` → `Path`. Since we have to know *how to adapt* the paths, we need to supply additional arguments to this function that contain all information about the change that we need to adapt the paths to the addresses of our references. Thus, the signature of the function might look like this:



(a) A reference between variable `EVar 'y'` and entry `E 'y' 12`



(b) A reference with an invalid path

Figure 4.4.: Evolution of two terms and a reference

```
adapt :: Args → Path → Path
```

The additional arguments depend on the actual transformation on the terms, thus we have to derive the arguments from the change itself.

We start with a motivating example. Figure 4.4 depicts the evolution of two terms and a reference between them. In the expression on the left, the variable `y` is replaced by `3*y`. In the result, the path that pointed to variable `y` points to the multiplication as depicted by the red path. Now, the consistency function that compared the entry `E 'y' 12` with the variable `EVar 'y'` will return an invalid result because it is applied to a different subterm, namely `Mul (Const 3) (EVar 'y')` after the change.

While the consistency functions introduced in the preceding section can detect whether the subterms themselves changed, the approach cannot deal with structural changes on the terms.

There are different ways to model changes on terms. Tree- or Graph-Rewriting is often used in compiler-construction to optimize terms or to translate them into another representation. To do so, such systems employ a fixed set of *rewriting rules*. In software development, these techniques can be applied when using a highly structured view on

the software, i.e. in structure-editors or when applying refactorings. But when it comes to implementation on the source-code level, a rule-based approach is infeasible because implementation is often done ad hoc by the developers. We will say more on this topic in the bibliographic notes at the end of this chapter.

Another, quite popular, presentation of changes is used when comparing different revisions of one file. This presentation is text-based. Two files are compared line by line, regardless of the syntax of the underlying language. While this approach is very flexible, and thus widely used, it does not use the inherent syntactic structure of the files. For example, one would like to know that "The name of attribute `x` changed to `y` throughout file `f`." rather than "Lines 3, 5, 8 and 13 were changed, line 15 was deleted and line 17 was inserted.".

For our needs, both approaches are insufficient so we will look for another presentation of changes, which lies between the highly structured and rule-based view of rewriting systems, and the poorly structured textual representation.

We are given two terms t_1 and t_2 which are the results of parsing two different revisions of a piece of code. We assume that both terms are well-typed, because otherwise, the parsers had failed in the first place. How can we infer the *structural changes* that t_1 underwent to become t_2? Our wording *change**s*** shows already, that we have to think of *multiple* changes at various locations in the term. That means, that we assume that any complex transformation on a term can be decomposed into a sequence of atomic operations. In the worst case, the whole term is replaced with another, in the best case, it stays the same. Thus any complex transformation can be modeled as a composition of atomic operations. To be able to compose the operations, we require that they are structure preserving in the sense that the structure of the type is respected. The atomic operations we think of are *insertion* and *deletion* of subterms.

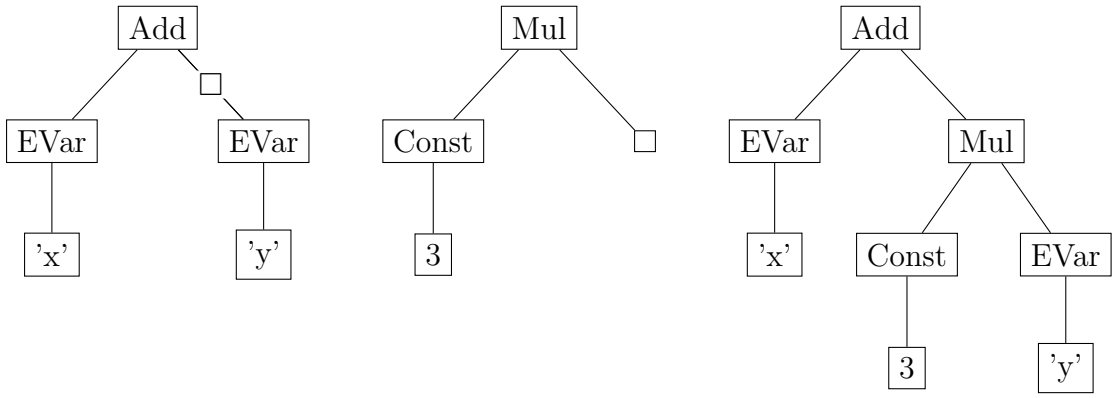
In the following we will describe how these changes can be modelled in our framework. Then we show what is needed to infer the knowledge we need to adapt the paths accordingly. We start with the description of generic insertion of a new subterm into an existing term.

We may only insert subterms of the *right type*, at the *right place* in a given term. The right type is conceptually clear: We may not insert a declaration, where an expression is expected. The right place is more interesting to look at:

A term is closed: there are no open ends or holes in it in which we could insert a subterm. We rather have to create such a hole, by cutting off a subtree r, leaving a term with a typed hole. Then we can insert a subterm i with a corresponding type there. Finally we have to plug the subtree r into i somewhere. This means, that we have to specify a corresponding typed hole in the insertion i too, and that this hole and the cut-off subtree r have to agree on their types.

As an example we consider the introduction of the multiplication by 3 of Figure 4.4. We start with the base term `Add (EVar 'x') (EVar 'y')`. Next, we cut off the subterm `EVar 'y'` at the position marked by the box. Then we insert the multiplication `Mul (Const 3)` $\square$ at that position, and finally plug in the cut-off rest (`EVar 'y'`) into the `Mul`-constructor at the location marked by the box.

We can describe both the relevant locations in the base term and in the insertion



(a) base term (b) insertion (c) result

Figure 4.5.: Insertion

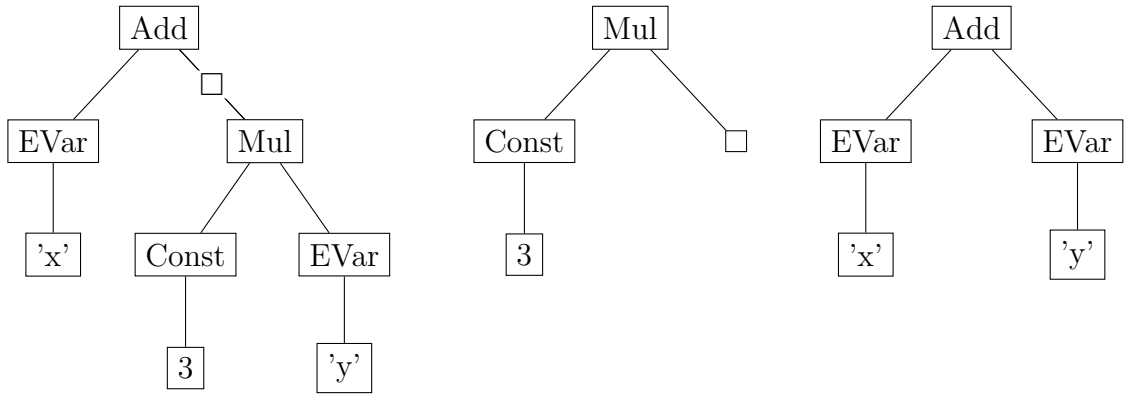
by paths. We want to insert at the right argument of the outermost `Add`. This can be described by the path $p_1 = [1]$. The hole of the insertion is located at the right argument of the constructor `Mul`, namely path $p_2 = [1]$. If we are interested in variable `EVar 'y'`, this is described by path $p_3 = [1]$.

When we insert the multiplication into the term, we have to change p_3 only if p_1 is a prefix of p_3. Otherwise (e.g. if we were interested in `EVar 'x'`) the insertion does not affect the subterm we are interested in. But if an insertion does, we have to insert p_2 just after after the prefix p_1 of p_3. This can be directly translated to Haskell as a simple function on lists of integers.

```
 -- adaption on paths without type checking
insert :: Path → Path → Path → Path
insert p1 p2 p3 = if (p1 `isPrefixOf` p3)
                                  then (p1 ++ p2)++(p3 \\ p1)
                                  else p3
```

Here, `(++)` is concatenation and `(\\)` is the difference on lists (cf. Appendix B.2)

The deletion of a subterm is then just the inverse operation. Consider the example in Figure 4.6. To remove the subterm `Mul (Const 3)` □ from the term on the left, we have to specify where the deletion starts ($p_1 = [1]$) and the location in the deletion and where we want to stop ($p_2 = [1]$). From these paths we can compute the adaption of path $p3 = [1]$ which models the location we are interested in, in our example `EVar 'y'`. Again, we only have to adapt p_3 if the change really affects the path to the place we are interested in.



(a) base term (b) deletion (c) result

Figure 4.6.: Deletion

```
delete :: Path → Path → Path → Path
delete p1 p2 p3 = if ((p1++p2) `isPrefixOf` p3)
                  then p1++(p3 \\ (p1++p2))
                  else p3
```

We can use these two operations and compositions thereof to model changes on terms. The key difference to the other approaches is that we can apply this to typed terms using the generic zipper. Of course, the paths themselves are not typed, but when they are applied, the generic zipper as introduced in [YHLJ09] will return an error if the types don't match.

Since a zipper represents terms without loosing information, we could even replace paths by the stacks of corresponding zippers, since they also have the notion of terms with (typed) holes.

Because we can represent changes in this way, it has to inferred from two snapshots of an evolving term, at what nodes changes occurred and how far the changes reach. This is an open problem, but there are approaches to structural diff algorithms both in a type-generic setting[LLL09] and in a untyped setting[CGM97] that could be used to complete the tool-chain. These algorithms take two snapshots and return an edit script of atomic operations that have to be applied to the original term to obtain the new one. These atomic operations have to be mapped to the insert and delete operations introduced in this chapter.

4.7. Conclusion

After the introduction of generic programming, we came back to the initial problem of making interlingual references explicit. We introduced the addressing of subterms by *paths* and the lookup of a subterm addressed by a path using the generic zipper. This

approach works for *any* language that comes with an adaptor to the multirec library for generic programming. Based on this approach, we defined addresses and references and extended them by exchangeable consistency functions. Finally we introduced the generic and higher-order function `check` which can check the consistency of such a reference.

While this can detect inconsistencies when the subterms themselves change, structural changes on the terms introduce another problem, namely that the paths might point to a wrong location after a structural change to the underlying syntax tree. The solution to this problem is to *adapt* the paths according to the change. For our purposes it suffices to adapt the paths. Given a suitable description of the transformation as a composition of insertions and deletions, we can induce a composition of adaptions on the paths we are interested in.

The main idea of our approach is, that these data structures and functions are defined *once and for all*, i.e. they capture the essential problem. For the accidental contexts of real projects, the developers will have to supply the multirec-adaptors and consistency functions on their own.

4.8. Bibliographic Notes

4.8.1. Early Programming Environments

The choice of Huet's Zipper shows that some of our ideas date back to the early eighties. Huet and others presented in [Lan85] a programming environment named MENTOR, based on abstract syntax trees. He used a specialized language, MENTOL, that is capable of navigating and manipulating syntax trees and based an interpreter and the programming environment on it.

At Cornell, Teitelbaum and Reps developed a unified programming environment called The Cornell Program Synthesizer [TR81] which had a number of influential features like incremental compilation or debugging and runtime tracing. First it was capable of working with PL/I programs. Then, later, it was extended to a synthesizer generator [RT84] to generate programming environments for various languages based on a template-based language-description, e.g. ProSet [Bub96].

In 1989, Borras et al. present the CENTAUR System [BCD+89], a generic interactive programming environment. Given a formal description of the syntax and semantics of a language, it yields a structure editor, an interpreter/debugger and other tools.

Finally, in 1990, the PAN language-based editing system [BGVDV92] was presented. Conceptually PAN allows for text-editing and wants to exploit language-based technologies as presented before.

The focus of these approaches lay on *editing* based on the abstract syntax, which did not succeed in small scale, because the edit-compile-debug cycles got faster with the advent of graphical user-interfaces and fast compilers and computers than thinking about transformations on abstract syntax trees. The notable exception is the work of Opdyke on large scale refactorings [OJ90] which are today standard operations in IDEs.

Especially the idea to *parametrize* a software tool by a language, like it is done in

CENTAUR or PAN, was highly influential for our work. Our conceptional and technical means is datatype generic programming based on a common platform like Haskell.

To the best of our knowledge, there are no approaches that extend this idea to *two or more* languages to express references between nodes of differently typed syntax trees.

In this work we pick up some of these old ideas for a static analysis feature and present our research on a modern platform.

4.8.2. Term Graphs and Graph Transformation

We have to distinguish our work from the research on graph grammars and graph transformation.

In the broader software engineering context, graphs appear in two flavors: As *term graphs* which exhibit a richer structure than ASTs. For example, different kinds of edges can be used to model reuse of common substructures. This leads to a more compact and specialized representation. The rewriting or transformation of such graphs is then subject to different research topics, like the graph representation (node/edge-labeled, hypergraphs, typing etc), production rule representation (single and double push out approach) and different execution engines (HOPS[Kah99], AGG[Tae04], Groove[Ren03], PROGRES[SWZ99]).

A pragmatic approach to inconsistency management using graphs and graph transformations as the basic model, can be found in the works of Schürr on triple graph grammars[Sch94]. Here two models represented as graphs are kept consistent via a third graph which captures the elements of the graphs that relate somehow. Graph production rules are then extended such that the rules also transform the third graph and thus keep the two models always in sync. This approach was successfully applied to the area of model driven development. For example, Königs developed an implementation of the QVT standard[Kön09] based on these ideas.

On the other hand, graphs appear often in design diagrams, most prominently the UML, but also to depict control or data flow, or finite state machines. Thus the field of graph transformation does not only work on term graphs, but uses graphs as a general model to represent highly structured data, such as UML class diagrams.

We have to argue, why we adopt the simpler model of ASTs to represent our data: Since our motivation was highly practical, the languages we have in mind are mostly *context free*. This naturally leads to a hierarchical view on terms which in turn leads – after the removal of unnecessary detail – to the notion of abstract syntax trees. With this in our mind, we chose an execution engine that allows to express such terms easily and is yet formally founded: Haskell with its underpinning in the typed lambda calculus.

We could equally well choose a graph representation for terms and employ a graph transformation approach but this had some pragmatic drawbacks and up to our knowledge only little benefits.

In the typed lambda calculus, and concretely Haskell, terms are *first class citizens*. We do not have to specify our own data structure to model terms in general. They are in the language itself. We only have to model the relevant types for the languages under investigation.

The languages that we want to investigate are typically context-free. Thus, a hierarchical representation is natural. Of course, for more sophisticated static analysis, one often transforms the ASTs to another – often graph based – representation, and then graph transformation techniques are the logical way to work with the data. But: If we chose a (custom, specialized) graph representation for terms in general, we would loose the power, convenience and strictness that comes with the typed lambda calculus in Haskell. For example, if we incorporated the types of a language under investigation, into a graph representation, we would need to argue about type-safety of our transformations in respect to these types, since Haskell only guarantees type-safety in respect to our chosen (universal) graph model for terms. The benefit would be, that we could encode more of the relevant information into the data structure.

Thus, hierarchical terms are natural, since we typically have context free languages. The typed lambda calculus gives us a formally well founded execution engine. Term graphs as a model have more structure, but we would have to specify our own special graph representation together with an execution environment. On a lower level, we can do everything with trees that we could do with graphs. Finally the main reason is a design choice: Because our main focus is on interlingual issues, we do not want to develop our own common model for many, potentially unknown languages, but rather reuse a common and proven hierarchical representation of terms.

In general, the graph grammar approach is not applicable to our work for two main reasons: First, we do not want to specify our own universe of terms together with a suitable execution engine, but reuse an existing and powerful universe that is formally founded *and* usable in practice (e.g. available parsers for a variety of languages). The second reason is that the graph grammar approach relies on the existence of a fixed set of rules. The description of term graph transformations in terms of the application of such rules (graph productions) is not feasible, since in the general case, we do not have such a fixed set of rules.

5. Implementation of a Prototype

Based on the concept of generic programming, the previous chapter introduced the core concepts and their implementation in Haskell. In this chapter we present a proof-of-concept, namely a prototypical implementation of the framework and a front-end tool that could be used in the day-to-day development cycles of software developers.

5.1. Introduction

We start by introducing the requirements. Because we are interested in the implementation phase of the software engineering process, we focus on the developer's role, i.e. the developer of a software project is our main user.

Since we assume that the users work primarily with source code in concrete syntax, the definition of references should be done from inside an *interactive source code editor*. That means in turn, that the editor for references should also be interactive. The user wants to select the relevant locations in two different files, create an explicit reference for these, and save it for later usage. Once the references are defined, developers want to *check* the consistency of the project while they are changing the project or add new features. This can be done interactively from within the editor or non-interactively inside a batch-process as commonly used in continuous-integration settings.

Thus we need two different modes for the front-end to define and consume references. The resulting tool has to build up upon the generic programming facilities introduced in chapters 3 and 4. In particular, we need common data structures to define generic references and an interface to the supported languages and the consistency functions.

One core requirement is, that the application should be extensible for new languages and contexts. Thus, our architecture has to separate a core library from context dependent and language specific issues.

5.2. Architecture

A conceptual view of the architecture is shown in Figure 5.1. We choose to present the architecture of the system as a UML package diagram, where we interpret a Haskell module as a package that contains types (shown as classes) and functions (shown as text notes containing their complete signature) and perhaps other modules. The main architecture can then be described in terms of dependencies between the modules.

The Haskell platform forms the basis of the system. Most other modules use convenience modules to deal with data structures or modules that provide combinator libraries, e.g. for different parsing approaches.

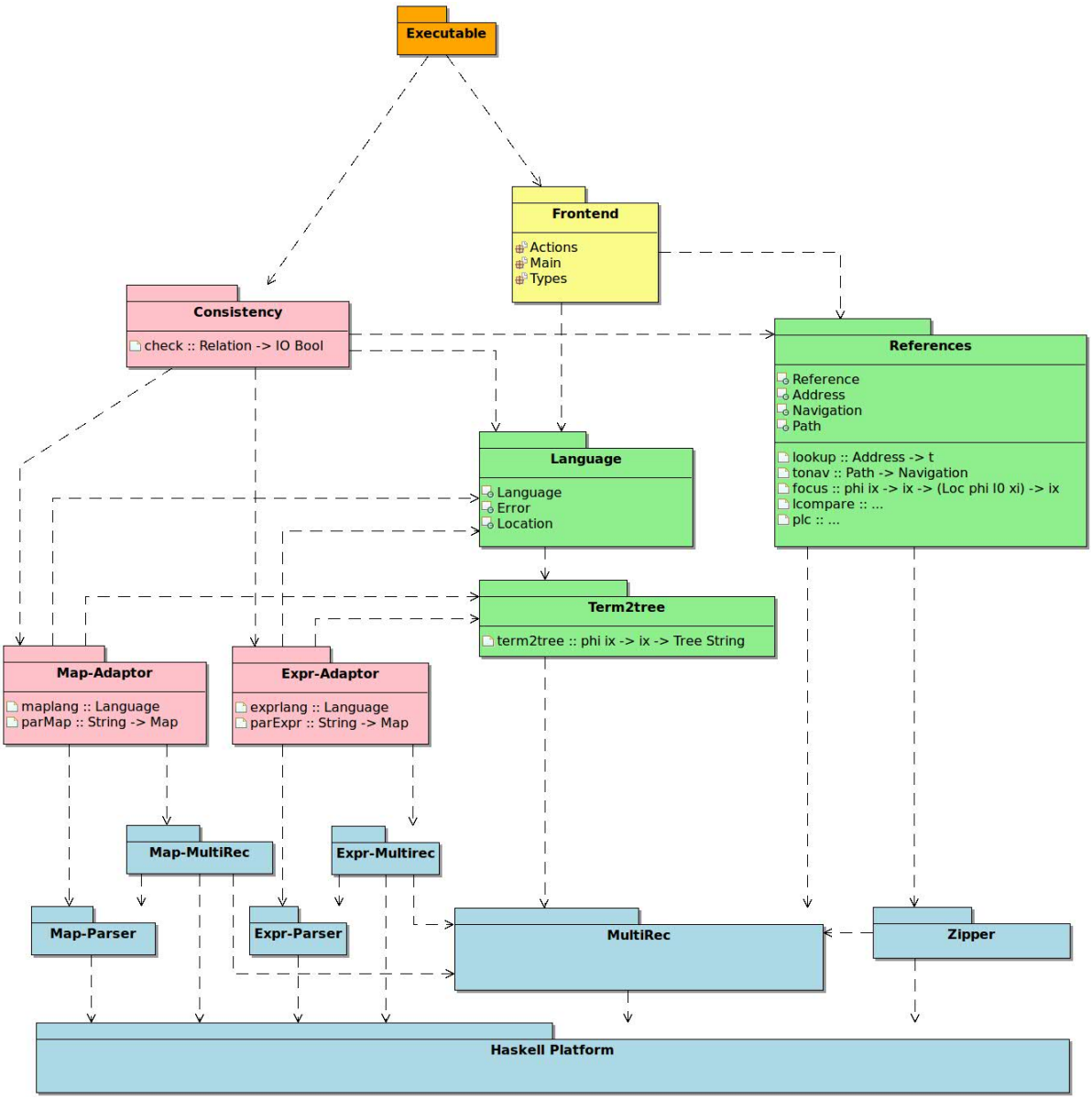


Figure 5.1.: Main architecture

On top of the Platform we emphasized some modules that are of special interest in this work. The generic programming libraries `multirec` and the `zipper` on the right and on the left – as an example – the modules containing the parsers and types for the abstract syntax of our toy languages for maps and expressions together with their adaptors to the multirec library (cf. 3.3). These modules (with a blue background) are supposed to be given. Note that the modules on the right hand side are independent of the languages, while the modules on the left hand side depend on the context. This separation continues at the next level.

On the right hand side, with a green background, we find the core modules of our approach as introduced in the previous chapters. The module `Term2tree` contains the generic function to translate a typed Haskell term to an untyped tree of strings as shown in 3.4.1. The module `References` defines the generic references and addresses and uses the generic zipper to specify the lookup and consistency-check of subterms that are available generically. The module `Language` defines some datatypes to deal with languages. A type for languages contains the name of the language, the possible file name extensions, the parsing function, a function to extract source coordinates of subterms inside a file etc. Further the module contains types to represent parser errors and source code locations.

On the left hand side, with a red background, we find the context-dependent modules, i.e. adaptors to the map and expression languages. We can see that these modules provide values for the types defined in the `Language` module by *using* the language-specific functions from the underlying parser modules. Further we see the module `Consistency` that defines the consistency functions for expressions and maps.

Finally, we developed a simple front-end, to define and check references. The front-end consists of a graphical user interface and a set of actions that can be executed by the user. This (generic) front-end can then be parametrized by the consistency module and the languages that should be supported to obtain a context specific consistency tool based on our generic framework.

5.3. A Library to manage References

We begin with the description of the second architectural layer, namely the core library, the language adaptors and the consistency module. Following the idea of Brooks to separate essence and accidents in software engineering, we divided the layer into the essential modules that are independent of any concrete language, from the accidental and context-dependent parsers and consistency functions.

5.3.1. Essence: References, Languages, Term2tree

The library consists of three modules which are completely language independent. The module `References` defines the functions and datatypes as explained in Chapter 4, the module `Language` defines a common interface to the languages that should be supported

and the module `Term2tree` the generic function that translates a term into a tree of strings as presented in Section 3.4.1.

Language

Any language that should be supported needs a parser that either transforms the program text into a term or returns an error. Further the program texts are supposed to reside in files with a certain extension to recognize the corresponding language. A language has a name to identify it and a syntax highlighter. To map source coordinates from within an editor to nodes in the syntax tree, each language needs a function that produces a list of such source locations for any tree of strings. Thus the specification of the language datatype looks like this:

```
data Language = forall a phi. Language
  { name   :: String                -- ^ language name
  , syntax :: String                -- ^ syntax highlighter name
  , exts   :: [String]              -- ^ file extensions
  , parse  :: String → Either Error a  -- ^ parse function
  , toTree :: a → Tree String       -- ^ term to tree
  , srcLoc :: Tree String → [Loc] -- ^ extract source locations
  }
```

We use record syntax to give names to the arguments of the single constructor `Language`, many of them have a function type, e.g. the `parse` and the `toTree` function. Using forall quantification for the free type variables `a` and `phi` allows us to build a homogeneous list of languages. The drawback is, that we cannot recognize the type `a` from an arbitrary value of the language type.

The `Language` module also comprises simple datatypes for parse-errors (`Error`) and source locations as offsets and coordinates (`Loc`).

References

The data structures and functions for explicit references as presented in Sections 1.1.2 and 4.3 are the generic addresses and references together with the lookup-functions that make use of the generic zipper. These are bundled in the module `References`.

The module further defines a generic, higher-order function called `plc` that, given a reference, parses both files, performs the address-lookup, and applies the given consistency function to the addressed subterms. This function is the very heart of our implementation as it forms the glue between parsing, locating subterms and checking for consistency. Thus we must explain it in depth. The definition is given in figure 5.2.

As always, once we understand the types, the implementation is fairly clear. Lines 1 to 4 constrain the polymorphic type variables: $\texttt{phi}_i$ are the families of mutual recursive types, and $\texttt{ix}_i$ are the respective members of these families. Further both families have to be in the `Zipper` type class.

In lines 5 and 6, we pass the two type representations which are needed to use the zipper later on. In lines 7 and 8, we pass two parse functions to `plc` which produce

```
plc :: (Fam phi1, El phi1 ix1,
        Zipper phi1 (PF phi1),
        Fam phi2, El phi2 ix2,
        Zipper phi2 (PF phi2))  =>
    phi1 ix1                  -- p1 typerep to enter l1
 → phi2 ix2                   -- p2 typerep to enter l2
 → ((String → ix1))           -- parse function for p1
 → ((String → ix2))           -- parse function for p2
 → (forall xi1 xi2. phi1 xi1 -- generic consistency function
                  → phi2 xi2
                  → xi1
                  → xi2
                  → Bool)
 → Relation                    -- paths and filepaths
 → IO Bool

plc p1 p2 gp1 gp2 gc (Relation (Elem pf1 fp1) (Elem pf2 fp2) _)
  = do
      l1 ← fmap (fromJust . tonav pf1 . enter p1 . gp1)
                (readFile fp1)
      l2 ← fmap (fromJust . tonav pf2 . enter p2 . gp2)
                (readFile fp2)
      return $ (lcompare gc) l1 l2


-- | Higher Order Compare of zipper locations along the
     lines of 'on' 'update' and 'modify'
lcompare :: (forall xi1 xi2. phi1 xi1
                           → phi2 xi2
                           → xi1
                           → xi2
                           → Bool)
         → Loc phi1 I0 ix1
         → Loc phi2 I0 ix2
         → Bool
lcompare gc
         (Loc p1 (I0 x1) cs1)
         (Loc p2 (I0 x2) cs2)
         = gc p1 p2 x1 x2
```

Figure 5.2.: Function plc: parse - locate - check

typed terms of type `ix`$_i$. In line 9 we pass a generic consistency function that works for *any* member type of the two families. Given two arbitrary subterms of the two languages, this function evaluates to `True` when they are consistent. We will show an example when discussing the context-dependent consistency module itself. Line 10 finally contains the data needed to perform the check: `Relation` is the persistable type for a reference, a value of type `Relation` contains the navigation paths into the syntax trees and the respective file names. Because we have to read the files from the file system, the function returns a monadic, Boolean valued IO-action to indicate whether the comparison succeeded.

The implementation of the function in lines 19 to 21 is then easy to understand: for both files, we read the contents from files `fp`$_i$ in the file system, then we parse the contents using the generic parse functions `gp`$_i$. We turn the resulting term into a zipper location using `enter` with the type representation `p`$_i$ of the root of the terms. Using `tonav` (cf. Section 4.3) and the navigation paths `pf`$_i$, we finally build zipper locations `l1` and `l2` that point to the right positions in the terms. Together with the generic consistency function `gc`, which is able to compare values of all types of the two families `phi`$_i$, the two zipper locations are passed to the helper function `lcompare` which simply applies the consistency function to the current foci `p`$_i$ of the two locations `l`$_i$.

Additionally, the module `References` defines the persistence layer for references, namely the datatype `Relation` together with a suitable parser and pretty-printer.

Term2tree

This module is just the generic function already discussed in Section 3.4.1 based on the multirec library. It simply provides a function that translates any typed term into a tree of strings. This is occasionally useful when we need an untyped representation of the syntax tree, e.g. when visualizing terms in a graphical tree widget whose data model is a tree of strings or when extracting source locations from the tree.

5.3.2. Accidents: Language Adaptors and Consistency

The accidental issues of a specific project context are the used languages and the meaning of consistency for two types in this context.

Adaptors

For each language that should be supported in our framework, one needs to implement the functions that make up a value of the `Language` datatype. This means, one needs a parser that produces an abstract syntax tree with additional information about the source-locations, an adaptor to make the types usable by generic functions (as shown in Section 3.3) and some meta information about the language such as a displayable name and a list of file name-extensions for this language.

Thus, everything that is specific to the language is bundled in these modules and the tools have a clean interface to any language. For example, the adaptors to our running

examples for maps and expressions, in Figure 5.1 on page 68 contain the values (a constant function) `maplang :: Language` and `exprlang :: Language` and we can access the parser for maps by selecting the parse field of the language value:

```
parse . maplang :: String → Either Error Map}.
```

Consistency

The consistency module exports the crucial `check` function which checks a given reference for consistency. Since this is the function where the concrete languages are *bound* to the generic framework, namely the function `plc`, we have to go into the details to show how this works.

A relation value is the persistent representation of a reference. It consists of two file names, two navigation paths and the name of a consistency function. Given these values, the function `check` determines by pattern-matching on the file name extensions how the core function `plc` should be called, especially in respect to the type representation and the type specific parsing functions:

```
check :: Relation → IO Bool
check r@(Relation (Elem (DPath pf1) fp1)
                  (Elem (DPath pf2) fp2)
                  cf
        )
 = do
   let ext1 = takeExtension fp1
   let ext2 = takeExtension fp2
   case (ext1,ext2) of
    (".map",".map") → plc (TheMap) (TheMap)   pM pM (cMM cf) r
    (".map",".ast") → plc (TheMap) (TheExpr)  pM pA (cMA cf) r
    (".ast",".ast") → plc (TheExpr) (TheExpr) pA pA (cAA cf) r
    (lext,rext)     → error $ "comparing file types "
                              ++lext++" and "++rext
                              ++" is currently not supported."
```

As we can see, for our running examples of arithmetic expressions and maps, we discriminate by means of the extensions `".map"` and `".ast"` which parsing functions (`pA` and `pM` stands for parseAST and parseMap) should be used. Additionally, we fix the type representations `TheMap` and `TheExpr` here. These are representations of the type of the root of the term and are needed by the generic zipper called in `plc` to `enter` the term (cf. Section 4.2).

Now, we explain the definition of consistency functions. As an example we look at the definition for the consistency of arithmetic expressions and maps. The kind of consistency we want to express in this example is, that every variable that occurs in an expression is defined as a key in a map. Thus, the map is a valuation for the free variables in the expression.

We can define a case for *all* pairs of types and constructors in the two languages we are interested in. For this example, we can enumerate all combinations, but for larger

	Expr		Decl
Entry	E k v	EVar k	
	E k v	Const i	E k v c := exp
	E k v	Neg e	E k v None
	E k v	Add e1 e2	E k v Seq d1 d2
	E k v	Mul e1 e2	
Map	Empty	EVar k	
	Empty	Const i	Empty c := exp
	Empty	Neg e	Empty None
	Empty	Add e1 e2	Empty Seq d1 d2
	Empty	Mul e1 e2	
	Cons e m	EVar k	
	Cons e m	Const i	Cons e m c := exp
	Cons e m	Neg e	Cons e m None
	Cons e m	Add e1 e2	Cons e m Seq d1 d2
	Cons e m	Mul e1 e2	

Figure 5.3.: All combinations of types and constructors for maps and expressions

types, this will be infeasible, thus the user has to make a choice which combinations are of interest to him. The possible combinations for expressions and maps are shown in Figure 5.3. The row and column headers contain the types and inside the cells, there is a line for each pair of constructors.

We will show the signature of the function `cMA` and pick out one of these cases for illustration. Note, that the dot notation is used here to denote types and constructors of the specific modules `Map` and `Expr`.

```
cMA :: CompareId
    → Map.MAP ix1 → Expr.AST ix2
    → ix1 → ix2
    → Bool

cMA (CompareId "keys")
    Map.TheEntry  Expr.TheExpr
    (Map.E k v)  (Expr.EVar var)
  = (k == var)
```

For now we ignore the first argument which may be used to implement different kinds of consistency for one combination of types. Again we have to provide type representations, now for the type of the addressed subterms. Then the implementation for the consistency of key-value-pair `E k v` in a map with a variable `EVar var` in an expression is just the equality of the variable names: `k == var`. For the consistency of such an entry with recursive expressions e.g. `Add e1 e2`, we apply the consistency function recursively with the nested expressions.

As a second example for a different kind of consistency, we could assume that, regardless of the key, the *value* of a map entry should coincide with the *evaluated* value of an

expression. The implementation for such a case might look like this then:

```
cMA (CompareId "eval")
    Map.TheEntry  Expr.TheExpr
    (Map.E k v)   exp
  = (v == eval exp )
```

To discriminate the two kinds of consistency for maps and expressions, we use a value `CompareId ‘‘eval’’` as opposed to the `CompareId ‘‘keys’’` in the previous example for a consistency function.

This shows, that the user of this framework can use the full power of Haskell to define consistency as appropriate for his context. In our example one might be interested in the consistency of the variable *names* or in the consistency of *values* that occur in different shapes in the two languages for maps and expressions.

5.4. A visual Manager for References

To define, edit and check references, we built a prototype to demonstrate the basic features. This makes up the front-end as introduced in Section 5.2.

For the sake of a prototype, the editor also comprises a number of debugging features (esp. a detailed view on the underlying abstract syntax trees) that would not be needed in a productive setting. In the following we present only the user perspective on the tool, as it is a standard graphical user interface to the underlying facilities.

After startup, the program presents a main window with two embedded source views and a list of references to edit. Of course, the main actions inside the editor are the creation, modification and check of references.

To create a reference, the user opens two source files via the Left/Right menu, selects two spans in the source views, and chooses the Add Reference action from the Relations menu[1] to add it to the list of references for the current project. For each reference, the user can choose the name (cf. `CompareId`) of the specific function to apply for the subterms (cf. Section 5.3.2).

A double click on an existing reference, opens the underlying files into the source views and jumps to the respective positions in the editors. The user can then modify the reference by selecting a different source location and choosing the Modify Reference action from the references menu.

To delete a reference, the user has to select the reference from the list, and choose the Delete Reference action from the menu.

The set of references can be saved and loaded from the File menu. The idea is here that such a set of references is defined for each project and saved into a so-called Rulefile.

To ease debugging, the prototype also comprises a graphical presentation of the underlying abstract syntax tree. From a text selection, the user can *jump* to the corresponding location in the graphical tree component. This is convenient to understand and work

[1] typographical note: Elements of the user interface are typeset in a sans-serif font

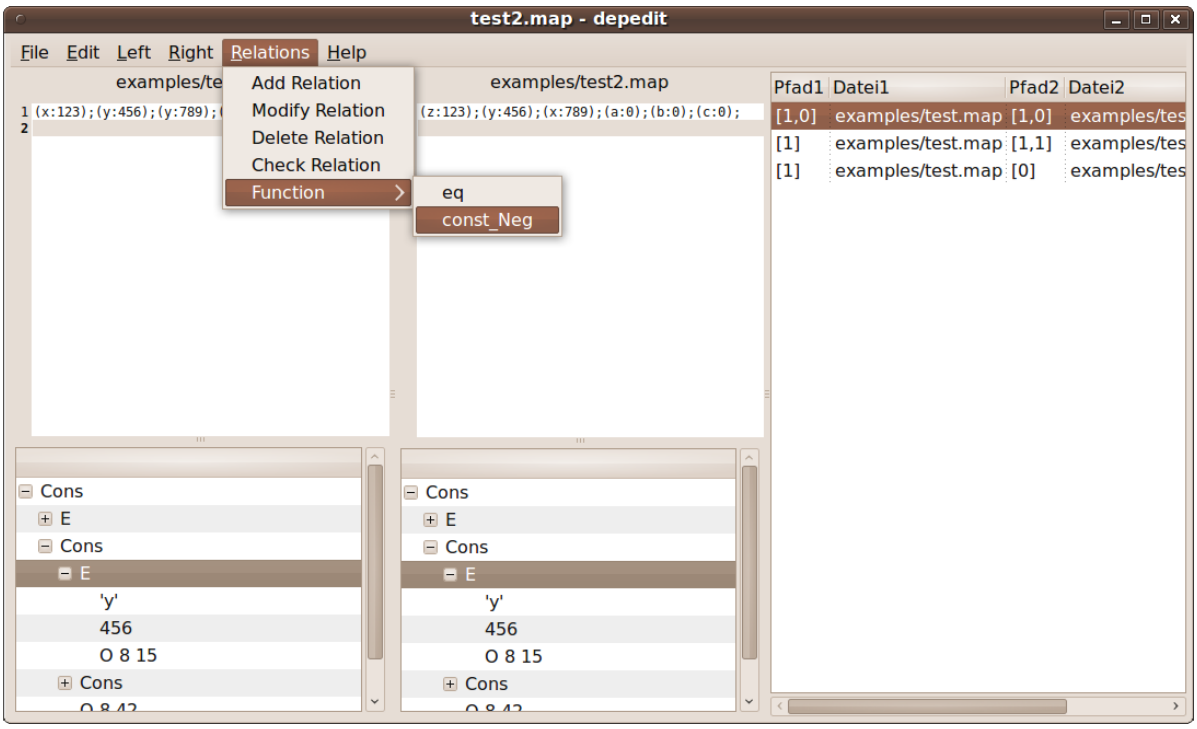


Figure 5.4.: **Depedit** the visual editor for references

with address-paths. While the checking of references is not meant to be made interactively, the editor also provides an action Check Reference in the the References menu.

Additionally, all references in the Rulefile can be checked non-interactively in a batch mode. This would be useful in a continuous integration setting, e.g. for nightly builds of a software product as the resulting report could be sent by Email to the developers or the quality assurance department.

5.5. Conclusion

In this chapter we presented a prototype for the explicit management and usage of references. The front-end can be used as a standalone interactive editor. When used in batch-mode, the tool could be integrated into continuous integration settings. The library and the front-end are independent of concrete languages: Once suitable adaptors and consistency functions are implemented, they can be linked together with the front-end to get another context-dependent reference editor and checker.

The main focus of this chapter was on the system architecture. Central is the library which builds on the generic programming paradigm for mutual recursive datatypes and the corresponding zipper as presented by Löh et al. in [YHLJ09]. If common languages and their Haskell parsers would come along with a *multirec-adaptor*, any such language could be used with our library without changing it substantially. Of course, due to the accidental aspects of the languages and consistency functions as opposed to the generic

design of the core library, the adaptors for supported languages and the consistency module have to be linked together with the library and front-end for each specific context.

The clean separation between the essential formulation of reference management from the accidental languages and consistency functions is the most important aspect of this work. Using the recent techniques of datatype generic programming, this separation is finally possible and can be used in a relatively easy way to do cross language program analysis.

To define consistency for two languages, one has to define what it means for any combination of constructor calls to be consistent. This is infeasible for realistic languages as we will see in the following chapter. Additionally, there might be different notions of consistency for the same combination. For example consider a string literal of a programming language and a plain text content of an arbitrary data format. We cannot give a meaningful definition of consistency for the corresponding constructors without additional knowledge about the context of the project. For these two reasons, namely the complexity of languages, and the accidental nature of consistency, we leave the definition of consistency to the user, who can then decide to implement only those combinations in the way she needs it in her project.

6. Case Study

6.1. Introduction

We will now conduct a case study to show how the approach works for a small application using *real* languages instead of our toy languages for maps and expressions. We continue with the example from the introduction by developing a template engine for web pages. Then, we will analyze the application for interlingual references and context-dependent consistency issues and show how these can be made explicit using our prototypical implementation from Chapters 4 and 5.

6.2. A simple Template Engine

A template engine for web pages is a good candidate for our purpose because it typically involves at least three different languages: Since the application is dynamic in the sense that it *generates* web pages, we need a core language to do the data processing. The data itself is made persistent in some dedicated format, and gets parsed by the main application before being processed further. Finally, the resulting web pages are given as templates, thus they are given in a markup language like HTML. This way, we have separated dynamics, data and appearance of the application into three dedicated languages which have to be reintegrated somehow.

We choose the dynamically typed language Python for the main application, the JSON format to represent data and XHTML as the markup language for the web pages. Before sketching the implementation, we have to explain the languages and their Haskell parsers shortly.

6.2.1. Python

Python is a dynamically typed language that is often used for so-called scripting tasks, namely small programs developed in a quick and dirty fashion. For larger applications it has also object-oriented features. In the mix, it is quite usable for web development and thus widespread. Our choice fell on Python rather than Java, C or Haskell, since it allowed for the development of a quite short and concise application.

The Haskell parser library we use is based on the Alex[DM05] and Happy[MG09] lexer/parser generators. The Python syntax has 27 different single types and 127 different constructors in total. Since the constructors have often lists, tuples or lists of tuples of the other types in their arguments, the number of types that has to be considered in

the multirec-adaptor, is much larger. For illustration we show an excerpt of the Haskell definition of the datatype for Python expressions:

```
data Expr
   = Var { var_ident :: Ident}
   | Int { value :: Integer, expr_literal :: String }
   | LongInt { value :: Integer, expr_literal :: String }
   | Float { value :: Double, expr_literal :: String }
   | Imaginary { value :: Double, expr_literal :: String }
   | Bool { value :: Bool }
   | ByteStrings { byte_string_strings :: [String] }
   | Strings { strings_strings :: [String] }
   | CondExpr
     { ce_true_branch :: Expr
     , ce_condition :: Expr
     , ce_false_branch :: Expr
     }
   | BinaryOp { operator :: Op, left :: Expr, right :: Expr}
   | UnaryOp { operator :: Op, arg :: Expr}
   | Lambda { lambda_args :: [Parameter], lambda_body :: Expr}
   | Tuple { tuple_exprs :: [Expr] }
   | List { list_exprs :: [Expr] }
   | Dictionary { dict_mappings :: [(Expr, Expr)] }a
   | ...
```

Note that this type references also lists of expressions and lists of pairs of expressions. For each such plain type we need the according adaption to the type classes for generic programming as shown in Section 3.3.

6.2.2. JSON

JSON stands for JavaScript Object Notation[Cro06]. It was designed as a data interchange format for the JavaScript language with less overhead than XML. It is a plain text format that allows for string and numeric values, heterogeneous arrays of values and key/value dictionaries. Since arrays and dictionaries are also JSON values, deeply nested structures are possible.

The Haskell parser `hjson` we choose has a simple abstract syntax and uses the monadic parser library Parsec[LM01]. Since the abstract syntax is so simple we show it for later purposes:

```
data Json = JString String
          | JNumber Rational
          | JObject (Map String Json)
          | JBool Bool
          | JNull
          | JArray [Json] deriving (Eq, Show)
```

We see that JSON dictionaries are called objects here and that they are represented using Haskell's datatype `Map`. JSON arrays are represented as lists of JSON values. This JSON representation has only one type and six constructors. It is still mutually recursive due to the usage of the Map datatype.

The concrete syntax of JSON is simple as well: Strings, numbers and Booleans are represented as in Java or C, arrays are a list of comma-separated values enclosed by brackets, an object is a list of comma-separated key/value pairs enclosed by braces. A key/value pair is a sequence consisting of a string, a colon and a value. Examples for the concrete syntax will follow in the next Section.

Note that by fortune or consciously, the syntax for JSON's objects and Python's dictionaries is just the same.

6.2.3. XHTML

XHTML is the XML based hypertext markup language. Thus any XML-Parser can parse it and thus validate it.

We use the general Haskell XML-Parser *HaXml* [WR99]. It defines general types for XML elements like `Element`, `Attribute` and `CData` to represent XML Documents with tags, attributes and character data between the tags. The abstract syntax has about 14 types and 17 constructors excluding the representation of DTDs. HaXml uses yet another parser combinator library called `polyparse`[Wal09].

6.2.4. A Note on the Languages

We chose these languages for pragmatic reasons: We needed something that is easy to present and has good Haskell support at the same time. But we can pinpoint an assumption we made in the very beginning when we chose Haskell as our main implementation language.

We assumed that there are many parser libraries for the various languages available in Haskell. Because these libraries all use Haskell as their common language, they all define the abstract syntax in terms of algebraic data declarations and a parsing function that maps strings to Haskell terms.

Playing the same game in Java for example, would result in a far more diverse situation. The object-oriented design of an abstract syntax is hardly as concise as the algebraic formulation employed in Haskell. In the object-oriented world, there would be additional features in the class-design of an individual parsing-library that would make them hard to integrate. As an example, we mention the Visitor pattern, that would be implemented in each of these libraries.

6.2.5. Sketch of the Implementation

Python has a feature to replace variables in strings by values of a dictionary. For example

```
dict = {"name":"Josefine"}
print "Hello {name}".format(**dict)
```

would print `Hello Josefine`. We will use this feature to embed variables into XHTML-pages. For example we might have an XHTML file that contains the following:

```
<p>{greeting} {name}</p>
```

Then our Python code reads the contents of that file into a string variable and replaces the variables in it by using `format`.

The data for the variables is stored in a JSON file. Continuing with the introductory example from above, the contents would look like this:

```
{"name"    :"Josefine"
,"greeting":"Hello"}
```

To configure our application, we maintain a registry that maps page and language identifiers to file names. This registry is also stored in a JSON file and is structured as nested dictionary consisting of two dictionaries for languages and templates.

```
{"langs": {"de":"dic/de.json"
          ,"en":"dic/en.json"}
,"pages": {"p1":"pages/page1.xhtml"
          ,"p2":"pages/page2.xhtml"
          ,"p3":"pages/page3.xhtml"
          ,"stats":"pages/stats.xhtml"}
}
```

We need two additional libraries for the Python implementation, one to parse the JSON data to Python's data structures and one to handle the communication with the browser. The use of Pythons JSON parser is quite easy, one just passes a file handle to the parser's load function: `regc =json.load(regf)` (here `regf` is the file handle to the registry file).

To call the application from the browser we use the old fashioned CGI interface: The web server calls the application and sends all standard output of the application back to the browser. Named parameters may follow after the name of the called web application: For example, the query `http://google.com/search?q=test&hl=de` calls the Google search engine with the query `test` passed as parameter `q` and the language `de` as parameter `hl` (stands for for host language). The dictionary of parameters can be accessed in a Python application using the following lines:

```
form = cgi.FieldStorage()
query = form.getvalue("q")
language = for.getvalue("hl")
```

After these technical preliminaries we sketch the operation of our main Python script `main.py`. The full source code is show in Figure 6.1 on page 83.

First, the program needs to get the values of two named CGI arguments, namely the page-id and the language-id (lines 8-11). The names for these arguments are `pid` and `lid` respectively. Thus a valid call could be: `http://localhost/main.py?pid=p1&lid=de`. The values of the arguments are stored in Python variables `cpid` and `clid` (current page/language id) and somehow make up the observable state of the application.

```
#!/usr/bin/env python
import cgi
import cgitb; cgitb.enable()  # for troubleshooting
import json
import os
import datetime

# get current page and lang id from http request
form = cgi.FieldStorage()
clid = form.getvalue("lid", "de")
cpid = form.getvalue("pid", "p1")

# open registry file and parse contents
regf = open("registry.json", 'r')
regc = json.load(regf)

# read template and language dictionary as specified in registry
pagef = open(regc["pages"][cpid], 'r')
pagec = pagef.read()
langf = open(regc["langs"][clid], 'r')
langc = json.load(langf)

# print http prolog (MIME-type)
print "Content-type: text/html"
print

# generate dynamic content: create dynamic variables that
# can be used in templates
vars = langc                          # start with dictionary
vars.update(os.environ)               # add  environment  vars
vars.update({'cpid':cpid})            # add  current  page id
vars.update({'clid':clid})            # add  current  lang id

# add time stamp
dateString = datetime.datetime.now().strftime("%Y-%m-%d %H:%M")
vars.update({'date':dateString})

# foreach lang-id create a variable that gets replaced with a link
for l in regc["langs"].keys() :
  langLink = "<a href=\"main.py?pid="+cpid+"&lid="+l+"\">"+l+"</a>"
  vars.update({l:langLink})

# foreach page-id create a variable that gets replaced with a link
for p in regc["pages"].keys() :
  pageLink = "<a href=\"main.py?pid="+p+"&lid="+clid+"\">"+p+"</a>"
  vars.update({p:pageLink})

# replace all variables and render the template
print pagec.format(**vars)
```


Figure 6.1.: Main template engine

Then, the program reads the JSON file `registry.json` and looks up the files for the current page and language (lines 13-15). The JSON file for the language is then parsed to a Python dictionary `langc`, the XHTML-template stored in a string variable `pagec` (lines 17-21).

Next, additional fields are added to the dictionary `vars`, e.g. the environment variables, the current date, or other dynamic values from the script's environment. To use page and language ids as variables in the templates, we also add entries that map these ids to XHTML hyperlinks (lines 27-46).

Finally, we replace all variables that occur in the template `pagec` by the values of the dictionary `vars` as shown above and print the resulting string to the standard output, which is then redirected to the requesting browser.

We created three small pages which all have a navigation bar on the left, title and content that should be localized. We also created two language dictionaries for the respective variables, one in English and one in German. A rendered example is depicted in Figure 6.2 on page 85. The figure shows two templates each in English and German language together with arrows to denote the transitions between the pages. The source code of the underlying templates and JSON files are given in Appendix C.

6.3. Analysis

In this section we analyse the application for implicit interlingual references and consistency restrictions that are out of scope of traditional program analysis tools. To do so we will show snippets from the source code and explain the references.

6.3.1. Python to XHTML

There are two kinds of references between the main program in Python and the XHTML templates: On the one hand, the program can define dynamic variables that may be used in the templates. On the other hand, the names of the arguments passed to the program in form of a URL, may be used in links in the templates.

For the first case, the addition of an entry to the `vars` dictionary in Python is given by the following statement:

```
vars.update("cpid",cpid)
```

In the XHTML template we refer to that variable by the following markup:

```
<p>The current page-id is: <b>{cpid}</b></p>
```

For the second case we have a Python statement like

```
cpid = form.getValue("pid","home")
```

while a link to the script in XHTML code would be:

```
<a href="main.py?pid=p1&lid=de>Page 1</a>
```

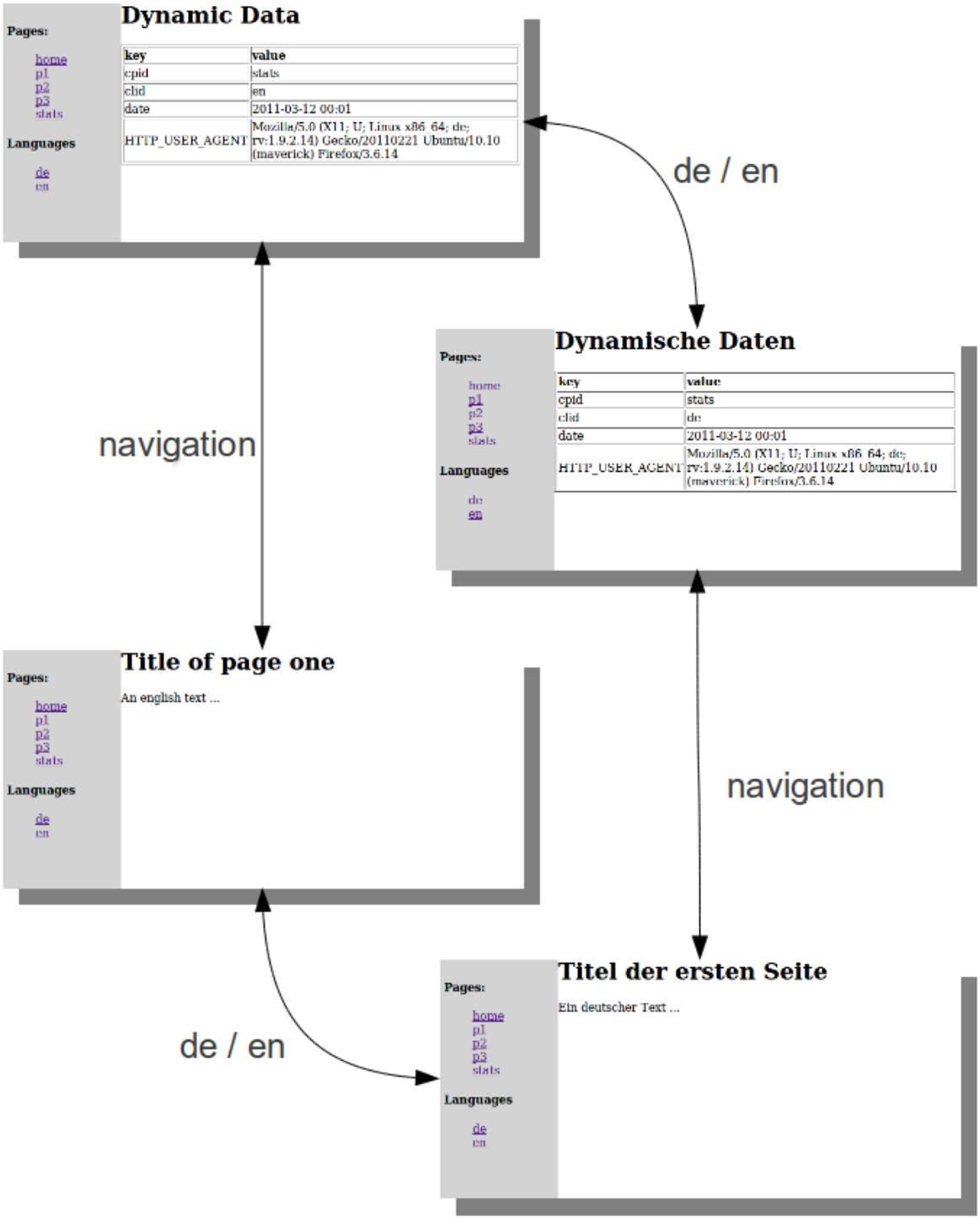


Figure 6.2.: Rendered templates with dynamic data

6.3.2. JSON to Python

When we want to look up a file name that belongs to a certain page or language id, we use the JSON-keys `"pages"` or `"langs"` as string literals ins Python.

After we parsed the JSON-data for the registry to a Python dictionary `regc`, we look up the template that is addressed via the variable `cpid` using these Python statements:

```
pages = regc["pages"]
pagef = open(pages[cpid], 'r')
pagec = pagef.read()
```

We look up the pages dictionary `pages` from the registry at key `"pages"` and then look up the filename for the page-id given in variable `cpid` before we read the contents of the file to variable `pagec`.

Thus we have a reference between the two occurrences of the string literal `"pages"` in Python and JSON.

6.3.3. XHTML to JSON

The most obvious kind of implicit references is that between variables that are used in XHTML templates and defined in JSON dictionaries:

```
{"title1":"Titel der ersten Seite"
,"text1":"Text der zweiten Seite"
, ...
}
```

This defines the German contents for variables `title1` and `text1`, whereas these variable are used in the following snippet of an XHTML template:

```
<div id="content">
  <h1>{title1}</h1>
  <p>{text1}</p>
</div>
```

Since we chose to allow for page and language as variables that expand to navigation links, we have also references between the navigation part of a template

```
<div id="nav">
  <h4>Pages:</h4>
  <ul>
    <li><a href="main.py?pid=p1\&lid=de">home</a></li>
    <li>{p1}</li>
    <li>{p2}</li>
    <li>{p3}</li>
    <li>{stats}</li>
  </ul>

  <h4>Languages</h4>
  <ul>
```

```
    <li>{de}</li>
    <li>{en}</li>
  </ul>
</div>
```

and the registry as seen above.

6.3.4. Intralingual References

Finally, we have additional intralingual, context dependent consistency restrictions. Consider the JSON dictionaries for English and German translations. We could require, that they define the *same* set of keys. We will use this example to show two things: Firstly, that we can use our framework for standard intralingual consistency checks. And secondly, that consistency checks can be made on more complex types than strings.

6.3.5. Summary

We give a short summary of the analysis of the implicit interlingual references that occur in our small template engine: The key role play the variables that are *used* in the templates but *defined* elsewhere, namely either statically in a JSON-file or dynamically in the Python program. Further we use the CGI-protocol to send arguments from a URL to the program. The names of these arguments may appear inside any hyperlink in a template and are accessed by their name in the Python program. We might also want to to perform intralingual static checks. The example of identical key sets for JSON dictionaries is one example, another could be to perform static checks on a Python program, e.g. to ensure the correct use of the dynamically typed variables. This kind of analysis on the level of the abstract syntax comes close to the intention of Crew's approach in [Cre97] for static program analysis.

6.4. Adapting the Framework

For the implementation of the context-dependent parts of the framework shown on the left of Figure 5.1 on page 68, we first need adaptors to the various languages. We already introduced the Haskell parsers in Section 6.2.

To treat the resulting Haskell terms that represent the files in abstract syntax generically we need adaptors to the generic programming library we use, i.e. `multirec`. These modules provide the adaption introduced in Section 3.3. The library itself contains a helper tool that generates both the needed pattern functor and the `from` and `to` functions for the generic representation of the terms from a list of types. The details are explained in [YHLJ09]. For large families of types – remember that our Python syntax has about 35 mutually recursive types and 130 constructors – the type of the resulting

pattern functor is really huge[1].

Based on the parsers and the multirec-adaptors we have to create a value of the `Language` datatype as introduced in Chapter 5. This contains a uniform access to the parser, the extraction of source code coordinates from the AST, name and file name extensions of the respective languages.

Once we have defined these adaptors and language-values for the three languages Python, JSON and XHTML, we have to define the crucial consistency module. As shown in the preceding chapter, we need specific consistency functions for each pair of languages and have to define a case for each pair of constructors. Of course, this is infeasible for the large number of types that are involved in the Python and XHTML languages. Thus we have to make a sensible selection based on the analysis of Section 6.3. Before showing the consistency functions for each pair of languages that we want to inspect, we shortly recapitulate the type of the consistency functions, which is:

```
CompareId → phi1 ix1 → phi2 ix2 → ix1 → ix2 → Bool
```

We have five arguments and return a Boolean. `ix1` and `ix2` are the type variables that represent the actual types of the two terms we want to compare. The type variables `phi1` and `phi2` stand for the type families, i.e. the languages to which the respective types belong. Thus `phi1 ix1` is the type representation for type `ix1` in family `phi1`. Since our generic framework *cannot* infer the types, we have to specify them explicitly. Finally the first argument is an additional discriminator in case we need different functions for the same pair of types.

Now we can start the discussion of the context-dependent consistency functions that are passed to the core generic function `plc` as defined in Chapter 5. We follow the same outline as in the analysis section above.

6.4.1. Python to XHTML

We have to define two cases for the references between Python and XHTML. Both relate a Python expression (i.e. a string literal) with XML content (i.e. plain text in between XML-tags). Thus we need a `CompareId` that can be chosen from the GUI. We start with the concrete type for this function:

```
cPX :: CompareId
    → Python.PYTHON ix1
    → Xml.XML ix2
    → ix1
    → ix2
    → Bool
```

We fix the type variable $\texttt{phi}_i$ to type families for Python and XML. Thus this function is applicable to every pair $(\texttt{ix}_1,\texttt{ix}_2)$ of types where $\texttt{ix}_1$ is a type of the Python family and $\texttt{ix}_2$ a member of the XML family. We call the function `cPX` to indicate that we **c**ompare a **P**ython value with a **X**ML value.

[1] Since this is infeasible to present here we chose to publish the adaptors (together with all other source-code) on the Internet. See Appendix A.

For the first case, we relate a dynamically created variable in Python with the use of such a variable in an XHTML template. We pass the compare-id, the two type representations for Python expressions (`TheExprSpan`) and XML content (`TheContent`) and two concrete values of these types to the generic consistency function and define that those subterms are consistent when the strings are equal except their first and last characters. (A string literal in the Python language is enclosed in double quotes, a variable in XML content is surrounded by curly braces. The helper function `strip` removes these.)

```
cPX (CompareId "dynvar")
    Python.TheExprSpan
    Xml.TheContent
    (Python.Strings [ss1] _)
    (Xml.CString _ ss2 _)
    = strip ss1 == strip ss2
```

The second case is the occurrence of URL-arguments that are used in Python to get the arguments from an HTTP-request, and in hyperlinks that appear in the XHTML templates. Because the types do not differ from the previous case, we have to use another compare-id. The consistency condition is then, that the stripped Python string literal is a specific infix of the XML element. The function `fetchHref` is another small helper function that returns the value of the `href` attribute of an XHTML-tag.

```
cPX (CompareId "httpvar")
  Python.TheExprSpan
  Xml.TheContent
  (Python.Strings [ss1] _)
  (Xml.CElem (Xml.Elem _ as _) _)
  = isInfixOf (strip ss1) (fetchHref as)
```

For any other combination of types or values, the consistency function returns an error `undefined`.

```
cPX _ _ _ _ _ = error "undefined"
```

6.4.2. JSON to Python

For the references between JSON data and Python we have only one case to consider. Any JSON Object is parsed to a Python dictionary, thus the keys in the JSON objects are used in Python as strings literals. After fixing the type families in the function's signature and the choice of type representation in the pattern, the consistency function for this case boils down to equality on strings.

```
cPJ :: CompareId
    → Python.PYTHON ix1
    → Json.JSON ix2
    → ix1
    → ix2
```

```
    → Bool
cPJ _
    Python.TheExprSpan
    Json.TheKey
    (Python.Strings [ss1] _)
    (Json.JKey _ ss2)
    = strip ss1 == ss2
cPJ _ _ _ _ _ = error "undefined"
```

6.4.3. XHTML to JSON

The references between XHTML and JSON are the most prominent case. And again, once we have fixed the type families and type representations, the consistency function `cJX` for this case is just equality on strings.

```
cJX :: CompareId
    → Json.JSON ix1
    → Xml.XML ix2
    → ix1
    → ix2
    → Bool

cJX _
    Json.TheKey
    Xml.TheContent
    (Json.JKey _ ss1)
    (Xml.CString _ ss2 _)
    = ss1 == strip ss2

cJX _ _ _ _ _ = error "undefined"
```

6.4.4. JSON to JSON

The consistency for of two JSON objects that model a dictionary is not clear. There might be different interpretations, depending on the context. In our context, we require that they have to have the same set of keys. Because this is not obvious, we use a compare-id to give a name to this kind of consistency. The consistency it self is implemented using some standard functions from Haskell's data structures for dictionaries and sets (`Data.Map` and `Data.Set`).

```
cJJ :: CompareId
    → Json.JSON ix1
    → Json.JSON ix2
    → ix1
    → ix2
    → Bool
```

```
cJJ (CompareId "samekeys")
    Json.TheJson Json.TheJson
    (Json.JObject _ dict1) (Json.JObject _ dict2)
    = ( keysSet $ Map.fromList $ dict1)
    == (keysSet $ Map.fromList $ dict2)

cJJ _ _ _ _ _ = error "undefined"
```

6.5. The crucial function `check`

The main interface between the generic framework and these context-specific consistency functions is implemented by a specific consistency module (cf. the main architecture in Section 5.2). The consistency module exports a function called `check` that determines from the file name extensions which parser and consistency function should be called before delegating the corresponding consistency function to the function `plc` in the generic-framework which in turn performs the zipper-based address lookup of the references and then checks for consistency.

```
check :: Relation → IO Bool
check r@(Relation (R.Elem (DPath pf1) fp1)
                  (R.Elem (DPath pf2) fp2)
                   cf
        ) = do
  let ext1 = takeExtension fp1
  let ext2 = takeExtension fp2
  case (ext1,ext2) of
    (".py",".xhtml")
      → plc (TheModuleSpan) (TheDocument)   pP pX (cPX cf) r
    (".py",".json")
      → plc (TheModuleSpan) (TheJson)       pP pJ (cPJ cf) r
    (".json",".xhtml")
      → plc (TheJson)       (TheDocument)   pJ pX (cJX cf) r
    (".json",".json")
      → plc (TheJson)       (TheJson)       pJ pJ (cJJ cf) r
    (lext,rext)
      → error $ "comparing filetypes "
              ++ lext++" and "++rext
              ++" is currently not supported. exiting"
```

Based on the pairs of file name extensions, we fix the type representations for the roots of the underlying abstract syntax trees and the parsing functions (here we use a similar naming scheme as for the consistency functions: `pP` stands for "parse Python" etc. ...). We can also see how the compare-id `cf`, which is a part of the Relation value `r`, is passed to the specific consistency functions. In case we have not defined a consistency function for the incoming pair of extensions, we abort with an error.

6.6. A Context Specific Reference Manager

Once we have defined the parsers, multirec-adaptors and consistency module, we can simply link these together with the front-end and the generic library discussed in the preceding chapter to get a reference editor and consistency checker for the specific context of our case study:

```
module Main where

import Language.Dependencies.Main        -- the generic front-end
import Consistency as C                  -- the consistency module

caseoptions :: Options                   -- options for the editor
caseoptions = Options
            "Monospace"
            9
            [ CompareId "dynvar"        -- CompareIds for the menu
            , CompareId "httpvar"
            , CompareId "samekeys"]
            C.check                          -- the check function


--  main entry point.  call  main' from the generic front-end and
--  configure it according to the current context.
main :: IO ()
main =  main' caseoptions C.languages
```

This tool can then be used to define the references interactively by selecting the respective portions of the source code, and perhaps choosing the wanted consistency function by its compare-id to make an implicit reference explicit. Figure 6.3 shows the tool in action. We have selected the string `pages` in the Python application and in the JSON-based registry. The tree-components at the bottom show the positions in the syntax trees, on the right of the screen we see the list of references.

Once a set of references is made explicit, we can use the tool also non-interactively to check if all the references we made explicit are still consistent.

6.7. Summary

In this chapter we have conducted a small case study. We developed a template system for the web that separates data processing from data and their appearance using different languages. The implicit references between these languages are given by named variables, which occur in one form or the other throughout the source code.

Using our framework, we defined *type-safe* consistency functions for the relevant cases. Since type-safety is generally considered as a benefit, this is an advantage in contrast to a home grown universal data structure for terms, because we reuse the strength of our host programming language Haskell.

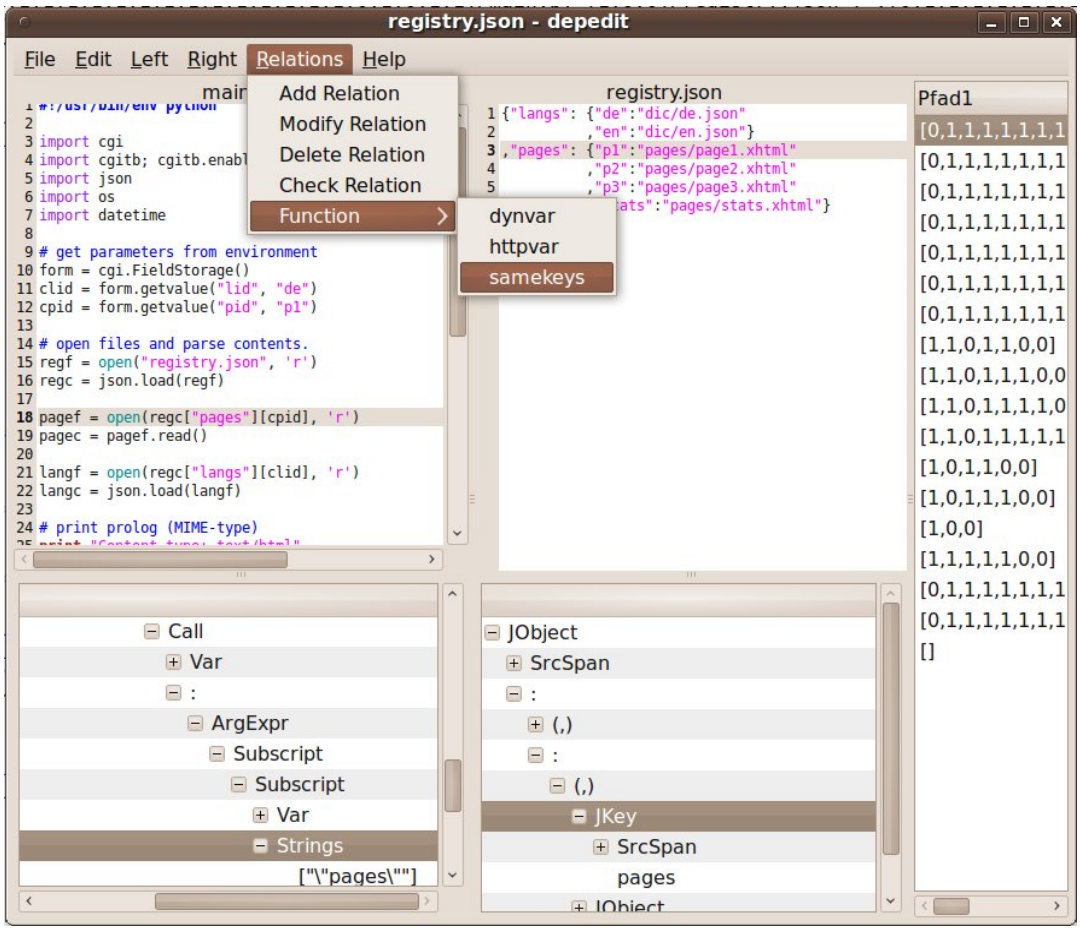


Figure 6.3.: Front-end, adapted to the case study.

Our case study showed that the concept does not only work for the toy languages we used in the preceding chapters but also for real-life languages and real-life applications.

Due to the underlying generic framework, it is quite easy to develop a context dependent notion of consistency that works across many languages.

Part III.

Final Notes

7. Conclusions

7.1. General Summary

We have presented a framework to make ubiquitous low level references between arbitrary constructs in source code given in arbitrary languages *explicit*. While the problems that arise due to these implicit interlingual references are well-known to practitioners, there is no adequate tool-based solution up to today. The reason is, that such a tool needs to be capable to analyze source code in many languages and that the choice of these languages is subject to the specific requirements of a project. Thus the tool has to be parametric in the languages themselves.

While the theoretical foundations we build up upon in this respect are not new, suitable implementations and libraries did only appear in recent years. The concept of *datatype generic programming*, developed in the functional programming community, builds up on ideas from category theory and there are working implementations especially in the Haskell-community. This approach finally allows to write type-safe software engineering tools that can be reused for (i.e. parametrized by) many languages.

We have analyzed the implicit interlingual references and found that they occur in a lot of situations and that they are in fact the remnants from the decomposition of a software project into different parts implemented in specialized languages. Formally, we have defined them as links between specific subtrees in abstract syntax trees. The notion of consistency for such a pair is then the definition of a function that maps two subterms to a Boolean value.

We have developed a framework that allows to manage these references, i.e. we can define, check and adapt them (provided that a input coming from a suitable structured diff is available). Our prototype also delivers a graphical front-end tool that can be used to perform the actions interactively. In the design of the framework and the tool we have taken great care to separate generic data structures and functionalities from accidental issues like concrete languages and consistency functions. The Haskell Platform serves us as a common execution engine that allows for a formal treatment of the underlying theory and as a practical programming language for everyday use at the same time.

To apply the generic framework to specific languages, users only need to supply quite simple adaptors that map the grammar of the languages to the functor representation. We have shown how this works in Section 3.3 and applied it to languages like Python and JSON in the case study in Chapter 6.

We have performed a case study that proved that our approach works for real life languages. The application under consideration was a toy program, but the implanted references and the employed languages do occur in real life projects.

7.2. Results and Retrospective

While the single results we achieved are not revolutionary, we will embed them now into a bigger picture to highlight our contribution in the field of tension between theory and application mentioned in the Preface.

A theme that often reoccurs in scientific software engineering is abstraction: we seek for solutions that are independent of application specific context. But software engineering is about *engineering*, thus there are real-life problems in real-life applications that have to be solved. That means we have to identify a practical problem, abstract from everything unnecessary, find a solution, and bring that solution back into practice.

This is quite a long way, and especially the last step is often overseen. In our case, the practical problem is well known among practitioners. At the same time, the abstract theories of programming languages and the relations to the even more abstract realms of algebra and category theory are well known to computer scientists and mathematicians for a long time (the fixed point result of Lambek dates back to 1968 [Lam68]).

We started with the problem of inconsistencies between artifacts of a different kind. The problem was that the underlying references are *interlingual*, thus we were in need of a consistent formal framework to formulate the problem. The choice we made was to express the underlying artifacts as terms that are typed with some *algebraic* datatype. Since we always had a practical prototype in mind to demonstrate the usefulness of the approach, we also decided to use Haskell has our implementation language: it has both algebraic datatypes and a lot of parsers and general infrastructure. To argue about references between terms of arbitrary algebraic datatypes, we needed an accessible specification of the signatures themselves. Formally this specification of specifications can be expressed using category theory: The notion of a functor that specifies the structure of a datatype is central in this respect. We found the according implementation in Haskell under the term *datatype generic programming* and decided to use this as the technical basis of the prototype. We developed the prototype using two simplistic toy languages. Finally we performed a case study to prove that the approach works with real-life languages and real-life problems and could ultimately get a grip on the original problem.

So our contribution is not only the development of a framework that solves a known problem in a quite complicated way (though we are not aware of other more promising solutions). We would like to emphasize that the rigor in which we look at software artifacts as typed terms with all the theory in background is a promising approach to software engineering in general. The possibility to develop approaches that are independent of the languages under consideration but still tackle the essential problems is quite inspiring. We are eager to see upcoming works and tools that will employ this approach.

7.3. Outlook and Open Ends

An issue we ignored completely in this work is a more elaborated error handling. In fact our library is unsafe in that we do not handle errors properly. On the one hand this

would be easy to accomplish using standard approaches (again, the category theoretic concept of *sum* comes to mind: "either return a result or a typed error value") on the other hand it costs some effort so we left it out. For the Boolean-valued consistency functions a more elaborated error type would be appropriate, e.g. one that differentiates between different severities and provides meaningful messages.

Another point of critic might be the availability of adaptors to the generic world. In fact, we had to develop all the adaptors ourselves (see Appendix A). On the other hand they are easy to generate with the help of the available Template Haskell tools.

The mapping between source code locations and nodes of the AST, was a challenge for the development. The standard approach is to add a source location type to the language, and decorate every node with a sibling of that type to hold the needed information. Interestingly this is another generic problem, and during this thesis, van Steenbergen published an approach to the generic formulation of such meta data in [VSMaJ10].

The missing piece to get the adaption of references to work is the availability of a generic structured diff algorithm from which the atomic composable operations *insert* and *delete* on paths can be derived. This is definitely a quite interesting problem to look at in future.

Finally, one could argue about the necessity to use such a complicated approach. We say yes, because we generally believe in strong, static typing on the one hand and in reusability on the other hand. These stand in conflict in respect to programming languages. The employed approach resolves this conflict based upon deep theoretic works. We close this work finding our opening quote of Kurt Lewin confirmed:

„Es gibt nichts Praktischeres als eine gute Theorie."

Part IV.

Appendix: Source code

A. Published Source Code

We published the source code and the multirec-adaptors under an open-source license to a publicly available repository. The repository consists of a number of packages for which we will give a short overview here. The main architecture is according to Figure 5.1 on page 68.

Everything can be built using GHC 6.12.1 and the standard procedures of the cabal build tool. The repository is available from: `https://depedit.googlecode.com`[1] Additional technical details can be found in the corresponding `README` or `INSTALL` files in the repository.

depedit

This package contains the main framework together with the graphical frontend and the toy-languages from Chapters 4 and 5. It also contains some example files to experiment with the framework quickly.

multirec-adaptors

We developed a number of adaptors to the multirec library for the languages used in Chapter 6. The languages that are supported are: Python, JSON, XHTML and Java. For the latter we also provide suitable parsers.

casestudy

The `casestudy` links together the depedit-framework with the multirec-adaptors and contains the consistency module sketched in Chapter 6.

web-app

The folder `web-app` contains the toy web-application developed in Section 6.2. It also contains a `Rulefile` to be used by the casestudy-tool.

[1] Browse the sources here: `http://code.google.com/p/depedit/source/browse/#svn/trunk`

B. Haskell Convenience

For the sake of selfcontainedness, we include some frequently used Haskell functions in this appendix. The functions are adapted for presentation and stem from standard modules available online.

B.1. Data.Function

```
--  Application operator.  This operator is redundant, since
--  ordinary application @(f x)@ means the same as (f '$' x).
--  However, '$' has low, right-associative binding precedence,
--  so it sometimes allows parentheses to be omitted:
--
--  > f $ g $ h x = f (g (h x))
--

($)    :: (a → b) → a → b
f $ x  =  f x
```

B.2. Data.List

```
-- deletes the first occurrence of an element in a list.

delete :: (Eq a) => a → [a] → [a]
delete _ []     = []
delete x (y:ys) = if (x == y) then ys else y : delete x ys

-- | list difference ((non-associative).
-- In the result of @xs@ '\\' @ys@, the first occurrence of each
-- element of @ys@ in turn (if any) has been removed from @xs@.
--
-- > (xs ++ ys) \\ xs == ys.

(\\) :: (Eq a) => [a] → [a] → [a]
(\\) =  foldl (flip delete)
```

B.3. Folding

Folding from the left can be best exemplified with the summation of a list of integers. A binary operation `acc` for list elements is given that maps to a type `b`, the neutral element

of this operation is given to be used for an empty list. Then the binary operation is applied recursively to the front of the list for all elements:

```
foldl :: (a → a → b) → b → [a] → b
foldl acc e a:as = (acc a (foldl acc e as))
foldl acc e [] = e

sum :: [Int] → Int
sum as = foldl (+) 0 as
```

B.4. Data.Maybe

The Maybe datatype is often used to model possible failures as a constant `Nothing`. As long as there is only one kind of failure this suffices, in other cases, the approach can be extended to the type `Either a b`.

```
data Maybe a = Nothing | Just a

-- This is an unsafe error-handler. If a function returns a
-- 'Nothing' (e.g. the zipper navigations), function fromJust
-- will throw a runtime error. Otherwise, the value wrapped
-- by Just is returned.
fromJust :: Maybe a → a
fromJust Nothing = error "Maybe.fromJust: Nothing"
fromJust Just a = a
```

B.5. Control.Monad

```
-- | Left-to-right Kleisli composition of monads.
(>=>)       :: Monad m => (a → m b) → (b → m c) → (a → m c)
f >=> g     = \x → f x >>= g
```

Since the Maybe datatype is a monad, this function is often used for the composition of functions with type `a → Maybe b` and `b → Maybe c` to a type `a → Maybe c`. As long as only Just-values are involved, normal function composition occurs. When one function returns a Nothing, this is propagated to the end.

C. A XHTML Template System written Python and JSON

This appendix presents data files and templates that can are used with the template engine presented in Chapter 6.

C.1. Dictionaries

Registry

```
{"langs": {"de":"dic/de.json"
          ,"en":"dic/en.json"}
,"pages": {"p1":"pages/page1.xhtml"
          ,"p2":"pages/page2.xhtml"
          ,"p3":"pages/page3.xhtml"
          ,"stats":"pages/stats.xhtml"}  }
```

English Dictionary

```
{"title1":"Title of page one"
,"title2":"Title of page two"
,"title3":"Title of page three"
,"titlestats":"Dynamic Data"
,"text1":"An english text ..."
,"text21":"First section ..."
,"text22":"Second Section ..."
,"text3":"A third english example text ..." }
```

German Dictionary

```
{"title1":"Titel der ersten Seite"
,"title2":"Titel der zweiten Seite"
,"title3":"Titel der dritten Seite"
,"titlestats":"Dynamische Daten"
,"text1":"Ein deutscher Text ..."
,"text21":"Erster Abschnitt ..."
,"text22":"Zweiter Abschnitt ..."
,"text3":"Ein dritter deutscher Text ..."  }
```

C.2. Templates

A page with a navigation bar and localized contents

```
<html>

<head>
  <link rel="stylesheet"
        type="text/css"
        href="/style.css"/>
  <title>{title1}</title>
</head>

<body>

<div id="nav">
  <h4>Pages:</h4>
  <ul>
    <li><a href="main.py?pid=p1&amp;lid=de">home</a></li>
    <li>{p1}</li>
    <li>{p2}</li>
    <li>{p3}</li>
    <li>{stats}</li>
  </ul>

  <h4>Languages</h4>
  <ul>
    <li>{de}</li>
    <li>{en}</li>
    </ul>
</div>

<div id="content">
  <h1>{title1}</h1>
  <p>{text1}</p>
</div>

</body>

</html>
```

A page with dynamic content

```
<html>

<head>
  <link rel="stylesheet"
        type="text/css"
        href="/style.css"/>
  <title>{titlestats}</title>
</head>

<body>

<div id="nav">
  <h4>Pages:</h4>
  <ul>
    <li><a href="main.py?pid=p1&amp;lid=de">home</a></li>
    <li>{p1}</li>
    <li>{p2}</li>
    <li>{p3}</li>
    <li>{stats}</li>
  </ul>

  <h4>Languages</h4>
  <ul>
    <li>{de}</li>
    <li>{en}</li>
  </ul>
</div>

<div id="content">
  <h1>{titlestats}</h1>
  <!--   A table containing dynamic data   -->
  <table border="1px">
    <tr><td><b>key</b></td><td><b>value</b></td></tr>
    <tr><td>cpid</td><td>{cpid}</td></tr>
    <tr><td>clid</td><td>{clid}</td></tr>
    <tr><td>date</td><td>{date}</td></tr>
    <tr><td>HTTP_USER_AGENT</td><td>{HTTP_USER_AGENT}</td></tr>
  </table>
</div>

</body>
</html>
```

Bibliography

A note on web addresses in the bibliography: To access possibly outdated web pages in the long term, the Internet Archive at http://www.archive.org is recommended.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, second edition, August 2006.

[App97] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997.

[BCD+89] Patrick Borras, Dominique Clement, Th. Despeyroux, Janet Incerpi, Gilles. Kahn, Bernard Lang, and V. Pascual. CENTAUR: the system. *SIGPLAN Notices*, 24(2):14–24, 1989.

[Ben86] Jon Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, 1986.

[BGVDV92] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan Language-Based Editing System. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, 1992.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2nd edition, May 1998.

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming — An Introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[BMS80] Rod M. Burstall, David B. MacQueen, and Donald Sannella. Hope: An experimental applicative language. In *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143, New York, NY, USA, 1980. ACM.

[Bro87] Frederick P. Brooks, Jr. No Silver Bullet — Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[Bub96] Andreas Bubolz. Generierung eines syntaxgesteuerten Editors für ProSet mit dem Synthesizer Generator. Diplomarbeit, Lehrstuhl 10, Fachbereich Informatik, Universität Dortmund, 1996.

[BW88] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[BW95] Michael Barr and Charles Wells. *Category theory for computing science*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, second edition, 1995.

[CGM97] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful Change Detection in Structured Data. *SIGMOD Record*, 26:26–37, June 1997.

[Cre97] Roger F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 229–242, Berkeley, CA, USA, 1997. USENIX Association.

[Cro06] Douglas Crockford. Introducing JSON. http://json.org, 2006.

[DM05] Chris Dornan and Simon Marlow. Alex: A Lexical Analyser Generator For Haskell. http://www.haskell.org/alex, 2005.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[Fow07] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, 2007.

[GN98] Carlo Ghezzi and Bashar Nuseibeh. Guest Editorial: Introduction to the Special Section. *IEEE Transactions on Software Engineering*, 24(11):906–907, 1998.

[GN99] Carlo Ghezzi and Bashar Nuseibeh. Guest Editorial: Introduction to the Special Section. *IEEE Transactions on Software Engineering*, 25(6):782–783, 1999.

[GTWW75] Jopseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Abstract data types as initial algebras and correctness of data representations. In *Proc. Conf. on Computer Graphics, Pattern Recognition and Data Structure*, page 89–93, 1975.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy With Class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[HJL07] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing Approaches to Generic Programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming 2006*, volume 4719 of *LNCS*. Springer, 2007.

[Hue97] Gerard Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[IBMC06] IBM International Business Machines Corp. Eclipse Platform Technical Overview. http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf, 2006.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[Kah99] Wolfram Kahl. The Term Graph Programming System HOPS. In Rudolf Berghammer and Yassine Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 136–149, Wien, 1999. Springer-Verlag.

[Kön09] Alexander Königs. *Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation*. Dissertation, Technische Universität Darmstadt, January 2009.

[Lam68] Joachim Lambek. A Fixpoint Theorem for Complete Categories. *Mathematische Zeitschrift*, 103(2):151–161, April 1968.

[Lam89] Joachim Lambek. Fixpoints revisited. In Albert Meyer and Michael Taitslin, editors, *Logic at Botik '89*, volume 363 of *Lecture Notes in Computer Science*, pages 200–207. Springer Berlin / Heidelberg, 1989.

[Lan85] Bernard Lang. Mentor – Design and Implementation of the Kernel of a Program Manipulation System. In J. McDermid, editor, *Integrated project support environments*, volume 1, pages 175–188. IEEE Software Engineering Series, 1985.

[LLL09] Eelco Lempsink, Sean Leather, and Andres Löh. Type-Safe Diff for Families of Datatypes. In *WGP '09: Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, pages 61–72, New York, NY, USA, 2009. ACM.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators For The Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001. http://legacy.cs.uu.nl/daan/parsec.html.

[Löh04] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, September 2004.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[LP05] Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate with Class: Extensible Generic Functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.

[Mal90] Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Department of Computing Science, Groningen University, 1990.

[MG09] Simon Marlow and Andy Gill. Happy: The Parser Generator for Haskell. http://www.haskell.org/happy, 2009.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[NER01] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, 58:171–180, 2001.

[Nog07] Pablo Nogueira. When is an abstract data type a functor? In Henrik Nilsson, editor, *Trends in Functional Programming*, volume 7, pages 217–231. Intellect, 2007.

[NRH+08] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A Lightweight Approach to Datatype-Generic Rewriting. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming*, pages 13–24, New York, NY, USA, 2008. ACM.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *In SOOPPA*, 1990.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[PWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[Ren03] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2003.

[RJJ+08] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno. Comparing Libraries for Generic Programming in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM.

[RT84] Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19, pages 42–48, New York, NY, USA, May 1984. ACM.

[Rut96] Jan J.M.M. Rutten. Universal coalgebra: a theory of systems. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1996.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer Verlag.

[SE03] Mehrdad Sabetzadeh and Steve Easterbrook. Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, page 12, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[Sim99] Charles Simonyi. Hungarian Notation. http://msdn2.microsoft.com/en-us/library/aa260976(VS.60).aspx, 1999.

[SPJ02] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[SPJSV09] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and Decidable Type Inference for GADTs. *SIGPLAN Not.*, 44(9):341–352, 2009.

[SWZ99] Andy Schürr, Andreas Winter, and Albert Zündorf. PROGRES: Language and Environment. In H. Ehrig, G. Engels, H. Kreowski, and G. Rozenberg, editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2, pages 487–550. World Scientific, Singapur, 1999.

[Tae04] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin / Heidelberg, 2004.

[TR81] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[UA91] Computer Systems Laboratory (U.S.) and European Computer Manufacturers Association. *Reference model for frameworks of software engineering environments*. U.S. Dept. of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, second edition, 1991.

[VSMaJ10] Martijn Van Steenbergen, José Pedro Magalhães, and Johan Jeuring. Generic selections of subexpressions. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 37–48, New York, NY, USA, 2010. ACM.

[Wad90a] Philip Wadler. Comprehending Monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM.

[Wad90b] Philip Wadler. Recursive types for free! `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`, 1990.

[Wal09] Malcolm Wallace. Polyparse. `http://hackage.haskell.org/package/polyparse`, 2009.

[Wir90] Martin Wirsing. Algebraic Specification. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 675–788. Elsevier Science Publishers B. V., 1990.

[WR99] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? *ACM SIGPLAN Notices*, 34:148–159, September 1999.

[Yak09] Alexey Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. PhD thesis, Utrecht University, 2009.

[YHLJ09] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 233–244. ACM, 2009.