Philipps Universität Marburg

Department of Mathematics and Computer Science
Programming Languages and Software Technology Group

Master Thesis

A Type System for Program Transformations based on Parametric Tree Grammars

Katharina Haselhorst

Supervisors:
Sebastian Erdweg, M.Sc.
Prof. Dr. Klaus Ostermann

September 2012

Abstract

Program transformations are an important area of metaprogramming. Compilers and preprocessors commonly apply multiple program transformations in a row, using the output of one transformation as input for the next. Thereby, each transformation relies on the output of the preceding transformation to be correct. For example, a C compiler depends on the preprocessor to expand all macro definitions as to be able to process the input. Syntactic extensibility of programming languages exposes the implementation of program transformations to the user, which makes it even more important to specify and check their input and output behavior.

Current general-purpose program transformation languages are usually untyped, hence they cannot ensure that the composition of multiple transformations is safe, i.e. does not lead to a runtime error. While there is research in designing type-safe program transformation languages for specific domains like XML, the safety properties of general-purpose program transformation languages have not yet been thoroughly studied.

To this end, we develop a core calculus for program transformation languages suited to formally explore the safety properties of program transformations. On top of that calculus, we design a domain-specific type system based on regular tree grammars. The idea is to use tree grammars as types describing the abstract syntax of the manipulated programs. This makes it possible to specify the input and output language of a transformation and hence to check the validity of composite transformations. We show that the type system is expressive enough to ensure the safety of program transformations in our calculus and formally prove its soundness.

Furthermore, we extend our calculus and the grammar-based type system with higher-order functions and parametric types. We believe that the resulting language constitutes a solid basis for further research on program transformation languages.

Zusammenfassung

Programmtransformationen sind ein wichtiger Anwendungsbereich der Metaprogrammierung. Compiler und Präprozessoren wenden üblicherweise mehrere aufeinanderfolgende Transformationen an, deren Ausgaben die Eingabe für die jeweils nächste Transformation darstellen. Transformationen benötigen korrekte Eingabedaten, weshalb sichergestellt sein muss, dass die Ausgabedaten der jeweils vorhergehenden Transformation gültig sind. Beispielsweise kann der C-Compiler das übergebene Programm erst verarbeiten, wenn Makro-Definitionen im Quelltext vom Präprozessor expandiert worden sind. Im Falle von syntaktisch erweiterbaren Programmiersprachen implememtieren die Benutzer diese Programmtransformationen, weshalb es umso wichtiger ist, dass deren Verhalten bezüglich Eingabe- und Ausgabedaten spezifiziert und überprüft werden kann.

Aktuell verwendete universelle Programmtransformationssprachen sind meistens ungetypt, weshalb nicht garantiert werden kann, dass die Komposition mehrerer Transformationen sicher ist, also nicht zu Laufzeitfehlern führt. Es gibt Forschungsarbeiten im Bereich typsicherer Transformationssprachen, jedoch lediglich für spezielle Domänen wie z.B. XML. Für universelle Programmtransformationssprachen wurden die Sicherheits-Eigenschaften allerdings noch nicht umfassend untersucht.

Wir entwickeln ein Basiskalkül für Programmtransformationssprachen, welches es uns erlaubt, die Sicherheits-Eigenschaften von Programmtransformationen formal zu untersuchen. Von diesem Kalkül ausgehend entwickeln wir ein domänenspezifisches Typsystem, welches auf regulären Baumgrammatiken basiert. Wir verfolgen den Ansatz, Baumgrammatiken als Typen zur Beschreibung der abstrakten Syntax der zu transformierenden Programme einzusetzen. Hierdurch können die Eingabe- und Ausgabesprachen der Transformationen spezifiziert und somit die Gültigkeit der zusammengesetzten Transformationen überprüft werden. Wir zeigen, dass das daraus resultierende Typsystem ausdrucksstark genug ist, um die Sicherheit von in unserem Kalkül ausgedrückten Programmtransformationen zu gewährleisten. Weiterhin beweisen wir dessen *soundness* formal.

Außerdem erweitern wir unser Kalkül und das grammatikbasierte Typsystem um Funktionen höherer Ordnung und parametrische Typen. Wir glauben, dass die resultierende Sprache als eine solide Basis für weitere Forschungsarbeiten im Bereich der Programmtransformationssprachen dienen kann.

Contents

1 Introduction

Programming is all about the manipulation of data. The kind of data is application specific: a text editor works on text files, a database engine accesses and manipulates database entries, an internet browser retrieves and renders HTML pages, etc. When the data a program works on is itself a program, the outer program is called a *metaprogram*. Hence, metaprograms can be thought of as functions from programs to programs. They get a program as input and produce another program as output.

This thesis deals with a specific area of metaprogramming, namely *program transformations*. In program transformations, a program is commonly represented by its abstract syntax tree. Hence the metaprograms consist of tree transformations. Important applications of program transformations can be found for example in compilers and preprocessors. The main task of a compiler is to translate a program written in one programming language into a program in another programming language. Compilers also often optimize a given program, i.e. according to some metric they produce an equivalent but better program in the same programming language.

Another important application for program transformations are *desugarings*. On top of a small core language, many programming languages offer a rich surface syntax, so that the language is convenient to use. The advantage of a small core language is that formal reasoning and the implementation of the language get easier. The surface syntax is defined in terms of the core language. This kind of syntax is called *syntactic sugar* and the process of translating syntactic sugar back into the core language is called desugaring. Desugaring is a common task of preprocessors.

One well-known example for syntactic sugar is Haskell's do notation [9]. The do notation allows a programmer to express sequences of actions in an intuitive way. The following program reads one line of input from the user and prints it out again.

```
main :: IO ()
main = do line <- getLine
          putStrLn line
```

This program desugars to the following—more verbose—version with higher order functions.

```
main :: IO ()
main = getLine >>= \line ->
         putStrLn line
```

Traditionally, defining syntactic sugar for a programming language is up to the language implementors. Hence, the process of desugaring is built into the compiler or interpreter. Recent research explores the possibility to transfer this task to the *user* of the language [8, 9]. This means a programmer can define his own syntactic sugar by specifying desugarings into the core language. SugarJ [8] is a framework offering syntactic extensibility for the programming language Java. For example, we can extend Java with a syntactic construct for pairs. This enables us to write a program of the following form.

```
public class Test {
  private (String, Integer) p = ("The answer is", 42);
}
```

The Java compiler does not know how to deal with this program, because we use new syntactic construct which are not part of the Java language. Therefore, a preprossesor first needs to desugar the program into plain Java before passing the resulting program to the compiler. The result of the desugaring might look as follows, depending on our concrete specification of the new syntax.

```
public class Test {
  private Pair<String, Integer> p =
    new Pair("The answer is", 42);
}
```

SugarJ-like syntactic extensions can also be *composed*. This means we can apply multiple different desugarings in a row each of them taking the output of the previous desugaring as input.

The main problem, as with all areas of programming, is that programmers make errors. There are different kinds of errors. First, executing an erroneous program can lead to a runtime error. This means the program does not produce any result or a wrong result. Wrong results may include unexpected side effects like formatting the hard drive. Another kind of error arises when multiple pieces of code are composed: the first piece of code might produce an output unsuitable as input for the second piece of code. For example, composing a function that produces string values with a function that expects integer values as input will most likely lead to a runtime error. In one of the previous examples we showed how to extend the Java syntax in SugarJ by adding syntactic sugar for pairs. In this case, the result of the desugaring serves as input to the Java compiler. Hence, it is crucial to ensure that the result of the desugaring is valid Java code.

In the context of program transformations the data itself is a program that will be executed. It follows that the problem of errors not only applies to the metaprogram itself but also to the program we are manipulating—the *embedded* program. The result of a program transformation could be a program that is not syntactically correct in terms of the embedded language or that will produce a runtime error when executed.

In a nutshell, there are three desirable properties that one could expect from a metaprogram:

1. The metaprogram produces a result. This means it does not produce a runtime error when executed.

2. The result is a correct program in terms of the syntax of the target programming language.

3. The result has some desirable properties in terms of the semantics of the target programming language (for example it produces a correct result when executed).

From the point of view of safety it makes sense to catch errors *before* the program is executed. This applies to metaprograms as well as to embedded programs. Current term transformation languages like Stratego [31] or PLT Redex [20] offer a rich set of tools for working on embedded programs. However, they are not able to statically ensure the validity of the transformations. This means that a programmer can not be sure that the result of a transformation yields a valid program in the target language.

In current research there are two different approaches to solve the problem. The first approach is to use general purpose languages and their type systems to express statically type-safe program transformations. The solutions usually suffer from the limitations in expressiveness of the general purpose type systems. The second approach is to design domain specific languages for program transformations. One example is the programming language XDuce for expressing type-safe transformations on XML programs [14]. In Section 11 we discuss these two approaches.

While there exists research in the area of typed program transformation language, the work applies only to a certain domain of programs as for example XML. This introduces many problems specific to the respective domain. However, we don't know of any framework which is suited to explore general properties of program transformation languages independent of the concrete application domain.

The goal of this thesis is to fill this gap by designing a core calculus as a basis for formally studying safety properties of program transformations. The core idea is to use a domain specific type system based on tree grammars. Tree grammars are a formalism to describe sets of tree structured data. Since the embedded programs are represented as abstract syntax trees, tree grammars are well suited to serve as types for our domain specific type system. To get back to our previous example, we can define a type `Java+Pairs` and another type `Java` representing the Java syntax extended with pairs and the core Java syntax respectively. The type system has to prove that the desugaring produces a result of type `Java` when given an input of type `Java+Pairs`.

Summarized, our contributions are as follows.

- We define a minimal set of language constructs which are necessary to express reasonably complex program transformations.
- Based on these language constructs we design a core calculus for program transformation languages by specifying the syntax and an operational semantics.
- We design a domain specific type system based on regular tree grammars and prove it sound.

- We extend the core calculus and the type system by higher order functions and parametric types.
- With two case studies we show the applicability of our calculus for complex program transformations.
- We provide a prototype implementation of the system on top of PLT Redex [20].

Outline

The reminder of this thesis is structured as follows.

In Section 2 we define a core calculus for metaprogramming. The goal is to design a calculus that is expressive enough to describe complex program transformation while still keeping it as small as possible as to simplify the study of its formal properties. The main features of this calculus are recursive function definitions and pattern matching as a means to deconstruct and analyze tree structured data.

In Section 3 we introduce the notion of regular tree grammars and discuss how to use them as types in a type system for metaprogramming.

In Sections 4 to 6 we define a base type system for the calculus from Section 2 yielding a simply typed language. The main challenge turns out to be the typing of pattern matching expressions. We discuss what it means for a pattern to match a type and how to decide if a set of pattern is exhaustive with respect to a type. Furthermore, we prove the soundness of the resulting system and describe a type checking algorithm on the basis of type inference. Finally, we demonstrate the ability of the simply typed language to express complex program transformations by a case study. The case study consists of writing an evaluator for a small programming language. While the case study shows that the simply typed language can express complex program transformations it also shows some limitations in terms of abstraction mechanisms. These limitations motivate further extensions of both the language and its type system.

In Sections 7 and 9 we discuss how to extend the simply typed language with higher order functions and type abstractions.

Although the main focus of the thesis is to study the formal properties of the calculus for metaprogramming, we provide a proof-of-concept implementation of the simply typed language on top of PLT Redex [20]. In Section 10 we discuss some of the algorithms used in the implementation, in particular the algorithms operating on tree grammars like subtyping. The source code of the implementation is available at `https://github.com/haselhorst/Tree-Grammars`.

In Section 11 we discuss related work, in Section 13 we conclude and in Section 12 we point to areas of future research that arise from the work of this thesis.

2 A Calculus for Program Transformations

In this section we describe a calculus for program transformations. There are two design goals for this calculus:

- It has to be expressive enough to describe reasonably complex program transformations.
- It should be as small and simple as possible for the following reasons: First, the smaller the language, the easier it is to formally reason about it. And second, a small language is a good starting point for experimenting with language or type system extensions.

Since the notion "reasonably complex" is vague, we need to be more concrete about the kind of operations we expect to be expressible. The most basic operation is the construction and deconstruction of terms. Construction means we want to be able to combine terms into more complex terms. Deconstruction refers to "looking inside" a term, i.e., decomposing a complex term into its subterms. In addition to deconstructing a term, it should be possible to take different actions depending on the shape of the term.

The syntax trees we are working on are inductively defined. This implies recursive functions as an appropriate unit of abstraction. As to keep the calculus as simple as possible we only use first order functions here. Later (see Section 7) we extend the calculus with higher order functions.

In the following we formally define the calculus. This is done in two steps: First, we specify the syntax and second, we define the meaning of a syntactically correct program: its semantics.

2.1 Syntax

The syntax of language is specified by a context free grammar. The definition is shown in Figure 1. A program consists of a set of function definitions and one expression. There are four kinds of expressions covering the requirements we identified in the beginning:

- Constructor applications of the form $c\texttt{(}expr\texttt{,}\ldots\texttt{,}expr\texttt{)}$: with that we can construct a new expression given some subexpressions. We allow to omit the parentheses for a constructor application without arguments.
- Pattern matching: the goal of a pattern matching expression

  $$\texttt{match}\ expr_m\ \texttt{case}\ pat_1\ \texttt{->}\ expr_1 \ldots \texttt{case}\ pat_n\ \texttt{->}\ expr_n$$

  is to deconstruct the expression $expr_m$ with the help of the patterns pat_1 to pat_n and take different actions depending on the shape of $expr_{match}$. Hence, constructor application and pattern matching are dual to each other: a constructor application *constructs* terms, whereas pattern matching *deconstructs* them.

- Function applications of the form f`(`$expr$`,`$\ldots$`,`$expr$`)`: this applies the function f to the argument expressions.

- Variables x.

To be able to distinguish a function application from a constructor application we syntactically separate constructor names from function names: constructor names start with a capital letter, while function names start with a lower case letter. Since a function application can be always distinguished from a variable by the context (a function application is always followed by an opening parenthesis), names for variables also start with a lower case letter.

In function definitions we give names to functions. Since we can use the name of the function we are currently defining within the body of the definition, the language supports recursion.

Before giving some examples that illustrate the syntax of the language we introduce some convenience notations:

- We allow to give names to entities like function definitions, expressions, etc. and then to use the names in the following instead of inlining the definitions.

- Instead of writing a program as a list of function definitions and an expressions, we will often just write the expression and mean "that expression in the context of all already defined functions".

$$
\begin{array}{lcl}
\mathit{program} & ::= & F\ \mathit{expr} \\
F & ::= & \mathit{fdef} \ldots \mathit{fdef} \\
\mathit{fdef} & ::= & f\texttt{(}x\texttt{,}\ldots\texttt{,}x\texttt{)}\ \texttt{=}\ \mathit{expr} \\
\mathit{expr} & ::= & x \\
 & & \mid f\texttt{(}\mathit{expr}\texttt{,}\ldots\texttt{,}\mathit{expr}\texttt{)} \\
 & & \mid c\texttt{(}\mathit{expr}\texttt{,}\ldots\texttt{,}\mathit{expr}\texttt{)} \\
 & & \mid \texttt{match}\ \mathit{expr}\ \mathit{caseexpr} \ldots \mathit{caseexpr} \\
\mathit{caseexp} & ::= & \texttt{case}\ \mathit{pat}\ \texttt{->}\ \mathit{expr} \\
\mathit{pat} & ::= & x \mid c\texttt{(}\mathit{pat}\texttt{,}\ldots\texttt{,}\mathit{pat}\texttt{)} \\
x & ::= & \text{variable names} \\
c & ::= & \text{constructor names} \\
f & ::= & \text{function names}
\end{array}
$$

Figure 1: Syntax of the Language

Example 2.1
`Zero` and `A(B,C,A(X))` are examples of very simply programs that only use constructor applications to build literal trees.

Let's look at a slightly more complex program using pattern matching.

```
match A(B,C)
  case A(x,A(y,z)) -> y
  case A(x,y)      -> x
  case x           -> NoA
```

Although we have not yet defined the semantics of the language, we would expect that this program evaluates to `B`: for an intuitive definition of matching, the second pattern `A(x,y)` matches the expression `A(B,C)` thereby assigning the value `B` to `x`.

Example 2.2
As a second example we show two definitions of functions called `positive` and `pred`. Their intention should be pretty clear from the names.

```
positive(x) = match x
                case Zero -> False
                case y    -> True

pred(x) = match x
            case Succ(y) -> y
```

2.2 Semantics

So far, we have only defined the syntax of the language. Hence, we can say that a given piece of code is a syntactically valid program according to the syntax definition. But we did not yet specify what the meaning of the program is: its *semantics*.

We formalize the semantics of the language as a small step operational semantics [24] on expressions. This means, we define a reduction relation that reflects the way an expression e_1 can reduce to another expression e_2 in one step.

Formally, the reduction relation is a ternary relation $\longrightarrow \subseteq (F \times \mathit{expr} \times \mathit{expr})$, where F is a set of function definitions and *expr* is an expression.

$(F, e_1, e_2) \in \longrightarrow$ means that e_1 can reduce to e_2 in one step assuming the function definitions of F. We write $F \vdash e_1 \longrightarrow e_2$ for $(F, e_1, e_2) \in \longrightarrow$. Furthermore, we define $\longrightarrow^*$ to be the reflexive and transitive closure of $\longrightarrow$. This means that $F \vdash e \longrightarrow^* e'$ if and only if there exist e_1 to e_k with $e_1 = e$ and $e_k = e'$ so that $F \vdash e_1 \longrightarrow \ldots \longrightarrow e_k$.

To be able to define the reduction relation, we need some preliminary notions that we introduce in the following.

2.2.1 Values

We define the semantics of a program as a small step operational semantics describing single evaluation steps. Nevertheless, the ultimate goal of a program is to produce a result and not to be trapped in an endless sequence of evaluation steps. The language is a calculus for program transformations, hence the data we are working with are programs in the form of syntax trees. This means we expect the program to return a syntax tree when finishing evaluation.

Values of a language are those expressions that we do not expect to reduce further but would accept as the result of an evaluation. Since our data has the form of syntax trees, those should at least be a subset of the values. In this case it showed sufficient to define the values to coincide exactly with the trees. Hence, values are syntactically defined as the following subset of the expressions (which coincides with the definition of trees in Section 3).

$$\mathit{value} \quad ::= \quad c\texttt{(}\mathit{value}, \ldots, \mathit{value}\texttt{)}$$

Constructors without arguments are leafs of the trees, otherwise they are inner nodes.

Example 2.3
The following expressions are values:

```
Zero, Succ(Plus(Succ(Zero),Zero))
```

but these are not:

```
x, f(), Succ(f(Zero))
```

`x` and `f()` are no trees at all. `Succ(f(Zero))` resembles a tree, yet contains the function application `f(Zero)` as a subexpression. But values allow only constructor applications. Furthermore, we would expect the expression `Succ(f(Zero))` to evaluate further by replacing `f(Zero)` with the result of applying the function `f` the `(Zero)`.

2.2.2 Evaluation Contexts

As the expression `Succ(f(Zero))` from the last example shows, evaluation sometimes has to take place in a subexpression of the current expression. *Evaluation contexts* [32] have proved to be a useful formalism to capture the notion of evaluations in subexpressions. An evaluation context E is defined as follows.

$$\begin{array}{lcl} E & ::= & \bullet \\ & & \mid f\texttt{(}\mathit{value}, \ldots, \mathit{value}, E, \mathit{expr}, \ldots, \mathit{expr}\texttt{)} \\ & & \mid c\texttt{(}\mathit{value}, \ldots, \mathit{value}, E, \mathit{expr}, \ldots, \mathit{expr}\texttt{)} \\ & & \mid \texttt{match}\ E\ \mathit{caseexpr} \ldots \mathit{caseexpr} \end{array}$$

Evaluation contexts can be thought of as expressions with a hole (represented by $\bullet$). Plugging an expression e_1 into the hole of an evaluation context E yields an expression e_2. This operation is written $E[e_1]$ and replaces $\bullet$ in E with the expression e_1.

An expression e can be decomposed into an evaluation context E and an expression e' so that $E[e'] = e$. This decomposition is not necessarily unique. But we will see in Section 2.2.5, that there is at most one decomposition $E[e'] = e$ of an expression e so that e' is a reducible expression (redex) in terms of the reduction relation.

With the help of evaluation contexts the reduction of the subexpression e_1 within the expression e can be written as follows: First, e is decomposed into $E[e_1] = e$. If e_1 evaluates to e_2 we can simply plug the result back into e by $E[e_2]$.

Note, that the definition of evaluation contexts constrains the set of subexpressions that can be plugged out of an expression by a decomposition. Take for example the expression `f(g(x), g(y))`. It is possible to decompose it into $E =$ `f(●, g(y))` and $e =$ `g(y)`. But the decomposition $E =$ `f(g(x), ●)` and $e =$ `g(y)` is invalid, because `g(x)` is not a value. Hence, this definition of evaluation contexts implies a left-to-right evaluation of arguments in function- and constructor applications.

2.2.3 Substitution

When evaluating a function application we will have to substitute the formal arguments of the function by the actual arguments of the application. Reducing a pattern match has similar requirements: in that case we must substitute the pattern variables by their matches in the corresponding expression. This is where the general notion of a substitution comes into play.

A substitution σ is a finite mapping from variables into a target domain. In this chapter, the target domain will be the set of values.

Applying a substitution σ to an expression e yields a new expression where each variable $x \in \mathrm{dom}(\sigma)$ is replaced by its substitute $\sigma(x)$ under some conditions. We write $\sigma(e)$ or $e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ for applying a substitution $\sigma = (x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)$ to an expression e.

Since there are expressions (namely pattern matches) in the language that bind variables, it would be wrong to literally replace all variables with their substitutes. More precisely, applying a substitution σ to expression e only replaces all *free* occurrences of variables in e. All *bound* occurrences of variables are left untouched. An occurrence of a variable is bound if it is in the scope of a pattern binding it. Otherwise the occurrence if free.

Figure 2 shows the straightforward inductive definition of substitution. Note, that we do not have to think about accidental variable capture [24] here, because we only replace variables with values which cannot contain free variables.

Example 2.4

$$\begin{array}{lcl}
\sigma(x) & = & \begin{cases} \sigma(x) & \text{if } x \in \mathrm{dom}(\sigma) \\ x & \text{otherwise} \end{cases} \\
\sigma(C(e_1,\ldots,e_n)) & = & C(\sigma(e_1),\ldots,\sigma(e_n)) \\
\sigma(f(e_1,\ldots,e_n)) & = & f(\sigma(e_1),\ldots,\sigma(e_n)) \\
\sigma\left(\begin{array}{l} \texttt{match}\ e \\ \quad \texttt{case}\ \mathit{pat}_1\ \texttt{->}\ e_1 \\ \qquad \vdots \\ \quad \texttt{case}\ \mathit{pat}_n\ \texttt{->}\ e_n \end{array}\right) & = & \sigma\left(\begin{array}{l} \texttt{match}\ \sigma(e) \\ \quad \texttt{case}\ \mathit{pat}_1\ \texttt{->}\ \sigma_1(e_1) \\ \qquad \vdots \\ \quad \texttt{case}\ \mathit{pat}_n\ \texttt{->}\ \sigma_n(e_n) \end{array}\right) \\
 & & \text{where } \sigma_i = \sigma \setminus \mathrm{vars}(\mathit{pat}_i)
\end{array}$$

Figure 2: Applying a Substitution to an Expression

As an example we apply the substitution $\sigma = \{\texttt{x} \mapsto \texttt{Zero}\}$ to the expression

```
match Plus(Succ(x),Zero)
  case Plus(x,y) -> x
  case y         -> Plus(x,x)
```

For that we need to apply the substitution recursively to the subexpressions:

- $\sigma($`Plus(Succ(x),Zero)`$)$ yields `Plus(Succ(Zero),Zero)`.
- When recursing into the right hand sides of the case expressions we need to be careful. The first pattern `Plus(x,y)` binds the variable `x`. Therefore we apply the modified substitution $\sigma \setminus \{$x,y$\} = \{\}$ to the subexpression `x` yielding `x`.
- The second pattern does not bind `x`. Hence its right hand side becomes `Plus(Zero,Zero)` after the substitution.

The resulting expressions thus is

```
match Plus(Succ(Zero),Zero)
  case Plus(x,y) -> x
  case y         -> Plus(Zero,Zero)
```

2.2.4 Matching of Patterns against Values

In the formulation of the reduction relation we will need the notion of a pattern matching against a value. Informally spoken, a pattern matches a value if it is an (possibly incomplete) part of the value starting at the root node of the tree. Incomplete means, that at some points there may be variables instead of subtrees. Additionally, if a variable occurs more than once in the pattern, the corresponding subtrees in the value must be the same.

Formally, we define that a pattern *pat* matches a value v if there exists a substitution σ, so that $\sigma(\mathit{pat}) = v$. Obviously, there can be at most one substitution σ fulfilling this requirement.

Example 2.5
The substitution $\sigma = (x \mapsto$ `Zero` $, y \mapsto$ `Succ(Zero)`$)$ shows that the pattern $pat =$ `Succ(Plus(x,y))` matches the value $v =$ `Succ(Plus(Zero,Succ(Zero)))`, since $\sigma(pat) = v$.

Figure 3 show the (partial) function match that computes the substitution unifying a pattern with a value. The predicate "consistent" in the definition means that the results of the recursive calls have to agree on bindings for the same variable.

$$
\begin{aligned}
\text{match}(c\texttt{(}pat_1, \dots, pat_n\texttt{)}, c\texttt{(}v_1, \dots, v_n\texttt{)}) &= \textstyle\bigcup_{i=1}^{n} bind_i \\
&\quad \text{where} \\
&\quad\quad bind_i = \text{match}(v_i, pat_i), \\
&\quad\quad \text{consistent}(bind_1, \dots, bind_n) \\
\text{match}(x, v) &= \{x \to v\}
\end{aligned}
$$

Figure 3: Matching a Pattern against a Value

As a shorthand we define the predicate matches as

$$\text{matches}(pat, v) = (pat, v) \in \text{dom}(\text{match})$$

.

2.2.5 Reduction Relation

With the help of the previous definitions the reduction relation $\longrightarrow$ can now be formalized. The rules shown in Figure 4.

$$\text{CONG}\ \frac{F \vdash e_1 \longrightarrow e_2}{F \vdash E[e_1] \longrightarrow E[e_2]}$$

$$\text{FAPP}\ \frac{f\texttt{(}x_1\texttt{,} \dots \texttt{,}x_n\texttt{)}\ \texttt{=}\ e \in F}{F \vdash f\texttt{(}v_1\texttt{,} \dots \texttt{,}v_n\texttt{)} \longrightarrow e[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]}$$

$$\text{MATCH}\ \frac{\begin{array}{c}\neg\,\text{matches}(pat_j, v), j \in \{1, \dots, i-1\} \\ \text{match}(pat_i, v) = \{x_1 \mapsto v_1, \dots, x_k \mapsto v_k\}\end{array}}{\begin{array}{c}F \vdash \texttt{match}\ v\ \texttt{case}\ pat_1 \texttt{->}\ e_1 \dots \texttt{case}\ pat_n \texttt{->}\ e_n \\ \longrightarrow e_i[x_1 \mapsto v_1, \dots, x_k \mapsto v_k]\end{array}}$$

Figure 4: Reduction Relation

The rules are read as follows: below the line is the conclusion and above the line are the premises. E.g. to conclude that the statement under the line is valid, all of the premises must hold. The reduction relation $\longrightarrow$ is defined as the smallest relation containing all elements that can be derived by the rules shown in Figure 4.

The reduction relation identifies two classes of reducible expressions. The first class are function applications where all arguments are values. The rule FAPP says that an expression of the form $f(v_1, \ldots, v_n)$ reduces to the body e of the function definition for f where the formal arguments are replaced by the actual arguments. The second class of reducible expressions are pattern matches where the expression matched against is a value. The rule MATCH tells us that we need to find the first pattern i where pat_i matches v. In this case the whole expression reduces to the right hand side e_i where the pattern variables are replaced by their matches.

In all other cases the reductions take place inside subexpression. The congruence rule CONG covers these cases. Note that there can be at most one decomposition of an expression into an evaluation context and a subexpression such that the subexpression is reducible according to the rules FAPP or MATCH. The definition of the evaluation contexts (see Section 2.2.2) uniquely specifies the path we have to take to search for reducible subexpressions.

Before we give some examples illustrating the reduction semantics, we define two functions for later use:

```
plus(x,y) = match x
              case Zero     -> y
              case Succ(x') -> plus(x',Succ(y))

remove-plus(x) =
  match x
    case Zero      -> Zero
    case Succ(y)   -> Succ(remove-plus(y))
    case Plus(y,z) -> plus(remove-plus(y),remove-plus(z))
```

The function `plus` computes the addition of two numbers. The function `remove-plus` uses the function `plus` to evaluate an arithmetik expression with addition and successor function. Is does this by recursively replacing all `Plus` nodes in the tree by the sum of their arguments.

Example 2.6
The first example illustrates the rules FAPP and MATCH. Let's look at the expression

```
plus(Succ(Zero),Zero)
```

Since both `Succ(Zero)` and `Zero` are values the rule FAPP applies. This means we can reduce the expression to the body of the function `plus` replacing the formal argument `x` by the value `Succ(Zero)` and the argument `y` by the value `Zero`. This yields the expression

```
match Succ(Zero)
  case Zero     -> Zero
  case Succ(x') -> plus(x',Succ(Zero))
```

Now we are in a situation where the rule MATCH applies, because `Succ(Zero)` is already a value. The first pattern `Zero` does not match the value `Succ(Zero)`. But the second pattern `Succ(x')` succeeds yielding the substitution $\{$`x'` $\mapsto$ `Zero`$\}$. According to the reduction rule the whole expression reduces to

```
plus(Zero,Succ(Zero))
```

Another application of the rule FAPP yields the expression

```
match Zero
  case Zero     -> Succ(Zero)
  case Succ(x') -> plus(x',Succ(Succ(Zero)))
```

and we finally arrive at the value `Succ(Zero)` by the rule MATCH.

Example 2.7
We now show the application of the CONG rule. Therefor we start with the expression

```
Succ(plus(Zero,Succ(Zero)))
```

This expression is not directly reducible, because neither FAPP nor MATCH applies. But we can decompose it into the evaluation context $E =$ `Succ(●)` and the expression $e =$ `plus(Zero,Succ(Zero))` so that
$E[e] =$ `Succ(plus(Zero,Succ(Zero)))`.

The expression e reduces to $e' =$

```
match Zero
  case Zero     -> Succ(Zero)
  case Succ(x') -> plus(x',Succ(Succ(Zero)))
```

in one step by FAPP. Hence, by CONG the expression
`Succ(plus(Zero,Succ(Zero)))` $= E[e]$ reduces to $E[e'] =$

```
Succ(
  match Zero
    case Zero     -> Succ(Zero)
    case Succ(x') -> plus(x',Succ(Succ(Zero))))
```

It is clear from the definition of the reduction relation that a value cannot reduce further since it does not contain reducible expressions. Hence, once we reach a value after a sequence of reduction steps the evaluation terminates. But are values the only expressions that cannot further reduce? The answer is no. To see why, we give some examples.

- Reaching an expression `x` consisting only of a variable is a dead end in the evaluation sequence. Recall that both formal arguments of function definitions and pattern variables are replaced by values during evaluation. This implies that whenever we arrive at the expression `x` after a number of evaluation steps `x` was a free variable of the original expression.

- An function application of the form $f(v_1, \ldots, v_n)$ is stuck if f is not defined or takes a different number of arguments as supplied.

- A pattern matching expression cannot reduce further if none of its patterns matches.

We expect the evaluation of a program to yield a value as to be of any use. This suggests that the notion of a syntactically well formed program is not precise enough to only allow valid programs. In the reminder of the thesis we will explore how type systems help us to separate valid programs from programs that we consider to have no meaning without having to evaluate them.

3 Regular Tree Grammars

When designing a type system for a language we want to define types that are able so describe sets of that language's values (see Section 1). In our calculus values are trees. Hence, it is natural to model types as sets of trees.

Tree grammars [5] are a formalism to concisely describe sets of trees. In short, tree grammars are the counterpart to common word grammars [12] that describe languages of words. Like for word grammars there is also a hierarchy of classes of tree grammars, e.g. regular tree grammars, context free tree grammars, etc. The higher we get in the hierarchy the more powerful the grammars become, i.e. the broader is the set of languages they can describe. On the other hand, we loose closure properties and decision problems raise in complexity or get undecidable.

In our case it showed sufficient to stick to regular tree grammars. Regular tree grammars share many desirable properties with regular word grammars. For example, they are closed under union and intersection. And decision problems like inclusion are decidable.

In the reminder of this section we formally define tree grammars and their languages. Most of this content is not new although we sometimes modify definitions slightly to meet our needs. A thorough introduction to the material presented here can be found for example in [5]. Furthermore, we discuss how to exploit regular tree grammars as types for our language.

3.1 Regular Tree Grammars

Let c be a set of constructor names and n a set of nonterminal names. Then a tree over c and n is inductively defined as follows.

$$\begin{array}{lcl} \textit{tree} & ::= & n \mid c\texttt{(}\textit{tree}\texttt{,}\ldots\texttt{,}\textit{tree}\texttt{)} \\ c & ::= & \text{set of constructor names} \\ n & ::= & \text{set of variable names} \end{array}$$

We distinguish constructor names from variable names syntactically: constructor names start with a capital letter while variables names start with a lower case letter. Furthermore, we allow to omit the parentheses for constructors without arguments. Note, that the definition of *tree* coincides with the definition of values from Section 2.2.1 for $n = \emptyset$.

A regular tree grammar is defined as follows.

$$\begin{array}{lcl} \textit{grammar} & ::= & \texttt{G(}n\texttt{, }\pi\texttt{)} \\ \pi & ::= & \texttt{(}\textit{prod}\texttt{,}\ldots\texttt{,}\textit{prod}\texttt{)} \\ \textit{prod} & ::= & n \texttt{ -> } \textit{tree} \texttt{ | } \ldots \texttt{ | } \textit{tree} \end{array}$$

It consists of a start symbol and a list of productions π. A production maps a nonterminal n to a set of trees over c and n.

A grammar G describes a set of values—its *language* $L(G)$. For specifying the language generated by a grammar we need the notion of *contexts* - a modified version of the evaluation contexts from Section 2.2.2. A context E is defined as a tree containing a hole:

$$E \quad ::= \quad \bullet \mid c\texttt{(}\mathit{tree}, \dots, \mathit{tree}, E, \mathit{tree}, \dots, \mathit{tree}\texttt{)}$$

A tree tree_b can be derived from a tree tree_a by the grammar $G = \texttt{G(}s\texttt{,}\pi\texttt{)}$ if and only if

- There exists a context E and a nonterminal n, so that $E[n] = \mathit{tree}_a$.
- There exists a production n `->` tree_1 `|...|` tree_k in π and an i so that $E[\mathit{tree}_i] = \mathit{tree}_b$.

This means, nonterminals can be substituted by right hand sides of productions. The language $L(G)$ is then defined as the set of trees over c and $n = \emptyset$ that can be derived from the start symbol of the grammar in a finite number of steps. The requirement $n = \emptyset$ ensures that all trees in $L(G)$ do not contain any variables, hence are values.

Example 3.1
Let's look at the grammar

```
num := G(n, (n -> Zero | Succ(n)))
```

This grammar describes the set of natural numbers. To see why, lets look at the trees derivable from the start symbol `n`. The value `Zero` can be derived from the start symbol in one step. Alternatively we can also derive `Succ(n)` in one step. `Succ(n)` is not yet a value because it contains the nonterminal `n`. Further derivations hence lead to the values `Succ(Zero)`, `Succ(Succ(Zero))` and so forth. Hence, the language $L(\texttt{num})$ is the set $\{\texttt{Succ}^i\texttt{(Zero)} | i \geq 0\}$ which describes exactely the peano numbers.

It is important to note that there can be several grammars describing the same set of trees. This means there can exist $G_1 \neq G_2$ with $L(G_1) = L(G_2)$.

Example 3.2
Take the following productions:

```
odd  -> Succ(even)
even -> Zero | Succ(odd)
```

With these definitions we can construct a different grammar also describing the set of natural numbers.

```
num' := G(n, (n    -> even | odd,
              odd  -> Succ(even),
              even -> Zero | Succ(odd)))
```

Our definitions of trees and tree grammars are slightly different than the standard definitions in the literature. Usually, each constructor name c is associated with a fixed arity n. This restricts the use of c to nodes with n children. Our definition is more liberal at that point. Constructor names can be used with different numbers of arguments at different positions. While this looks like our formalism is more powerful, in fact it is not. The number of arguments n is clear from the context at each point where an application $c(t_1, \dots, t_n)$ is used. This means we can define a whole class of constructors (c, n) for each constructor c separating the instances of that constructor with different numbers of arguments. This association induces an isomorphism both between grammars and trees such that a tree is in the language of a grammar if and only if the same tree with renamed constructors is in the language of the grammar with renamed constructors. While our formalism for tree grammars is equivalent to the standard formalism, it simplifies many of the following definitions and proofs.

3.2 Operations on Grammars

Regular tree grammars are closed under union and intersection.[5] This means that, given grammars G_1 and G_2, we can build the regular tree grammars $G_1 \cup G_2$ and $G_1 \cap G_2$ such that $L(G_1 \cup G_2) = L(G_1) \cup L(G_2)$ and $L(G_1 \cap G_2) = L(G_1) \cap L(G_2)$.

Furthermore, equivalence and inclusion is decidable for regular tree grammars.[5] This means there exist algorithms to check whether $L(G_1) = L(G_2)$ and $L(G_1) \subseteq L(G_2)$ for given grammars G_1 and G_2.

Finally, we can decide if a grammar G generates the empty language, i.e. if $L(G) = \emptyset$. [5]

In Section 10 some of the algorithms used in the prototype implementation of the system are explained. For the theoretical analyses in the following sections it is enough to know that there exist algorithms to compute the above mentioned operations.

3.3 Normalization

Working with grammars becomes much simpler if they are *normalized.* We define a normalized grammar as follows:

- All right hand sides of a production have the form `c(`n_1`,...,`n_k`)` where c is a constructor name and n_1 to n_k are nonterminals. Especially, no productions of the form n_1 `->` n_2 are allowed.

- For all nonterminals mentioned within a grammar there exists exactely one production mapping that nonterminal to a (possibly empty) set of right hand sides.

For each grammar G it is possible to construct a normalized grammar G' so that $L(G) = L(G')$. An algorithm is shown in Section 10. Hence, from now on we assume all grammars to be normalized if not otherwise mentioned.

3.4 Grammars as Types

As mentioned at the beginning of this section, our goal is to model types as sets of values. Tree grammars are a means of describing sets of values by the language they generate. But we have seen that the grammar describing a certain set of values is not unique: there may exist different grammars generating the same language. From a semantic point of view these grammars are equivalent. Hence, we define types as set of values that can be described by a regular tree grammar. So grammars are only a textual *representation* of types.

However, we will use the terms grammar and type interchangeably in the following keeping in mind the abovementioned distinction. Since equivalence of grammars is decidable there is no formal problem with that less strict terminology.

3.5 Subtyping

We define a *subtype* relation on types as follows. t_1 is a subtype of t_2 if and only if every value in t_1 is also in t_2. We write $t_1 <: t_2$ for t_1 is a subtype of t_2. Hence subtyping corresponds to inclusion of languages. Since types are represented by regular tree grammars, a type t_1 represented by a grammar G_1 is a subtype of a type t_2 represented by a grammar G_2 if $L(G_1) \subseteq L(G_2)$.

As mentioned above, inclusion is decidable for regular tree grammars. It follows, that subtyping is also decidable.

4 Simply Typed Language

We have seen in Section 2.2.5 that there exist syntactically correct expressions that are neither a value nor can take a reduction step. These expressions are stuck—we cannot assign any meaning to them. The goal of designing a type system for the language is to separate these invalid terms from valid ones. Hence we want to define the subset of *well-typed* terms enjoying the following property: evaluation of a well-typed expression according to the operational semantics (see section 2.2) will never reach a dead end. By dead end we mean to reach an expression that is neither a value nor can take a further reduction step.

Note, that we do not require the stronger property that every well-typed expression reduces to a value in a finite number of steps. The reason is that we allow arbitrary recursion via function definitions.

The goal of the type system is to generate conclusions of the form expression e has type t. Since expressions always occur in a context of function definitions we need to lift the notion of typing to whole programs. A program $F\ e$ is well-typed if all function definitions in F are well-typed and there exists a type t so that, in the context of the function definitions F, e has type t.

In order to be able to check each function definition separately we need to extend the syntax of the core language with some type annotations. In function definitions we annotate a type for each formal argument and a result type. These annotations define an "interface" for a function. This implies two things: First, we can type the body of the function in isolation assuming the formal arguments have the annotated type. And second, when typing function applications we can use the type information from the annotations and don't need to look into the implementation of the function again.

There is a second—less obvious—place in the syntax that requires type annotations for typing. Let's consider an expression of the form

$$\texttt{match}\ e_m\ \texttt{case}\ pat_1\ \texttt{->}\ e_1 \ldots \texttt{case}\ pat_n\ \texttt{->}\ e_n.$$

The patterns pat_i can contain pattern variables. This implies that these variables may occur free in the expressions e_i. During evaluation the pattern variables will bind to actual values and all occurrences in e_i will be replaced by these values. Hence, each case expressions acts like a function definition where the pattern variables correspond to the formal arguments and the right hand side e_i corresponds to the function body. This means we need to know the types for the pattern variables to be able to type the right hand sides e_i. There are two possibilities for placing type annotations to solve the issue: Either we annotate each pattern variable or we annotate the expression e_m and compute the types for the pattern variables from that. We choose to annotate e_m for two reasons. First, this version requires only one annotation per pattern matching expression instead of on annotation per pattern variable. And second, this approach is common in type systems with record- or sum types [24] where the typing rules are related to our rules for pattern matching.

Figure 5 shows the extended syntax for the simply typed language. The differences to the core language are highlighted with boxes. In addition to the annotations for function definitions and pattern matches we require to annotate the top level expression of a program. Type checking a program hence means to check whether all functions adhere to their interfaces and to check whether the top level expression has the annotated type assuming the functions' interfaces.

$$\begin{array}{lcl}
\mathit{program} & ::= & F\ \mathit{expr}\ \boxed{:\ t} \\
F & ::= & \mathit{fdef} \ldots \mathit{fdef} \\
\mathit{fdef} & ::= & f\texttt{(}x\ \boxed{:\ t}\texttt{,} \ldots \texttt{,}x\ \boxed{:\ t}\texttt{)}\ \ \boxed{:\ t}\ \texttt{=}\ \mathit{expr} \\
\mathit{expr} & ::= & x \\
 & & \mid f\texttt{(}\mathit{expr}\texttt{,} \ldots \texttt{,}\mathit{expr}\texttt{)} \\
 & & \mid c\texttt{(}\mathit{expr}\texttt{,} \ldots \texttt{,}\mathit{expr}\texttt{)} \\
 & & \mid \texttt{\textbf{match}}\ \mathit{expr}\ \boxed{:\ t}\ \mathit{caseexpr} \ldots \mathit{caseexpr} \\
\mathit{caseexp} & ::= & \texttt{\textbf{case}}\ \mathit{pat}\ \texttt{->}\ \mathit{expr} \\
\mathit{pat} & ::= & x \mid c\texttt{(}\mathit{pat}\texttt{,} \ldots \texttt{,}\mathit{pat}\texttt{)} \\
t & ::= & \texttt{G(}n\texttt{, (}\mathit{prod}\texttt{,} \ldots \texttt{,}\mathit{prod}\texttt{))} \\
\mathit{prod} & ::= & n\ \texttt{->}\ \mathit{rhs}\ \texttt{|} \ldots \texttt{|}\mathit{rhs} \\
\mathit{rhs} & ::= & c\texttt{(}n\texttt{,} \ldots \texttt{,}n\texttt{)} \\
x & ::= & \text{variable names} \\
c & ::= & \text{constructor names} \\
f & ::= & \text{function names} \\
n & ::= & \text{nonterminal names}
\end{array}$$

Figure 5: Syntax of the Simply Typed Language

Strictly speaking, the operational semantics from Section 2.2 is only defined for untyped terms. Since types do not play any rule during evaluation we lift the definition of the reduction rules to typed terms as follows: all typing annotations are ignored to determine which rule applies. But still the annotations are propagated unchanged to the result.

For the formalization of the type system we need the notion of a *typing context*. A typing context Γ is a finite mapping from variables to types. We write them as $\Gamma = \{x_1{:}t_1, \ldots, x_n{:}t_n\}$ keeping in mind that Γ is a mapping (hence no variable may occur more than once). The empty context is abbreviated by $\emptyset$.

A typing context Γ can be extended by another typing context Γ', written as $\Gamma \cup \Gamma'$. Since the result must again be a valid typing context, we define that the second context overwrites the first one if both have mappings to different types for the same variable.

Typing of expressions then is defined as a 4-ary relation $\subseteq (F \times \Gamma \times expr \times t)$, where F is a set of function definitions, Γ a typing context, e an expression and t a type as defined in Section 3.1. (F, Γ, e, t) in the typing relation means that, under the assumption of F and Γ, e has type t. We write it as $F, \Gamma \vdash e : t$.

Using the typing relation for expressions we can formally define what it means for a program to be well-typed.

$$\text{T-PROG}\ \frac{F, \{x_{i_1} : t_{i_1}, \ldots, x_{i_{k_i}} : t_{i_{k_i}}\} \vdash e_i : t_{res_i} \qquad F, \emptyset \vdash e : t}{\begin{array}{c} F = \{f_1(x_{1_1}\ :\ t_{1_1}, \ldots, x_{1_{k_1}}\ :\ t_{1_{k_1}})\ :\ t_{res_1} = e_1, \ldots, \\ f_n(x_{1_n}\ :\ t_{1_n}, \ldots, x_{n_{k_n}}\ :\ t_{n_{k_n}})\ :\ t_{res_n} = e_n\} \\ e:\ t \end{array}}$$

This rule says that a program is well-typed if all function definitions adhere to their interfaces and if the top level expression can be given the annotated type assuming the function context F and an empty typing context. To prove that a function definition adheres to its interface we check that the body of the function definition has the annotated type as result type. The function body can contain free variables, namely the formal arguments of the function. Therefore we need the typing context to contain the assumptions that the formal arguments have the annotated type for typing the body.

The goal of this Section is to define a type system for the simply typed language and to prove it sound. Typing pattern matching expressions turned out to be quite challenging. Therefore we need some preliminary considerations as a basis for the specification of the type system. In the reminder of this Section we discuss how to compute types for pattern variables and how to check for exhaustiveness of a set of patterns with respect to a type. Furthermore, we define the type system as a set of inference rules and prove it sound.

4.1 Computing Types for Pattern Variables

A pattern matching expression has the form

$$\begin{array}{l} \texttt{match}\ e_m\ \texttt{:}\ t_m \\ \quad \texttt{case}\ pat_1 \texttt{->}\ e_1 \\ \quad\quad \cdots \\ \quad \texttt{case}\ pat_n \texttt{->}\ e_n \end{array}$$

Depending on the value of the expression e_m at runtime the whole expression reduces to one of the right hand sides e_1 to e_n where the pattern variables are replaced by the values resulting from the pattern match. Type checking takes place before runtime. This means that we do not know the exact value of e_m. Which again implies that we also do not know which of the patterns will match.

But if we can show that e_m has the annotated type t_m we know the set of possible values of e_m, namely all values in t_m. Hence, the type annotation acts as an interface between the expression e_m matched against and the case expressions.

Since we do not know which pattern will actually match at runtime we need to type-check all right hand sides e_1 to e_n. The challenge is that these expression can contain free variables, namely the pattern variables bound by the respective patterns. Hence, to typecheck the right hand sides e_1 to e_n we need to know the set of values for the pattern variables. This means we need a way to compute a type for each pattern variable that contains all these values.

The only information we have is the type annotation t_m. This annotation gives us the set of values of the expression e_m at runtime. The goal is to find an algorithm that takes a pattern *pat* and a type t as input and yields a type t_x for each pattern variable x containing all possible values of x. Put otherwise, the algorithm should yield an appropriate typing context Γ_{pat} with $\mathrm{dom}(\Gamma) = \mathrm{vars}(pat)$.

Example 4.1
Let's consider the expression

```
match x : G(n, (n -> Zero | Succ(n)))
  case Zero    -> Zero
  case Succ(y) -> y
```

For typechecking this expression we do not know the exact form of `x`. All we know is that `x` will evaluate to a value in `G(n, (n -> Zero | Succ(n)))` at runtime—at least if we assume the type annotation to be correct. The first pattern `Zero` does not contain pattern variables. Therefore we can check the corresponding right hand side without additional information.

On the other hand, in order to typecheck the right hand side of the second case expression, namely `y`, we need a type for the pattern variable `y`. Hence, we have to compute the set of values of the pattern variable `y` when matching the pattern `Succ(y)` against a value of type `G(n, (n -> Zero | Succ(n)))`.

All values that match the pattern `Succ(y)` have to start with an application of the constructor `Succ`. On the other hand, for deriving a value from the grammar `G(n, (n -> Zero | Succ))` that starts with an application of the constructor `Succ` we have to use the production `n -> Succ(n)` as first derivation step. Hence, the pattern variable `y` could match to any value generated by the grammar starting with `n`. Since `n` is again the start symbol of the original grammar, the range of values of `y` is described by `G(n, (n -> Zero | Succ(n)))`.

There are two fundamental design choices for typing pattern matching expressions that we discuss in the following.

Design Choice: Non-Matching Patterns

As an example we consider the program fragment

```
match x : num
  case Foo(x,y) -> ...
  case ...
```

If the program adheres to the type annotation `num`, the pattern `Foo(x,y)` will never match at runtime, because there are no values in `num` of this form. Therefore the program behaves correctly at runtime if and only if it behaves correctly without the case expression `case Foo(x,y) -> ...`. Put otherwise, this case expression has no influence on the runtime behavior of the program. The right hand side of the case expression with the pattern `Foo(x,y)` is dead code.

There are two points to discuss. First, we have to decide if we want to allow to write patterns that are guaranteed to never match at runtime. And second, we need to decide if we require dead code to be well-typed.

- In the untyped version of our calculus constructor names can be used essentially everywhere. There are no runtime errors that are caused by using a constructor in a wrong way, because constructor names are not tied to type definitions as for example in Haskell's algebraic data types. Hence, from the point of view of the untyped calculus is makes perfectly sense to allow arbitrary patterns, since the input values are not restricted by type annotations. Although we have no direct benefit of allowing non-matching patterns in the simply typed language it does no harm either. However, we will later see in the context of parametric types (see Section 9) that it is useful to allow arbitrary patterns in a pattern matching expression. As an example consider the following code with a parametric type annotation.

  ```
  match x : X
    case Zero -> Succ(Zero)
    case y    -> y
  ```

  Independently of the concrete instantiation of the type variable `X` this piece of code transforms the value `Zero` into `Succ(Zero)` and leaves all other values as is.

  And there is another—more technical—reason for allowing non-matching patterns. If we allow only matching patterns we need a means to determine if the annotated type contains at least one value that matches the pattern. This implies that we need to now if certain types are empty or not. For example the grammar

  ```
  G(s, (s -> Succ(s))
  ```

  generates the empty language. In order to reject the pattern `Succ(x)` we need to know that `s` is empty. For the simply typed language this is no problem. In fact we use an algorithm that determines if there are matching values in the grammar when computing the corresponding typing contexts. But matters are different in the context of type variables inside grammars. In order to determine if the grammar is empty we need to know if the type variable abstracts over an empty or a non-empty type which makes the language and the type system much more complicated.

  For these reasons we choose to allow arbitrary patterns in pattern matching expressions.

- The second question concerns the type checking of dead code. One could argue that dead code is guaranteed to never execute and hence does not need to we well-typed. From the point of view of soundness this is correct. Dead code has no influence on the runtime behavior of a program. On the other hand, if we do not require dead code to be well-typed we classify some “strange” programs as well-typed. Take for example the following program.

  ```
  match Zero : num
    case Foo     -> y
    case Zero    -> ...
    case Succ(x) -> ...
  ```

  The pattern Foo does not match any values of type `num`. Hence, the right hand side of the first case expression, namely `y`, is dead code. The right hand side `y` is clearly ill-typed, because it consists of a free variable. Nevertheless, the program as a whole would be classified as well-typed in the case that we don’t require the type checking of dead code.

  In our opinion it is confusing to allow such strange programs. Therefore, we require also dead code to be well-typed.

  But in order to typecheck the right hand side of a case expression we need types for the pattern variables of the corresponding pattern, since they occur free inside of the respective right hand side. We specified that a typing context for checking the right hand side of a case expression should map a pattern variable to a type containing all values that can bind to that variable at runtime. A non-matching pattern is guaranteed to never match any value at runtime. This implies that there are also no values that can bind to its pattern variables at runtime. Therefore, it is straight forward to map a pattern variable to the empty type in the resulting typing context. Note, that using the empty type for a variable is the least restriction we can put on the type checking of expressions. Due to subtyping, an expression with the empty type can be used in any context, because the empty type is a subtype of every other type. The only remaining requirement is that the expression does not contain free variables.

Design Choice: Precision

The second design choice concerns the precision of the typing contexts for typechecking the right hand sides of case expression. We illustrate the issue by a small example.

Example 4.2
We consider the pattern

```
A(x,y)
```

and the grammar

```
g ::= G(s, (s -> A(b,c) | A(d,e),
            b -> B, c -> C,
            d -> D, e -> E))
```

This grammar generates the finite set of trees {`A(B,C)`, `A(D,E)`}. Hence, the set of possible values of the pattern variable `x` is {`B`, `D`} and the respective set for the pattern variable `y` is {`C`, `E`}. This means, we can use the typing context $\{\mathtt{x} : \mathtt{b} \cup \mathtt{d}, \mathtt{y} : \mathtt{c} \cup \mathtt{e}\}$ for type checking the right hand side `e` of the case expression in the following program.

```
match x : g
  case A(x,y) -> e
```

Since the typing context maps each pattern variable to a type containing all possible values of that variable, any evaluation of the expression `e` will behave correctly if `e` is well-typed under this typing context. Hence, computing the typing context in the abovementioned way is sound.

However, consider the expression

```
match x : g
  case A(x,y) -> A(x,y)
```

This piece of code looks like the identity function on expressions of type `g`. The problem is, that we cannot conclude that the right hand side of the case expression `A(x,y)` has the type `g` if we use the typing context $\{\mathtt{x} : \mathtt{b} \cup \mathtt{d}, \mathtt{y} : \mathtt{c} \cup \mathtt{e}\}$. The best type that we can give to that expression is the type

```
G(s, (s -> A(bd,ce),
      bd -> B | D,
      cd -> C | E))
```

Hence, we loose type information by the process of deconstructing and reconstructing expressions. This is due to the fact that we unified the results of the two different productions for the start symbol of the grammar `g` in the course of computing the typing context. However, each of the productions `s -> A(b,c)` and `s -> A(d,e)` establishes a relationship between the first and second argument of the constructor `A` respectively. The first production only allows the combination `B,C` while the second production restricts the range of possible arguments to `D,E`. Hence, the combinations `A(B,E)` and `A(C,D)` are not possible.

In order to propagate this information about possible combinations of arguments to the type checking process of the right hand side of case expressions we have to use a set of typing contexts instead of one aggregated result. In the last example this means to use the two separate typing contexts, namely $\Gamma_1 = \{\texttt{x} : \texttt{b}, \texttt{y} : \texttt{c}\}$ and $\Gamma_2 = \{\texttt{x} : \texttt{d}, \texttt{y} : \texttt{e}\}$, instead of aggregating the data into one single typing context. Both typing contexts still contain all possible values of the pattern variables. Therefore, it is sound to check the well-typedness of the whole expression by checking if the right hand side is well-typed under each of the typing contexts Γ_1 and Γ_2 separately. Hence, this more precise approach leads to a more complete type system: more "valid" programs can be typed.

The disadvantage of this method is that type checking becomes much more expensive in terms of complexity. Instead of one check per case expression in a pattern match there may now be many of them. And this effect propagates in an exponential way through the process off type checking. Subexpressions of pattern matches can again contain pattern matches that require more than one subderivation per case expression and so forth.

Since the scope of this thesis is to investigate the formal properties of the calculus, complexity is not a major concern. Therefore we choose to use the following compromise: we specify the algorithm for the computation of the typing contexts in a way that propagates the most precise information. In the formulation of the type system we aggregate the result of the algorithm into one single typing context as to make the typing rules and the corresponding soundness proof easier to understand. But thereby we keep in mind that it is possible to arrive at a more complete type system by exploiting the extra information we get from our version of the algorithm.

The Algorithm

The algorithm for computing the set of typing contexts for the pattern *pat* with respect to type $t = \texttt{G(}s\texttt{,}\pi\texttt{)}$ has the following general form.

- If the pattern consist only of a pattern variable, i.e. $pat = x$, we return the singleton set containing the typing context $\Gamma = \{x : t\}$.
- If the pattern *pat* has the form $C\texttt{(}p_1\texttt{,} \ldots \texttt{,}p_n\texttt{)}$ we proceed as follows:
    1. We filter all productions of the form s `->` $C\texttt{(}n_1\texttt{,} \ldots \texttt{,}n_n\texttt{)}$ from π. These are the only productions that can be used when deriving values from the grammar that match the pattern $C\texttt{(}p_1\texttt{,} \ldots \texttt{,}p_n\texttt{)}$.
    2. For each right hand side $C\texttt{(}n_1\texttt{,} \ldots \texttt{,}n_n\texttt{)}$ we recursively compute sets of typing contexts G_1 to G_n for the subpattern p_i and the grammar $\texttt{G(}n_i\texttt{,}\pi\texttt{)}$ respectively.
    3. For each right hand side we aggregate the resulting sets of typing contexts G_1 to G_n.
    4. We return the result of the aggregation.

It remains to specify how to aggregate the sets of typing contexts belonging to recursive calls for the same right hand side of a production, namely step 3 in the recursive case of the above algorithm.

To simplify the explanation we first consider the case where each subcall returns a single typing context.

Example 4.3
We consider the pattern

```
A(x,y)
```

and the grammar

```
G(s, (s -> A(b,c),
       b -> ..., c -> ...))
```

The two recursive calls for the production `s -> A(b,c)` return the typing contexts $\Gamma_1 = \{\mathtt{x} : \mathtt{b}\}$ and $\Gamma_2 = \{\mathtt{y} : \mathtt{c}\}$ respectively. Each value that matches the complete pattern `A(x,y)` has one subvalue matching the subpattern `x` and one subvalue matching the subpattern `y`. Hence, we need to unify the information of the typing contexts Γ_1 and Γ_2 into the resulting typing context $\Gamma = \Gamma_1 \cup \Gamma_2 = \{\mathtt{x} : \mathtt{b}, \mathtt{y} : \mathtt{c}\}$.

In the last example, each pattern variable occurs only once inside the pattern. Hence, the domains of the typing contexts Γ_1 and Γ_2 are disjoint and the union $\Gamma_1 \cup \Gamma_2$ yields a valid typing context. The resulting typing context is correct in the sense that it maps each pattern variable to a type containing all possible values of that pattern variable at runtime. But in some cases it may be too imprecise. Consider the case, where the grammar allows to derive values from the nonterminal `b` but not from the nonterminal `c`. This means that there cannot be derived any values from the grammar. It follows, that the typing context $\Gamma' = \{\mathtt{x} : \emptyset, \mathtt{y} : \emptyset\}$ is more appropriate. Especially, in this case the grammar is equivalent to the grammar `G(`s`, (`s` -> ))` where we would return this more precise typing context right away. Since annotating an expression with different but equivalent grammars should not lead to different result during type checking, we have to return the more precise typing context in both cases.

A pattern variable that occurs more than once inside a pattern puts additional constraints on the process of pattern matching. These additional constraints need to be considered when computing the corresponding typing context.

Example 4.4
We take the pattern

```
A(x,x)
```

and the grammar

```
G(a, (a -> A(b,c),
       b -> B | D, c -> B | D,
       d -> D))
```

The subcalls for the right hand side `A(b,c)` return the typing contexts $\Gamma_1 = \{\texttt{x} : \texttt{b}\}$ and $\Gamma_2 = \{\texttt{x} : \texttt{c}\}$. We can see that the domains of Γ_1 and Γ_2 are no longer disjoint.

If we recall that the pattern only matches a values when all occurrences of x bind to exactly the same value, it is clear that we have to take the intersection of the types `b` and `c` when unifying Γ_1 and Γ_2. Hence, the result should be $\Gamma = \{\texttt{x} : \texttt{b} \cap \texttt{c} = \texttt{d}\}$ resulting from a union with pointwise intersection of the typing contexts Γ_1 and Γ_2. We write it as $\Gamma_1 \cup_\cap \Gamma_2$.

In the case that the recursive calls return more than one typing context respectively we have to lift the operation "unify" to lists of typing contexts.

Example 4.5
We consider the pattern

```
A(B(x1,x2),
  C(x3,x4))
```

and the grammar

```
G(s, (s -> A(b,c),
      b -> B(b1,b2) | B(b3,b4),
      c -> C(c1,c2) | C(c3,c4)))
```

We omit the productions for the nonterminals `b1` to `c4`, since they are not relevant for the following discussion.

For deriving a value from the grammar that matches the pattern we have to always use the production `s -> A(b,c)` as a first step. But for both nonterminals `b` and `c` we can choose between two different productions for the next derivation step. Since this choice is independent for both nonterminals `b` and `c`. It follows that we get $2 * 2$ possibilities to start a derivation of a value that matches the complete pattern.

Computing the set of typing contexts for the subpattern `B(x1,x2)` and the grammar `y` yields the set $G_1 = \{\Gamma_{1_1}, \Gamma_{1_2}\}$ with $\Gamma_{1_1} = \{\texttt{x1}:\texttt{b1}, \texttt{x2}:\texttt{b2}\}$ and $\Gamma_{1_2} = \{\texttt{x1} : \texttt{b3}, \texttt{x2} : \texttt{b4}\}$. The two typing contexts Γ_{1_1} and Γ_{1_2} propagate the constraints about possible combinations of arguments to the constructor `B`.

In the same way, we get the set of typing contexts $G_2 = \{\Gamma_{2_1}, \Gamma_{2_2}\}$ with $\Gamma_{2_1} = \{\texttt{x3} : \texttt{c1}, \texttt{x4} : \texttt{c2}\}$ and $\Gamma_{2_2} = \{\texttt{x3} : \texttt{c3}, \texttt{x4} : \texttt{c4}\}$ for the subpattern `C(x3,x4)`.

As discussed above, the derivations for the arguments `b` and `c` of the constructor `A` are independent from each other. Put otherwise, the range of values of the arguments to the constructor `A` consists of the complete product space of values that can be derived from the nonterminals `b` and `c`. This means we have to compute the cross product of the sets of typing contexts G_1 and G_2 to get the final result. This yields the set of typing contexts $G = \{\Gamma_{1_1} \cup_\cap \Gamma_{2_1}, \Gamma_{1_1} \cup_\cap \Gamma_{2_2}, \Gamma_{1_2} \cup_\cap \Gamma_{2_1}, \Gamma_{1_2} \cup_\cap \Gamma_{2_2}\}$. We write the operation as $G_1 \times_{\text{uni}} G_2$.

After the preceding discussion it is now straight forward to formalize the algorithm that computes the sets of typing contexts for a pattern *pat* and a type t. The definition is shown in figure 6. The main function "contexts" uses the helper function "contexts′". The helper function has the following semantics: for each combination of values of the pattern variables there is a typing context in the result of the helper function containing this combination. Furthermore, the helper function returns the empty set if there are no values that can match the pattern.

In the case that the helper function returns the empty set, the main function yields the singleton set containing a typing context $\Gamma_\emptyset$. $\Gamma_\emptyset$ maps all pattern variables to the empty set. Otherwise the result of the helper function is used unchanged. This approach ensures that the algorithm yields the same result for equivalent grammars.

The helper function "contexts′" works as follows. If the pattern consists of only a pattern variable x, the singleton set containing the typing context mapping the variable x to the type t is returned in case that the type t is not empty. Otherwise we return the empty set. Returning the empty set propagates through the cross product operation in case of recursive calls. Building the cross product of a set of sets where at least one of the sets is empty yields the empty set. Which is exactely what we want: if one of the subpattern does not match a value, the whole pattern does not match either.

In the recursive case with a pattern of the form $C(p_1, \ldots, p_n)$ we first filter all matching right hand sides of the form $C(n_{1_j}, \ldots, n_{n_j})$ from the grammar's productions. Then we recursively compute the sets of typing contexts for the subpatterns p_1 to p_n and the corresponding subgrammars. The results from recursive calls belonging to the same right hand side of a production are aggregated using the cross product. Since we use the intersection operator in the cross product when unifying typing contexts containing the same pattern variable, the result may contain a typing context with an empty type in the codomain. This means that there is no value that can bind to the corresponding pattern variable at runtime in the respective combination. Hence we have to drop that typing context. This is done by the operator "filter-nonempty". Finally the union of the resulting sets of typing contexts from all right hand sides are returned as final result.

There remains one corner case that we need to consider, namely the meaning of the cross product of zero factors. This happens every time we encounter a constructor application without arguments. In this case, the pattern does not contain any pattern variables, hence we should return the singleton set consisting of the empty typing context. Note, that it would be wrong to return just the empty set. This empty set would propagate through the cross product operation up to the final result which is not what we want. Therefore, we define the cross product of zero factors as $\{\emptyset\}$.

For the formulation of the type system we still need to define how to aggregate a set of typing contexts into one single typing context. This is simply done by pointwise computing the union of the resulting set of typing context.

$$\text{contexts}(p, t) = \begin{cases} \{\Gamma_\emptyset\} & \text{if } G = \emptyset \\ G & \text{otherwise} \end{cases}$$
where
$G = \text{contexts}'(p, t)$
$\Gamma_\emptyset = \{x_i \mapsto \emptyset\}, x_i \in \text{vars}(p)$

$$\text{contexts}'(x, t) = \begin{cases} \emptyset & \text{if empty}(t) \\ \{\{x : t\}\} & \text{otherwise} \end{cases}$$

$$\text{contexts}'(C\texttt{(}p_1\texttt{,} \ldots \texttt{,}p_n\texttt{)}, G\texttt{(}s\texttt{,}\pi\texttt{)}) = \bigcup G'_j$$
where
$$\begin{aligned} C\texttt{(}n_{1_j}\texttt{,} \ldots \texttt{,}n_{n_j}\texttt{)} &\in \pi(s) \\ G_{i_j} &= \text{contexts}'(p_i, \texttt{G(}n_{i_j}\texttt{,}\pi\texttt{))} \\ G_j &= G_{1_j} \times_{\text{uni}} \cdots \times_{\text{uni}} G_{n_j} \\ G'_j &= \text{filter-nonempty}(G_j) \end{aligned}$$

Figure 6: Computation of Typing Contexts for Pattern Variables

We write the aggregation by pointwise union as $\bigcup_\cup$. The function is shown in figure 7

$$\text{context}(pat, t) = \bigcup\nolimits_\cup \text{contexts}(pat, t)$$

Figure 7: Aggregation of Typing Contexts

4.2 Exhaustiveness of Patterns

With the algorithm from the last section we are able to typecheck the right hand sides of case expressions in a pattern match. But we still do not know if for any possible value of the expression matched at least one pattern matches.

If we look at the reduction relation (see figure 4) we see that a pattern matching expression of the form

$$\texttt{match } e \texttt{ : } t \texttt{ case } pat_1 \texttt{->} \ e_1 \ldots \texttt{case } pat_n \texttt{->} \ e_n$$

is stuck (cannot reduce further) if e is a value and none of the patterns matches it. Since the goal of the type system is to eliminate stuck expressions we have to make sure that such an expression is not well typed.

Typing an expression is a static operation which means that we cannot know the exact shape of e, because typing happens before actually evaluating the expression. But the type annotation t tells us the range of possible values of e. To ensure that the expression will never get stuck during evaluation there has to be at least one patching pattern for each value in t. If we can rely on the fact that e will actually evaluate to a value in t (this will be shown in section 4.4 in the course of proving soundness of the type system) and show the abovementioned property, the expression cannot get stuck at runtime.

The fact, that for each value of a type t there is one matching pattern in a set of patterns $P = \{pat_1, \ldots, pat_n\}$ is called *exhaustiveness* of the patterns P with respect to the type t.

Assume that we can construct a type $t_i <: t$ for each pattern pat_i with the property that for all values $v \in t_i$ the pattern pat_i matches v. If t is a subtype of the union $\bigcup t_i$ we can be sure that there cannot be any value in t without a corresponding matching pattern.

Example 4.6
Let's look at the expression

```
match x : num
  case Zero    -> ...
  case Succ(y) -> ...
```

The only value in `num` matching the pattern `Zero` is `Zero`. Hence, the biggest type `t1` with the property `t1` $<:$ `num` and forall $v \in$ `t1` the pattern `Zero` matches v is the grammar

```
t1 = G(z, (z -> Zero))
```

The pattern `Succ(y)` will match all values in `num` that start with the application of the constructor `Succ`. So the biggest appropriate type for this pattern is the grammar

```
t2 = G(s, (s -> Succ(n),
           n -> (Zero) | Succ(n)))
```

The union `t1` $\cup$ `t2` contains all values of `num`, hence `num` is a subtype of `t1` $\cup$ `t2` and therefore the patterns are exhaustive.

In the last example there a two things to note:

- Although we are formally only checking that $t <: \bigcup t_i$, we are in fact showing that $\bigcup t_i \cong t$. This is due to the requirement that all t_i are subtypes of t. Which means that there are no values in t_i that are not also in t (which carries over to the union). This property gives us the relation $\bigcup t_i <: t$. Hence $\bigcup t_i$ is a (possibly not disjoint) decomposition of t. We will later relax the requirement $t_i <: t$ as to simplify the exhaustiveness checking.

- The types `t1` and `t2` contain *exactly* the values of `num` that match the appropriate pattern. This is a stronger property than the requirement that `t1` and `t2` only contain values of `num` that match the respective pattern (which would allow `t1` and `t2` to be smaller). In the following we will see that there are cases where we have to exploit this possibility of making types smaller than the set of values the pattern matches.

A pattern without variables matches exactly one value, namely the value that is syntactically the same. For such a pattern is it straight forward to construct a type containing only the one value it matches. For patterns containing variables the construction of a type containing exactly the values it matches (with respect to a type) is slightly more involved and in some cases even impossible. Some examples will help to understand the construction (and its limitations).

Example 4.7
Let's start with the pattern

```
A(x,y)
```

and the type

```
G1 = G(a, (a -> A(b,c) | A(d,e),
           b -> B, c -> C,
           d -> D, e -> E))
```

Matching the pattern against the type yields the typing context $\Gamma =$ {`x : bd , y : ce`}, where `bd` is defined as `G(bd, (bd -> B | D))` and `ce` as `G(ce, (ce -> C | E))`.

If we construct a grammar from the pattern using this typing context we arrive at

```
G2 = G(a, (a -> A(x,y),
           x -> B | D,
           y -> C | E))
```

All values of the grammar `G2` match the pattern `A(x,y)`. But the grammar `G2` contains values that are not in `G1` (for example the value `A(b,e)`). With the help of the collecting algorithm contexts it would be possible to construct a grammar `G3` containing only the values from `G1` that match the pattern. Computing `G2` $\cap$ `G1` would yield the same result. In this case, `G3` is the same as `G1`, since all values of `G1` match the pattern `A(x,y)`.

But computing the more exact grammar `G3` is unnecessary here. For checking exhaustiveness it does not matter if the type t_i we choose to represent the pattern pat_i contains values that are not in the type t matched against. The subtype relation $t <: \bigcup t_i$ will still hold iff the patterns are exhaustive with respect to the type t.

Example 4.8
In the last example every pattern variable occurred only once. It was possible to construct type out of a pattern and a typing context that is

sufficiently exact to check exhaustiveness. Matters are different if a pattern variable occurs more than once. As an example we will look at the pattern `Plus(x,x)` matching against the type `arith-expr` (see section 2.1 for the definition).

The matching will yield the typing context $\Gamma = \{$`x : arith-expr`$\}$. This is correct: any value of `arith-expr` may bind to `x` at runtime. But if we try to construct a grammar in a similar way as we did in the last example we arrive at

```
G = G(s, (s -> Plus(ae,ae),
          ae -> ...))
```

where `ae` is defined similar as in `arith-expr`. This grammar contains many values that do not match the pattern `Plus(x,x)`, since the pattern can only match if both subvalues of the constructor `Plus` are identical. To see why using this grammar for checking exhaustiveness leads to wrong results, consider the case where the pattern `Plus(x,x)` is used together with the patterns `Succ(y)` and `Zero` in a pattern matching expression: the exhaustiveness test will succeed which is clearly wrong.

Moreover, it is not possible to construct a grammar that contains all values of `arith-expr` that match the pattern `A(x,x)`: to ensure that the subvalues of the constructor `Plus` are always identical we would need one grammar for each possible value of `arith-expr`. But since there are infinitely many such values (and grammars are finite objects), we cannot build such a grammar. In general, it is impossible to construct a grammar for a pattern where a variable occurring more than once maps to an infinite type in the respective typing context.

Since an exact grammar is not constructable we could try to use a finite approximation of the type. Note, that this will not make the type system unsound, since we err "on the right side": if the exhaustiveness check succeeds the answer is correct (but there are expressions that would never get stuck but that we will reject because of an incorrectly failing exhaustiveness check).

For example, if we approximate the grammar by the empty type the exhaustiveness check for the expression

```
match e : arith-expr
  case Plus(x,x)            -> ...
  case Plus(Zero,Plus(x,y)) -> ...
  case Plus(Zero,Succ(x))   -> ...
  case Plus(Plus(x,y),Zero) -> ...
  case Plus(Succ(x),Zero)   -> ...
  case Succ(x)              -> ...
  case Zero                 -> ...
```

would incorrectly fail. If we make the approximation a little bit better by using the grammar

```
G(s, (s -> Plus(z,z),
      z -> Zero))
```

the exhaustiveness check would correctly succeed. The better the approximation is, the more exhaustiveness checks will correctly succeed. But for each approximation we can find an expression where the exhaustiveness incorrectly fails. So, how good should the approximations be? The complexity of the expression above (which is the smallest example for a expression that incorrectly fails the exhaustiveness check for the worst approximation, namely the empty type) hints at a reasonable answer: in most cases it is sufficient to use the empty type as a representation for a pattern containing variables with more than one occurrence. The complexity of expressions that incorrectly fail the exhaustiveness check arises from the fact that a pattern cannot require that two variables have different values. This means, that in most expressions where a pattern `C(x,x)` is used, there will also be a pattern of the form `C(x,y)` somewhere below it in the list of case-expressions. Since the pattern `C(x,x)` is tried before the pattern `C(x,y)` the second pattern will actually only match with different bindings for `x` and `y`.

Figure 8 shows the formal definition of the exhaustiveness check. The function pat2gram constructs a grammar out of a pattern and a given typing context analog to examples we have seen. Patterns that contain multiple occurrences of one variable are represented by the empty type (first clause of the definition). The exhaustiveness check computes one representative grammar t_i for each pattern pat_i and checks the inclusion $t <: \bigcup t_i$.

$$\begin{array}{lcl}
\text{exhaustive}(\{pat_1,\ldots,pat_n\},t) & = & t <: \bigcup t_i \\
 & & \text{where} \\
 & & \quad \Gamma_i \;=\; \text{context}(pat_i, t) \\
 & & \quad t_i \;=\; \text{pat2gram}(pat_i, \Gamma_i) \\
 & & \\
\text{pat2gram}(pat, \Gamma) & = & \emptyset,\ \text{if} \\
 & & \quad \exists x : \text{count}(x, pat) > 1 \\
 & & \\
 & & \texttt{G(s, (}\quad \texttt{s -> }c\texttt{(n1, ... ,nn),} \\
\text{pat2gram}(c\texttt{(}pat_1\texttt{, ... ,}pat_n\texttt{)}, \Gamma) & = & \qquad\qquad \texttt{n1 -> }t_1\texttt{,} \\
 & & \qquad\qquad \texttt{...} \\
 & & \qquad\qquad \texttt{nn -> }t_n\texttt{))} \\
 & & \text{where} \\
 & & \quad t_i = \text{pat2gram}(pat_i, \Gamma), \\
 & & \quad \texttt{s}, \texttt{n1}, \ldots, \texttt{nn}\ \text{fresh} \\
 & & \\
\text{pat2gram}(x, \Gamma) & = & \texttt{G(s, (s -> }\Gamma(x)\texttt{))}
\end{array}$$

Figure 8: Exhaustiveness Check

4.3 Typing Relation

On the basis of the definitions from the last Sections we are now able to define the typing relation for expressions. The typing rules are shown in figure 9. As in the rules of the reduction relation, the conclusion is shown below and the premises above the horizontal line. A conclusion of the form $F, \Gamma \vdash e : t$ means that, assuming the function definitions in F and the assignment of variables to types in Γ, the expression e has type t. The typing relation is defined as the smallest relation containing all conclusions derivable by the typing rules.

$$\text{T-SUB}\ \frac{F, \Gamma \vdash e : t_{sub} \qquad t_{sub} <: t}{F, \Gamma \vdash e : t} \qquad\qquad \text{T-VAR}\ \frac{(x : t) \in \Gamma}{F, \Gamma \vdash x : t}$$

$$\text{T-FAPP}\ \frac{f\texttt{(}x_1\ \texttt{:}\ t_1\texttt{,}\ \ldots\ \texttt{,}x_n\ \texttt{:}\ t_n\texttt{)}\ \texttt{:}\ t\ \texttt{=}\ e \in F \qquad F, \Gamma \vdash e_i : t_i}{F, \Gamma \vdash f\texttt{(}e_1\texttt{,}\ \ldots\ \texttt{,}e_n\texttt{)} : t}$$

$$\text{T-CAPP}\ \frac{\pi(s_t) = C\texttt{(}n_1\texttt{,}\ \ldots\ \texttt{,}n_n\texttt{)} \qquad F, \Gamma \vdash e_i : \texttt{G(}n_i\texttt{,}\pi\texttt{)}}{F, \Gamma \vdash C\texttt{(}e_1\texttt{,}\ \ldots\ \texttt{,}e_n\texttt{)}\ \texttt{:}\ \texttt{G(}s_t\texttt{,}\pi\texttt{)}}$$

$$\text{T-MATCH}\ \frac{\begin{array}{c} F, \Gamma \vdash e : t_e \qquad \text{context}(pat_i, t_e) = \Gamma_i \\ \text{exhaustive}(\{pat_1, \ldots, pat_n\}, t_e) \qquad F, \Gamma_i \cup \Gamma \vdash e_i : t \end{array}}{F, \Gamma \vdash \texttt{match}\ e\ \texttt{:}\ t_e\ \texttt{case}\ pat_1\texttt{->}\ e_1 \ldots \texttt{case}\ pat_n\texttt{->}\ e_n : t}$$

Figure 9: Typing Relation for the Simply Typed Language

The rule T-SUB is called the subsumption rule. It says that whenever an expression is well-typed with type t_{sub} it is also well-typed with type t if t is a supertype of t_{sub}. Recall that subtyping is defined as subset relation on languages. This means all terms of t_{sub} are also terms of t. It follows that it is valid to use an expression of type t_{sub} in the context where an expression of type t is required. The subsumption rule is essential to make the system usable in practice. Without the subsumption rule arguments of function applications for example would have to be well-typed exactely with the type the function expects.

A variable x can only be a valid expression if we are inside a function definition with a formal argument called x. In all other cases the occurrence of x is free. Free variables will not be replaced by values during evaluation. This means we will arrive at the expression x after a number of reduction steps. A variable x is neither a value nor can reduce further, hence the expression should be rejected by the type system. If we are inside the body of a function definition all formal arguments are mentioned in the typing context Γ with their respective types. Therefore the rule T-VAR says that an expression of the form x is well-typed with type t if the typing context Γ contains the assumption $x : t$.

The rule T-FAPP deals with expressions that are function applications. As already discussed we use only the interfaces of the function definitions defined by their type annotations for the typing of a function application. A function application of the form $f(e_1, \ldots, e_n)$ is well-typed with type t under the following conditions.

- The function context F contains an appropriate function definition in terms of number of arguments.
- The result type of the function equals t.
- The actual arguments e_1 to e_n are well-typed with the types t_1 to t_n of the corresponding formal arguments.

Constructor applications are covered by the rule T-CAPP. The rule says that a constructor application has type t under the following conditions:

- the start symbol of the grammar $G(s_t,\pi)$ exactely one right hand side, namely a constructor with the right number of arguments.
- the actual arguments of the constructor call have the type of the above grammar with the respective nonterminal from the right hand side as start symbol.

It may seem too restrictive to constrain the productions for the start symbol of the grammar $G(s_t,\pi)$ to this one right hand side. It is clear that any type for the expression $C(e_1, \ldots, e_n)$ needs to have *at least* one right hand side of the start symbol's productions of the abovementioned form. Otherwise the grammar could not produce values with the constructor C at the top (remember that grammars are assumed to be normalized). But couldn't there possibly be a better type for the expression with more than one right hand side of that form? This is not the case. The key to understand why is that the subexpressions of constructor applications are independent from each other. If the subexpression e_{i_1} is well-typed with type t_{i_1} it can evaluate to *any* value in t_{i_1} at runtime. Independently from that the expression e_{i_2} can evaluate to *any* value in t_{i_2}. Hence, the best type we can give to the constructor application $C(e_1, \ldots, e_n)$ is a grammar that contains the whole product space of t_1 to t_n as possible subexpressions for the topmost constructor. This means that the one right hand side is in fact enough.

The rule T-MATCH says that a pattern match has type t under the following conditions:

- the expression e matched against has the annotated type e_t
- the patterns are exhaustive with respect to the annotated type e_t (see section 4.2)
- each right hand side of the case expressions e_i has type t assuming the typing context Γ extended by the typing context Γ_i (which assigns types to the pattern variables of the pattern pat_i).

We now illustrate the type system and the process of typing expression by a set of examples.

Example 4.9
Let's start very simple with the expression `Zero`. The following derivation tree witnesses that `Zero` has type `num`.

$$\text{T-SUB}\ \dfrac{\text{T-CAPP}\ \dfrac{\pi(\texttt{z}) = \texttt{Zero}}{F, \emptyset \vdash \texttt{Zero} : \texttt{zero}} \qquad \texttt{zero} <: \texttt{num}}{F, \emptyset \vdash \texttt{Zero} : \texttt{num}}$$

where `zero` is defined as `G(z, (z -> Zero))`.
We first prove that `Zero` has type `zero` using the rule T-CAPP. Since `Zero` has no subexpression there are no further subderivations in the tree. And then we use the subsumption rule to conclude that `Zero` also has type `num`.

Example 4.10
As a more complex example we show the typing of the following function `plus` that adds two numbers.

```
plus(x : num, y : num) : num =
  match x : num
    case Zero     -> y
    case Succ(x') -> plus(x',Succ(y))
```

The typing rule for programs T-PROG tells us that we need to type the body of the expression with $\Gamma = \{\texttt{x} : \texttt{num}, \texttt{y} : \texttt{num}\}$ as typing context.

Since the outermost expression of the body is a pattern match the derivation tree starts with an instance of the rule T-MATCH.

$$\text{T-MATCH}\ \dfrac{F, \Gamma \vdash \texttt{y} : \texttt{num} \qquad F, \Gamma \cup \{\texttt{x'} : \texttt{num}\} \vdash \texttt{plus(x',Succ(y))} : \texttt{num}}{F, \Gamma \vdash \begin{array}{l}\texttt{match x : num}\\ \quad\texttt{case Zero -> y}\\ \quad\texttt{case Succ(x')}\\ \qquad\texttt{-> plus(x',Succ(y))}\end{array} : \texttt{num}}$$

The patterns `Zero` and `Succ(x')` are exhaustive with respect to `num`. Furthermore, matching the pattern `Succ(x')` against `num` yields the typing context $\{\texttt{x'} : \texttt{num}\}$. The pattern `Zero` also matches the type `num`. But the matching in this case yields the empty typing context because `Zero` contains no pattern variables. Hence, for proving the final result we need the following two subderivations. For simplicity we define Γ' to refer to $\Gamma \cup \{\texttt{x'} : \texttt{num}\}$.

$$\text{T-VAR}\ \dfrac{(\texttt{y} : \texttt{num}) \in \Gamma}{F, \Gamma \vdash \texttt{y} : \texttt{num}}$$

$$\text{T-FAPP}\ \dfrac{\text{T-VAR}\ \dfrac{(\texttt{x'} : \texttt{num}) \in \Gamma'}{F, \Gamma' \vdash \texttt{x'} : \texttt{num}} \qquad \text{T-CAPP}\ \dfrac{\text{T-VAR}\ \dfrac{(\texttt{y} : \texttt{num}) \in \Gamma'}{F, \Gamma' \vdash \texttt{y} : \texttt{num}}}{F, \Gamma' \vdash \texttt{Succ(y)} : \texttt{num}}}{F, \Gamma' \vdash \texttt{plus(x',Succ(y))} : \texttt{num}}$$

The first derivation tree shows that the right hand side of the first case expression, namely `y`, has type `num`. This immediately succeeds with the rule T-VAR, because the typing contexts assigns the type `num` to `y`.

The second derivation tree shows the well-typedness of the right hand side of the second case expression, namely `plus(x',Succ(y))`. Since the expression starts with a function application we have to use the rule T-FAPP in the first place. The result type of the function `plus` is annotated as `num` which matches with the goal we are trying to prove. What remains to be done is to show that the arguments `x'` and `Succ(y)` have the correct type, namely `num`.

In the process of matching the pattern `Succ(x')` against `num` the assignment `x'` : `num` has been added to the typing context. Therefore the subderivation for the first argument succeeds immediately with the rule T-VAR. For the second argument we need an additional instance of the rule T-CAPP to close the prove.

Note, that it is important to use the interface of the function `plus` for checking the function application `plus(x',Succ(y))`. Since we are in the midst of typing the body of that very function it is not possible to use the implementation instead of the interface.

4.4 Soundness of the Type System

To goal of the type system is to define the subset of syntactically correct terms that cannot lead to a runtime error during evaluation. If this property holds for all expression e that can be given a type t the type system is *sound*.

In the following we prove the soundness of the type system as defined by the inference rules in Figure 9. Similar to Pierce [24] the prove requires two: *progress* and *preservation*. Progress means that a well-typed expression is either a value or can take a reduction step according to the operational semantics. Preservation ensures that the well-typedness is an invariant of the reduction relation: whenever an expression e has type t and can reduce to e' in one step, e' also has type t. Progress and preservation together imply the soundness of the type system.

So far, we have defined what it means for a value to be generated by a grammar (see Section 3). The type system on the other hand specifies what it means for an expression to be well-typed with a type t. Values are a subset of the expressions, hence we have implicitly defined what it means for a value to be well-typed with a type t. In the following proofs we need the fact that a value of type t can be derived by the grammar describing the type t. Although the intention is that these two notions coincide, the fact is not obvious and we need to prove it.

Lemma 4.1
$F, \emptyset \vdash v : t \quad \Rightarrow \quad v \in t$

Proof. We prove the statement by structural induction on the typing derivation showing $F, \emptyset \vdash v : t$. As induction hypothesis we can assume that the statement holds for all direct subderivations of the form $F, \emptyset \vdash v : t$. We proceed by case analysis on the final rule in the derivation tree. A value v has the form $C\texttt{(}v_1\texttt{, ... ,}v_n\texttt{)}$. Hence, there are only two matching rules: T-CAPP and T-SUB.

- The last rule is T-CAPP with the conclusion $F, \emptyset \vdash C\texttt{(}v_1\texttt{, ... ,}v_n\texttt{)} : \texttt{G(}s\texttt{,}\pi\texttt{)}$. Furthermore, we know from the premises that $\pi(s) = C\texttt{(}n_1\texttt{, ... ,}n_n\texttt{)}$ and that there are subderivations showing $F, \Gamma \vdash v_i : \texttt{G(}n_i\texttt{,}\pi\texttt{)}$. Hence, by the induction hypothesis it follows that $v_i \in \texttt{G(}n_i\texttt{,}\pi\texttt{)}$. According to the definition of derivations of grammars (see Section 3) this implies immediately that $C\texttt{(}v_1\texttt{, ... ,}v_n\texttt{)} \in \texttt{G(}s\texttt{,}\pi\texttt{)}$.

- The last rule is T-SUB with the conclusion $F, \emptyset \vdash v : t$. This means we have a subderivation showing $F, \emptyset \vdash v : t_{sub}$. By the induction hypothesis it follows that $v \in t_{sub}$. Since $t_{sub} <: t$ we know by the definition of subtyping as inclusion of sets that $v \in t$.

$\square$

Theorem 4.1 : Progress
If $F, \emptyset \vdash e : t$ then either

- e is a value or

- there exists an e' to that $F \vdash e \longrightarrow e'$.

Proof. We prove the statement by structural induction on the typing derivation showing $F, \emptyset \vdash e : t$. As induction hypothesis we can assume that the statement holds for all direct subderivations of the form $F, \emptyset \vdash e : t$. We proceed by case analysis on the final rule in the derivation tree.

- The rule T-VAR cannot occur as last rule in the derivation tree, because it requires a non empty typing context Γ.

- The last rule is T-SUB with the conclusion $F, \emptyset \vdash e : t$. This means we have a subderivation showing $F, \emptyset \vdash e : t_{sub}$ with $t_{sub} <: t$. By the induction hypothesis we know that there exists an expression e' so that $F \vdash e \longrightarrow e'$. Which is exactely what we need in this case.

- The last rule is T-FAPP with the conclusion $F, \emptyset \vdash f\texttt{(}e_1\texttt{, ... ,}e_n\texttt{)} : t$. This implies subderivations of the form $F, \emptyset \vdash e_i : t_i$ for all i. By the induction hypothesis we know that either e_i is as value or can reduce to e_i'. We can distinguish two cases:

  - All e_i are values. The rule T-FAPP tells us in the premises that the function f is defined in F as $f(x_1 : t_1, \dots, x_n : t_n) : t = e$. It follows that the whole expression $f(e_1, \dots, e_n)$ can reduce to $e[x_i \mapsto e_i]$ by FAPP.
  - There exist a smallest i so that all e_j are values for $j < i$ and e_i is not a value. We know by the induction hypothesis that e_i can reduce to e'_i. Hence, by CONG the whole expression reduces to $f(e_1, \dots, e_{i-1}, e'_i, \dots, e_n)$.

- The last rule is T-CAPP with the conclusion $F, \emptyset \vdash C(e_1, \dots, e_n) : t$. We have subderivations showing $F, \emptyset \vdash e_i : t_i$ for all i. Analog to the case for T-FAPP either all e_i are values or can reduce to e'_i. If all e_i are values the whole expression is already a value. Otherwise the whole expression can reduce by CONG.

- The last rule is T-MATCH with the conclusion
  $F, \emptyset \vdash$ `match` e_{match} : t_{match} `case` pat_1 `->` $e_1 \dots$ `case` pat_n `->` $e_n : t$.
  There is a subderivation showing $F, \emptyset \vdash e_{match} : t_{match}$. By the induction hypothesis we know that either e_{match} is a value of can reduce to e'_{match}.
  - If e_{match} reduces to e'_{match} the whole expression reduces by CONG to `match` e'_{match} : t_{match} `case` pat_1 `->` $e_1 \dots$ `case` pat_n `->` e_n.
  - If e_{match} is a value we know by Lemma 4.1 that this value is in t_{match}. Since the patterns pat_1 to pat_n are exhaustive with respect to t_{match} each value in t_{match} will be matched by at least one of the patterns. This means that also e_{match} will be matched by at least one pattern. Let i be the smallest i so that $match(pat_i, e_{match}) = \sigma$. By MATCH the whole expression reduces to $\sigma(e_i)$.

□

In the proof of preservation we need the following Lemma.

Lemma 4.2 : Substitution Lemma
If $F, \Gamma \vdash e_1 : t_1 \dots F, \Gamma \vdash e_k : t_k$ and $F, \Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \vdash e : t$ then $F, \Gamma \vdash e[x_1 \mapsto e_1, \dots, x_k \mapsto e_k] : t$.

Proof. The proof uses a structural induction on the typing derivation of $F, \Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \vdash e : t$.

Let $\sigma = \{x_1 \mapsto e_1, \dots, x_k \mapsto e_k\}$ be the substitution we want to apply. Let us further assume that $F, \Gamma \vdash e_1 : t_1, \dots, F, \Gamma \vdash e_k : t_k$ and that $F, \Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \vdash e : t$.

We proceed by cases analysis on the last rule used in the typing derivation. As inductive hypothesis we can assume the statement to hold for all direct subderivations.

- The last rule is T-VAR.
  This implies that e has the form $e = x$ for a variable x. The definition of substitution tells us that $\sigma(x) = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$ and $\sigma(x) = x$ if $x \notin \mathrm{dom}(\sigma)$. Let's look at both cases separately.

  – If x is in the domain of σ, i.e. there exists an i with $x = x_i$ the result of $\sigma(x)$ is e_i. Since we assumed $F, \Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \vdash x : t$, t has to be equal to t_i and hence $F, \Gamma \vdash \sigma(x) = e_i : t_i = t$.

  – If x is not in the domain of σ then $\sigma(e) = \sigma(x) = x$. From that it follows immediately from the assumptions that $F, \Gamma \vdash \sigma(e) = e : t$.

- The last rule is T-SUB.
  This implies a subderivation showing $F, \Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \vdash e : t_{sub}$ with $t_{sub} <: t$. By the inductive hypothesis it follows that $F, \Gamma \vdash \sigma(e) : t_{sub}$ and by T-SUB that $F, \Gamma \vdash \sigma(e) : t$.

- The last rule is T-CAPP.
  This implies that e has the form $e = C\texttt{(}a_1\texttt{, ... ,}a_n\texttt{)}$. According to the definition of substitution, $\sigma(C\texttt{(}a_1\texttt{, ... ,}a_n\texttt{)})$ yields $C\texttt{(}\sigma(a_1)\texttt{, ... ,}\sigma(a_n)\texttt{)}$. Furthermore, we have subderivations showing $F, \Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \vdash a_i : t_{a_i}$. By the inductive hypothesis it follows that $F, \Gamma \vdash \sigma(a_i) : t_{a_i}$. And hence by T-CAPP $F, \Gamma \vdash C\texttt{(}\sigma(a_1)\texttt{, ... ,}\sigma(a_n)\texttt{)} : t$.

- The last rule is T-FAPP.
  This case is analog to the T-CAPP case.

- The last rule is T-MATCH. This implies that e has the form $e = \texttt{match}\ e_{match}\ \texttt{:}\ t_{match}\ \texttt{case}\ pat_1\texttt{->}\ e_1 \dots\ \texttt{case}\ pat_n\texttt{->}\ e_n$. A subderivation tells us that $F, \Gamma \vdash e_{match} : t_{match}$. Hence by the inductive hypothesis $F, \Gamma \vdash \sigma(e_{match}) : t_{match}$.

  Furthermore, we have subderivations of the form $F, \Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \cup \Gamma_i \vdash e_i : t$ where Γ_i is a typing context with $\mathrm{dom}(\Gamma_i) = \mathrm{vars}(pat_i)$. Recall that we defined the union of typing contexts in a way that the second typing context overwrites the first one if the domains are not disjoint. Let $\{x_{i_1}, \dots, x_{i_l}\} = \{x_1, \dots, x_k\} \setminus \mathrm{dom}(\Gamma_i)$. Hence, we can rewrite $\Gamma \cup \{x_1 : t_1, \dots, x_k : t_k\} \cup \Gamma_i$ to $\Gamma \cup \Gamma_i \cup \{x_{i_1} : t_{i_1}, \dots, x_{i_l} : t_{i_l}\}$.

  By the definition of substitution the substitution σ_i is applied to the subexpression e_i. σ_i is defined as σ without mappings for pattern variables occurring in the pattern pat_i. Hence, $\mathrm{dom}(\sigma_i) = \{x_{i_1}, \dots, x_{i_l}\}$. With these insights we can apply the inductive hypothesis yielding $F, \Gamma \cup \Gamma_i \vdash \sigma_i(e_i) : t$. Together with $F, \Gamma \vdash \sigma(e_{match}) : t_{match}$ we can conclude by T-MATCH that $F, \Gamma \vdash e : t$.

$\square$

Theorem 4.2 : Preservation
$F, \emptyset \vdash e : t$ and $F \vdash e \longrightarrow e'$ implies $F, \emptyset \vdash e' : t$.

Proof. Again, we do a structural induction on the typing derivation showing $F, \emptyset \vdash e : t$. As induction hypothesis we can assume the statement to hold for all direct subderivations of the form $F, \emptyset \vdash e : t$. We proceed by case analysis on the final rule in the derivation tree.

- The rule T-VAR cannot occur as last rule since it requires a non empty typing context Γ.
- The last rule is T-SUB with the conclusion $F, \emptyset \vdash e : t$. This means there is a direct subderivation showing $F, \emptyset \vdash e : t_{sub}$ with $t_{sub} <: t$. If $F \vdash e \longrightarrow e'$ we know by the induction hypothesis that $F, \emptyset \vdash e' : t_{sub}$. But then, by T-SUB, $F, \emptyset \vdash e' : t$ since $t_{sub} <: t$.
- The last rule is T-FAPP with the conclusion $F, \emptyset \vdash f\texttt{(}e_1\texttt{, ... ,}e_n\texttt{)} : t$. From the premises of the rule we know that f is defined as $f\texttt{(}x_1 \texttt{ : } t_1\texttt{, ... ,}x_n \texttt{ : } t_n\texttt{) : } t \texttt{ = } e$. There a two possibilities on how the expression could reduce.
  - There exists a smallest i so that $F \vdash e_i \longrightarrow e'_i$. Since we have a subderivation showing $F, \emptyset \vdash e_i : t_i$ the induction hypothesis applies. This means we can conclude that $F, \emptyset \vdash e'_i : t_i$. Hence, the whole expression reduces by CONG and by T-FAPP the resulting expression $f\texttt{(}e_1\texttt{, ... ,}e_{i-1}\texttt{,}e'_i\texttt{, ... ,}e_n\texttt{)}$ has also type t.
  - All e_i are values and the whole expression reduces by the rule FAPP to $e[x_1 \mapsto e_1, \ldots x_n \mapsto e_n]$. All functions in F are assumed to be well-typed. This implies $F, \{x_1 : t_1, \ldots, x_n : t_n\} \vdash e : t$. Furthermore, we know that $F, \emptyset \vdash e_i : t_i$ from the subderivations. It follows by Lemma 4.2 that $F, \emptyset \vdash e[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] : t$.
- The last rule is T-CAPP with the conclusion $F, \emptyset \vdash C\texttt{(}e_1\texttt{, ... ,}e_n\texttt{)} : t$. The only possibility for the expression to reduce is by the rule CONG. This case is analog to the congruence case for function applications. Hence, by the same argumentation the resulting expression also has type t.
- The last rule is T-MATCH with the conclusion $F, \emptyset \vdash \texttt{match } e_{match} \texttt{ : } t_{match} \texttt{ case } pat_1 \texttt{ -> } e_1 \ldots \texttt{ case } pat_n \texttt{ -> } e_n : t$. There are two possibilities for this expression to reduce.
  - The expression reduces by the rule CONG. This implies that there exists an expression e'_{match} so that $F \vdash e_{match} \longrightarrow e'_{match}$. Hence, by the induction hypothesis we know that $F, \emptyset \vdash e'_{match} : t_{match}$ and thus by T-MATCH $F, \emptyset \vdash \texttt{match } e'_{match} \texttt{ : } t_{match} \texttt{ case } pat_1 \texttt{ -> } e_1 \ldots \texttt{ case } pat_n \texttt{ -> } e_n : t$.

– The expression reduces by the rule MATCH. This means that e_{match} is a value and there exists a smallest i so that match$(pat_i, e_{match}) = \sigma$. Furthermore, the rule T-MATCH tells us that match$(pat_i, t_{match}) = \Gamma_i$. This means that for all pattern variables x of the pattern pat_i $F, \emptyset \vdash \sigma(x) : \Gamma_i(x)$. Recall that matching a pattern against a type returns a typing context assigning the pattern variable x to a type containing all possible values that x might bind to at runtime.

  We have a subderivation showing $F, \Gamma_i \vdash e_i : t$. Hence, by Lemma 4.2 $F, \emptyset \vdash \sigma(e_i) : t$.

□

5 Type Inference and Type Checking

The typing relation as defined by the rules in Figure 9 is not algorithmic. This means that it does not induce a type checking algorithm directly. In this Section we show how to implement a type checking algorithm on the basis of type inference.

5.1 Type Checking

The problem case for implementing a type checker in the basis of the typing rules shown in figure 9 is the subsumption rule T-SUB: the type t_{sub} is only mentioned in the premise but not in the conclusion. This means that when constructing a typing derivation bottum up we have to *guess* t_{sub}. All other rules though are algorithmic.

The rules in figure 9 allow for the subsumption rule to be used at any place in a derivation tree. Especially it would be possible to use the subsumption rule multiple times in a row. Since the subtyping relation is transitive, such rows of subsumption rules in a derivation tree can be collapsed into one instance of subsumption rule without affecting the validity of the derivation. Furthermore, the use of the subsumption rule directly below a rule instance of T-MATCH can be “pushed up” into the subderivations for the subexpressions e_i.

Hence, one could try to inline the subsumption rule into the others in a manner that enables algorithmicity. For the rules T-VAR, T-FAPP and T-MATCH this works pretty well. The rule T-MATCH needs not to be modified at all as mentioned above. The alternative rules for T-VAR and T-FAPP are shown in figure 10.

$$\text{T-VAR'}\ \frac{(x : t_1) \in \Gamma \qquad t_1 <: t}{F, \Gamma \vdash x : t}$$

$$\text{T-FAPP'}\ \frac{\texttt{f(}x_1 \texttt{ : } t_1\texttt{,}\ldots\texttt{,}x_n \texttt{ : } t_n\texttt{) : } t_{res} \texttt{ = } e \in F \qquad F, \Gamma \vdash e_i : t_i \qquad t_{res} <: t}{F, \Gamma \vdash \texttt{f(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{) : } t}$$

Figure 10: Typing Rules with Inlined Subsumption

But the rule T-CAPP poses a problem because of the definition of subtyping. What we know is the following: Any type $t = \texttt{G(}s\texttt{, } pi\texttt{)}$ that an expression $\texttt{C(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}$ might have must contain at least one right hand side of the form $\texttt{C(}n_1\texttt{,}\ldots\texttt{,}n_n\texttt{)}$ in $\pi(s)$. Suppose that $\pi(s)$ contains the matching right hand sides $\texttt{C(}n_{1_1}\texttt{,}\ldots\texttt{,}n_{1_n}\texttt{)}$ to $\texttt{C(}n_{k_1}\texttt{,}\ldots\texttt{,}n_{k_n}\texttt{)}$. The problem is that there might not exists a j so that $e_i : \texttt{G(}n_{j_i}\texttt{, }\pi\texttt{)}$ while still $\texttt{C(}e_1\texttt{, }\ldots\texttt{,}e_n\texttt{)} : t$ is derivable. Hence, subtyping is not structural.

To overcome this problem we will approach the task of designing a type checking algorithm differently. Suppose we had an algorithm that, given an aribtrary expression, it computes the “best” type of this expression. A type t_{sub} of an expression e is the best type for that expression if for all t, $e : t$ implies $t_{sub} <: t$. Hence, the best type contains the all information about types of an expression: from derivation tree for $e : t_{sub}$ we can construct a derivation tree for $e : t$ for all types t that e can have by adding an instance of the subsumption rule at the bottom.

Computing a type for an expression is called *type inference*. Thereby the goal is not to infer an arbitrary type for an expression but to infer the type containing the most information about that expression. In our case most information means to compute the most specific type. We will later see that in other contexts we get different definitions for what most information means.

If we assume an algorithm infer-type that computes the most specific type for an expression, we can define a type checking algorithm as follows:

$$\text{has-type}(F, \Gamma, e, t) = \begin{cases} \text{true} & \text{if infer-type}(F, \Gamma, e) <: t \\ \text{false} & \text{otherwise} \end{cases}$$

5.2 Type Inference

Figure 11 shows the algorithm for type inference. Although the algorithm is written as a set of inference rules, it describes a (partial) function from expressions to types (assuming a set of function definitions and a typing context). The function is partial, because we can only infer a type for a well typed expression. We write $F, \Gamma \vdash e :_{\rightarrow} t$ if the inferred type for the expression e is t.

The rule I-VAR says that the best type we can infer for a variable x is the type assigned to it in the typing context Γ.

The best type we can infer for a function application is the result type of the function since we only use the signature of the function for type checking. In addition, the inferred types for the arguments need to be subtypes of the types annotating the function’s formal arguments. Note the similarity to the rule T-FAPP. The only difference is that in I-FAPP the best type for the arguments is inferred and subtyping is checked already there. If the rule T-FAPP is used within a derivation tree for the same expression the subtyping will have to be checked further up in the subderivations via the rule T-SUB (unless the arguments’ types correspond exactely to the formal arguments’ types).

The best type for a constructor application is the type where all values start with that constructor and the arguments comprise the complete product space of the types inferred for the arguments.

The rule for match expressions I-MATCH is also very similar to its counterpart in the typing relation. The difference to T-MATCH is that in this case the resulting type is computed as the union of the inferred types of the different case expression. Here, it becomes very obvious that in type inference the type is an output of the function. It only depends on the premises and the input.

$$\text{I-VAR}\ \frac{(x : t) \in \Gamma}{F, \Gamma \vdash x :_{\rightarrow} t}$$

$$\text{I-FAPP}\ \frac{\texttt{f(}x_1\ \texttt{:}\ t_1\texttt{,}\ldots\texttt{,}x_n\ \texttt{:}\ t_n\texttt{)}\ \texttt{:}\ t\ \texttt{=}\ e \in F \qquad F, \Gamma \vdash e_i :_{\rightarrow} t_{e_i} \qquad t_{e_i} <: t_i}{F, \Gamma \vdash f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} :_{\rightarrow} t}$$

$$\text{I-CAPP}\ \frac{F, \Gamma \vdash e_i :_{\rightarrow} t_i \qquad \texttt{s}\ \text{fresh}}{F, \Gamma \vdash C\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} :_{\rightarrow} \texttt{G(s, (s -> C(}t_1\texttt{, ... ,}t_n\texttt{)))}}$$

$$\text{I-MATCH}\ \frac{\begin{array}{c} F, \Gamma \vdash e :_{\rightarrow} t_e \qquad t_e <: t \qquad \text{match}(pat_i, t) = \Gamma_i \\ \text{exhaustive}(\{pat_1, \ldots, pat_n\}, t) \qquad F, \Gamma_i \cup \Gamma \vdash e_i :_{\rightarrow} t_i \end{array}}{F, \Gamma \vdash \texttt{match}\ e\ \texttt{:}\ t\ \texttt{case}\ pat_1 \texttt{->}\ e_1 \ldots \texttt{case}\ pat_n \texttt{->}\ e_n :_{\rightarrow} \bigcup t_i}$$

Figure 11: Type Inference for the Simply Typed Language

In the following we will prove that type inference really computes the best type for an expression with respect to the typing relation. The proof will be done in two steps: first, we will show that $F, \Gamma \vdash e :_{\rightarrow} t$ implies $F, \Gamma \vdash e : t$, e.g. we will show that the inferred type is really type for that expression as justified by the typing relation. And second, we will show that for all types t, $F, \Gamma \vdash e : t$ implies that the inferred type for e is a subtype of t.

To simplify the notation we define $\text{infertype}_{F,\Gamma}(e)$ to refer to the type t so that $F, \Gamma \vdash e :_{\rightarrow} t$. As discussed above, the inference rules actually define a function so that $\text{infertype}_{F,\Gamma}(e)$ is well defined.

Theorem 5.1
$F, \Gamma \vdash e :_{\rightarrow} t \Rightarrow F, \Gamma \vdash e : t$

Proof. We will show that we can transform a derivation tree for $F, \Gamma \vdash e :_{\rightarrow} t$ into a derivation tree yielding $F, \Gamma \vdash e : t$ by structural induction on the inference derivation tree. This means that by induction hypthesis we can assume the statement $F, \Gamma \vdash e :_{\rightarrow} t \Rightarrow F, \Gamma \vdash e : t$ to hold for all direct subderivations of the form $F, \Gamma \vdash e :_{\rightarrow} t$. Using this we have to show that we can construct a derivation tree witnessing the final conclusion. We approach that step by case analysis on the last rule used in the inference derivation tree.

<u>I-VAR</u>: The instance derivation tree

$$\text{I-VAR}\ \frac{x : t \in \Gamma}{F, \Gamma \vdash x :_{\rightarrow} t}$$

becomes

$$\text{T-VAR}\ \frac{x : t \in \Gamma}{F, \Gamma \vdash x :_{\rightarrow} t}$$

I-FAPP: in this case the derivation tree ends with

$$\text{I-FAPP}\ \dfrac{\mathtt{f(}x_1\ \mathtt{:}\ t_1\mathtt{,}\ldots\mathtt{,}x_n\ \mathtt{:}\ t_n\mathtt{)}\ \mathtt{:}\ t\ \mathtt{=}\ e \in F \qquad \dfrac{\vdots}{F,\Gamma \vdash e_i :_\rightarrow t_{e_i}} \qquad t_{e_i} <: t_i}{F,\Gamma \vdash \mathtt{f(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)} :_\rightarrow t}$$

with subderivations showing $F,\Gamma \vdash e_i :_\rightarrow t_{e_i}$. By the induction hypthesis we know that there exist derivation trees showing $F,\Gamma \vdash e_i : t_{e_i}$. With that we can construct the following derivation tree for the complete expression:

$$\text{T-FAPP}\ \dfrac{\mathtt{f(}x_1\ \mathtt{:}\ t_1\mathtt{,}\ldots\mathtt{,}x_n\ \mathtt{:}\ t_n\mathtt{)}\ \mathtt{:}\ t\ \mathtt{=}\ e \in F \qquad \dfrac{\dfrac{\vdots}{F,\Gamma \vdash e_i : t_{e_i}} \qquad t_{e_i} <: t_i}{F,\Gamma \vdash e_i : t_i}\ \text{T-SUB}}{F,\Gamma \vdash \mathtt{f(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}\ \mathtt{:}\ t}$$

I-CAPP: This case is analog to the I-FAPP case, only that we don't need the instance of the subsumption rule here.

I-MATCH: in this case the derivation tree ends with

$$\text{I-MATCH}\ \dfrac{\begin{array}{c}\dfrac{\vdots}{F,\Gamma \vdash e :_\rightarrow t_e} \qquad t_e <: t \qquad \dfrac{\vdots}{F,\Gamma_i \cup \Gamma \vdash e_i :_\rightarrow t_i} \\ \text{match}(pat_i, t) = \Gamma_i \qquad \text{exhaustive}(\{pat_1, \ldots, pat_n\}, t)\end{array}}{F,\Gamma \vdash \mathtt{match}\ e\ \mathtt{:}\ t\ \mathtt{case}\ pat_1 \mathtt{->}\ e_1 \ldots \mathtt{case}\ pat_n \mathtt{->}\ e_n :_\rightarrow \bigcup t_i}$$

with subderivations showing $F,\Gamma \vdash e :_\rightarrow t_e$ and $F,\Gamma \vdash e_i :_\rightarrow t_i$. By the induction hypthesis we know that there exist derivation trees showing $F,\Gamma \vdash e : t_e$ and $F,\Gamma \vdash e_i : t_i$. With the observation that $t_i <: \bigcup t_i$ for all i we can construct the corresponding typing derivation tree:

$$\text{T-MATCH}\ \dfrac{\begin{array}{c}\text{T-SUB}\ \dfrac{\dfrac{\vdots}{F,\Gamma \vdash e : t_e} \qquad t_e <: t}{F,\Gamma \vdash e : t} \qquad \dfrac{\dfrac{\vdots}{F,\Gamma \vdash e_i : t_i} \qquad t_i <: \bigcup t_i}{F,\Gamma_i \cup \Gamma \vdash e_i :_\rightarrow \bigcup t_i}\ \text{T-SUB} \\ \text{match}(pat_i, t) = \Gamma_i \qquad \text{exhaustive}(\{pat_1, \ldots, pat_n\}, t)\end{array}}{F,\Gamma \vdash \mathtt{match}\ e\ \mathtt{:}\ t\ \mathtt{case}\ pat_1 \mathtt{->}\ e_1 \ldots \mathtt{case}\ pat_n \mathtt{->}\ e_n :_\rightarrow \bigcup t_i}$$

□

Theorem 5.2
$F,\Gamma \vdash e : t \Rightarrow \text{infertype}_{F,\Gamma}(e) <: t$.

Proof. For this proof we use the result from the last Theorem, namely how to construct a derivation tree showing $F,\Gamma \vdash e : \text{infertype}_{F,\Gamma}(e)$.

Furthermore, we observe the following: for each subexpression of the expression e all derivation trees witnessing $F,\Gamma \vdash e : t$ for some type t contain exactely one instance of the corresponding typing rule. This means, for each subexpression that is a variable there is exactely one instance of the T-VAR rule, for each subexpression that is a function application there is exactely one instance of the T-FAPP rule, etc. This follows directly from the fact that subderivations in these rules are always conducted on direct subexpressions. In addition, the order of rule instances in the derivation tree corresponds to the respective nesting of subexpressions.

Example 5.1
Any derivation tree witnessing $F, \Gamma \vdash \mathtt{C(f(x),y)} : t$ will have the following form.

$$\dfrac{\begin{array}{c}\text{T-CAPP}\ \dfrac{\begin{array}{c}\text{T-FAPP}\ \dfrac{\begin{array}{c}\text{T-VAR}\ \dfrac{(\mathtt{x} : t_x) \in \Gamma}{F, \Gamma \vdash \mathtt{x} : t_x}\\ \vdots\end{array}}{F, \Gamma \vdash \mathtt{f(x)} : t_2}\\ \vdots\end{array} \qquad \begin{array}{c}\text{T-VAR}\ \dfrac{(\mathtt{y} : t_y) \in \Gamma}{F, \Gamma \vdash \mathtt{y} : t_y}\\ \vdots\end{array}}{F, \Gamma \vdash \mathtt{C(f(x),y)} : t_1}\\ \vdots\end{array}}{F, \Gamma \vdash \mathtt{C(f(x),y)} : t}$$

Hence, two derivation trees witnessing $F, \Gamma \vdash e : t$ and $F, \Gamma \vdash e : t'$ can only differ by instances of the subsumption rule.

Using this insight, we proof $F, \Gamma \vdash e : t \Rightarrow \text{infertype}_{F,\Gamma}(e) <: t$ by showing that we can transform any derivation tree witnessing $F, \Gamma \vdash e : t$ into the derivation tree witnessing $F, \Gamma \vdash e : \text{infertype}_{F,\Gamma}(e)$ followed by one instance of the subsumption rule.

By construction, the derivation tree T_1 showing $F, \Gamma \vdash e : \text{infertype}_{F,\Gamma}(e)$ only contains instances of the subsumption rule directly above the rule T-FAPP and the rule T-MATCH. More precisely, T_1 contains exactely one instance of the subsumption rule for each argument of a function application. Furthermore, it contains one instance of the subsumption rule for the subderivation showing that the expression matched against has the annoated type. And finally there are instances of the subsumption rule witnessing that the inferred type for each right hand side e_i of the case expression i is a subtype of the union of the inferred types for all e_i.

The derivation tree T_2 showing $F, \Gamma \vdash e : t$ on the other hand can contain instances of the subsumption rule at arbitrary places.

Since both trees can only differ by instances of the subsumption rule we have to show that we can recursively push additional instances of the subsumption rule in T_2 further down. As already observed, multiple instances of the subsumption rule in a row can be collapsed into one single instance.

Therefore, we can assume that T_2 does not contain multiple instances of the subsumption rule in a row.

Furthermore, the type annotations in function definitions and match expressions act as a stopper for pushing subsumption rule instances further down the tree. Hence, we need to consider only the following two cases:

- T_2 contains an instance of the subsumption rule directly above an instance of T-CAPP.
- T_2 contains an instance of the subsumption rule directly above an instance of T-MATCH in a subderivation for a right hand side of a case expression. Although T_1 can contain instances of the subsumption rule at these places the instances in T_2 might differ from those in T_1 because there are no type annotations constraining the use of subsumption rules.

Let's look at these thow cases separately. Thereby we assume that all subderivations are already transformed into the respective subderivations of T_1 followed by one instance of the rule T-SUB.

- T-CAPP:
  This means we have an extract of T_2 looking as follows.

  $$\text{T-CAPP}\ \frac{\dfrac{F,\Gamma \vdash e_i : t_{sub_i} \qquad t_{sub_i} <: t_i}{F,\Gamma \vdash e_i : t_i}\ \text{T-SUB}}{F,\Gamma \vdash C\mathtt{(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}\ \mathtt{:}\ \mathtt{G(}s_t\mathtt{,}\ \mathtt{(}s_t\ \mathtt{->}\ C\mathtt{(}t_1\mathtt{,}\ldots\mathtt{,}t_n\mathtt{)))}}$$

  We can replace this by the equivalent extract

  $$\text{T-SUB}\ \frac{\text{T-CAPP}\ \dfrac{F,\Gamma \vdash e_i : t_{sub_i}}{F,\Gamma \vdash C\mathtt{(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}\ \mathtt{:}\ G_1} \qquad G_1 <: G_2}{F,\Gamma \vdash C\mathtt{(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}\ \mathtt{:}\ G_2}$$

  where
  $G_1 = \mathtt{G(}s_t\mathtt{,}\ \mathtt{(}s_t\ \mathtt{->}\ C\mathtt{(}t_{sub_1}\mathtt{,}\ldots\mathtt{,}t_{sub_n}\mathtt{)))}$ and
  $G_2 = \mathtt{G(}s_t\mathtt{,}\ \mathtt{(}s_t\ \mathtt{->}\ C\mathtt{(}t_1\mathtt{,}\ldots\mathtt{,}t_n\mathtt{)))}$.

- T-MATCH: This means we have an extract of T_2 looking as follows.

  $$\text{T-MATCH}\ \frac{\cdots \qquad \dfrac{F,\Gamma \vdash e_i : t_i \qquad t_i <: t}{F,\Gamma \vdash e_i : t}\ \text{T-SUB}}{F,\Gamma \vdash \mathtt{match}\ e_m\ \mathtt{:}\ t_m\ \mathtt{case}\ pat_1\mathtt{->}\ e_1 \ldots\ \mathtt{case}\ pat_n\mathtt{->}\ e_n : t}$$

  Since we assumed that all subderivations are already transformed into the corresponding subderivations of T_1 followed by one instance of the subsumption rule, we know that $t_i = \text{infertype}_{F,\Gamma}(e_i)$. This implies we can replace this extract of T_2 by the equivalent extract

$$\text{T-CAPP}\ \frac{\cdots \qquad F,\Gamma \vdash e_i : t_i}{F,\Gamma \vdash \texttt{match}\ e_m\ \texttt{:}\ t_m\ \texttt{case}\ pat_1 \texttt{->}\ e_1 \ldots\ \texttt{case}\ pat_n \texttt{->}\ e_n : \bigcup t_i}$$
$$\text{T-SUB}\ \frac{\bigcup t_i <: t}{F,\Gamma \vdash \texttt{match}\ e_m\ \texttt{:}\ t_m\ \texttt{case}\ pat_1 \texttt{->}\ e_1 \ldots\ \texttt{case}\ pat_n \texttt{->}\ e_n : t}$$

$\square$

The last two Theorems together imply the correctness of type checking defined in terms of inference. Hence we can use the following algorithms as a type checker for expressions of the simply typed language.

$$\text{has-type}(F,\Gamma,e,t) := \begin{cases} \text{true} & \text{if infertype}_{F,\Gamma}(e) <: t \\ \text{false} & \text{otherwise} \end{cases}$$

6 Case Study I: Evaluators

So far we have only seen very small example programs. The goal of the following case study is to demonstrate that the simply typed language is suited to implement more complex term transformations. In this case study we show how to write an evaluator for a small programming language.

But while this case study illustrates the applicability of the simply typed language for complex term transformation it also shows some limitations. These limitations motivate further extension of the simply typed language.

We start with a very small language of arithmetic expressions. The syntax of the language is defined by the following grammar.

```
arith-expr ::= G(ae, (ae -> Zero | Succ(ae) | Plus(ae,ae)))
```

It consists of the literal `Zero` and the constructors `Succ` and `Plus` that can construct more complex arithmetic expressions out of smaller ones.

Example 6.1
`Zero` and `Succ(Plus(Succ(Zero),Zero))` are examples for arithmetic expressions as defined by the grammar.

When writing an evaluator for a language we need a target domain, i.e. we need to specify the range of values of that language. In the case of arithmetic expressions it makes sense to choose the natural numbers as target domain. Hence, our evaluator for arithmetic expressions takes an arithmetic expression as argument and produces a value of type `num`:

```
eval-ae(e : arith-expr) : num = ...
```

Now, let's look at the body of the implementation.

```
eval-ae(e : arith-expr) : num =
  match e : arith-expr
    case Zero      -> Zero
    case Succ(y)   -> Succ(eval-ae(y))
    case Plus(y,z) -> plus(eval-ae(y),
                           eval-ae(z))
```

We need to pattern match on the input `e`. If `e` is `Zero` we are done, because `Zero` is already a value. In the other cases of `Succ` and `Plus` we make a recursive call of `eval-arith-expr` on the respective subexpressions. In the `Succ` case the recursive call returns a number and we can construct the result by applying the constructor `Succ` to that number. In the `Plus` case we get back two numbers from the recursive calls. For combining these two numbers into the final result we need the helper function `plus`.

```
plus(x : num, y : num) : num =
  match x : num
    case Zero    -> y
    case Succ(z) -> plus(z, Succ(y))
```

The function `plus` successively shifts the `Succ` constructors from its first argument `x` to its second argument `y`.

Both functions `eval-arith-expr` and `plus` are well-typed. Note the interplay between the type annotations of the two functions. The function `eval-arith-expr` uses the result of `plus` as its own result. This is only possible because the type annotations of the function `plus` guarantee that the result has type `num`.

Example 6.2
The program

```
eval-ae(
  (Succ(Plus(
          Succ(Succ(Zero)),
          Plus(
            Succ(Zero),
            Succ(Succ(Zero))))))
```

is well typed and yields the value

```
Succ(Succ(Succ(Succ(Succ(Succ(Zero))))))
```

after evaluation as expected.

Next, we extend the language of arithmetic expression with a predecessor construct.

```
arith-expr-p ::= G(aep, (aep -> Zero | Succ(aep) | Pred(aep)))
```

We leave out the `Plus` constructor so simplify the presentation.

Example 6.3
This grammar for example generates the values

```
Pred(Succ(Zero))
```

or

```
Succ(Succ(Pred(Succ(Zero))))
```

But we can also build an expression of the form

```
Pred(Zero)
```

If we want to evaluate an `arith-expr-p` to a value of type `num` this expression presents a problem. Since what should the predecessor of `Zero` be in the context of the natural numbers?

As the last example has shown, the grammar for the language of `arith-expr-p` allows to construct terms that "make no sense". The simply typed language we are using as meta-language does only allow to write total functions. Hence, we have to explicitly encode the possibility of failing when writing an evaluator for expressions of type `arith-expr-p`. This means we also have to extend the target domain by a value representing the case of no result.

```
maybe-num ::= G(mn, (mn -> None | Just(n),
                      n  -> Zero | Succ(n)))
```

In the style of the Haskell datatype `Maybe a`, we define a `maybe-num` to be either the literal `None` or `Just` a value of type `num`. Note that the definition of `num` is inlined into the grammar of `maybe-num`.

With this definition it is straight forward to write an evaluator for `arith-expr-p`.

```
eval-aep(e : arith-expr-p) : maybe-num =
  match e : arith-expr-p
    case Zero -> Just(Zero)
    case Succ(x) -> match eval-aep(x) : maybe-num
                      case None    -> None
                      case Just(y) -> Just(Succ y)
    case Pred(x) -> match eval-aep(x) : maybe-num
                      case Just(Succ(y)) -> Just(y)
                      case y             -> None
```

This implementation shows two important things:

- The type annotations make it obvious that `eval-aep` is in fact a partial function on expressions of type `arith-expr-p`. Furthermore, the simply typed language does enforce these explicit annotations. Assume we try to implement the function `eval-aep` as a function with result type `number`. This means the case for `Pred` looks as follows:

  ```
  case Pred(x) -> match eval-aep(x) : num
                    case Succ(y) -> y
  ```

  But then the function does not pass the type checker, because the patterns are not exhaustive with respect to the type annotation `num`. For the patterns to be exhaustive we would need the type annotation `positive-num`. But in this case the program would not be well-typed either, because the result of the recursive call `eval-aep(x)` does not have the type `positive-num`.

- The obligation to explicitly deal with partial functions by appropriate type annotations is a good thing from the point of view of safety. But if we look at the implementation of `eval-aep` we see that it leads to much boilerplate code: First, we need to unwrap subresult every time before processing them. And second, a subresult of `None` is always propagated to the final result. One paradigm of good programming style is to never repeat oneself and instead abstract over common behavior. Unfortunately, the simply typed language does not contain mechanisms to abstract over these kinds of common behavior. One way to tackle that shortcoming is to add higher order functions to the language. With higher order functions we could write a function `bind` of the form

  ```
  bind(x : maybe-num,
       f : number -> maybe-num) : maybe-num =
    match x : maybe-num
      case None    -> None
      case Just(y) -> f(y)
  ```

abstracting over the common parts of `eval-aep`. With the help of `bind` we could rewrite `eval-aep` to the more concise version `eval-aep'`.

```
eval-aep'(e : arith-expr-p) : maybe-num =
  match e : arith-expr-p
    case Zero    -> Just(Zero)
    case Succ(x) -> bind(eval-aep'(x),j-succ)
    case Pred(x) -> bind(eval-aep'(x),m-pred)
```

where

```
j-succ(n : num) : maybe-num = Just(Succ(n))

m-pred(n : num) : maybe-num =
  match n : num
    case Zero    -> None
    case Succ(x) -> Just(x)
```

In Section 7 we examine the addition of higher order functions to the simply typed language in more detail.

In our next example we extend the arithmetic expressions to a more realistic expression language.

```
expr ::= G(e, (e -> True | False | If(e,e,e)
                | Zero | Succ(e) | IsZero(e)))
```

In addition to the arithmetic expressions this language contains the boolean values `True` and `False`. Furthermore, we have the control structure `If` encoding the well known if-then-else statement. And finally we have the term `IsZero` that tests if the argument is zero. We again omit the `Plus` constructor from the first example, because is shows nothing new in this case.

The values of the expression language hence are

```
val ::= G(v, (v -> True | False | Zero | Succ(n),
              n -> Zero | Succ(n)))
```

Again, the grammar for the language of expressions allows to build meaningless terms like

```
Succ(True)
```

or

```
If(Succ(Zero),False,True)
```

This means we have to explicitly deal with the case of failing in the evaluator for `expr`. Therefore, we enhance the values with a value representing failure.

```
maybe-val ::= (mv, (mv -> None | Just(v),
                    v -> ...))
```

Now, let's look at the evaluator.

```
eval-expr (e : expr) : maybe-val =
  match e : expr
    case Zero  -> Just(Zero)
    case True  -> Just(True)
```

```
      case False -> Just(False)
      case If(test,then,else) ->
        match eval-expr(test) : maybe-val
          case Just(True)  -> eval-expr(then)
          case Just(False) -> eval-expr(else)
          case y           -> None
      case Succ(y) ->
        match eval-expr(y) : maybe-val
          case Just(Zero)    -> Just(Succ(Zero))
          case Just(Succ(z)) -> Just(Succ(Succ(z)))
          case y             -> None
      case IsZero(y) ->
        match eval-expr(y) : maybe-val
          case Just(Zero)    -> Just(True)
          case Just(Succ(z)) -> Just(False)
          case z             -> None
```

Like the evaluator for `arith-expr-p` this evaluator contains much boilerplate code. We have to wrap and unwrap the results all the time. But even if evaluating a subexpression yields a value we have to check if the result has the correct type to continue. Take for example the case for the constructor `If`. The result of evaluating the subexpression `test` can only be used if it is either `True` or `False`. In all other cases the whole expression is meaningless and we have to return `None`.

In the context of algebraic datatypes writing an evaluator for languages similar to `expr` is a common motivating example for introducing generalized algebraic datatypes [17, 4]. The main problem is that the grammar for the language (represented by an algebraic datatype) allows to construct meaningless terms.

But we are using tree grammars instead of algebraic datatypes. Tree grammars are more powerful than algebraic datatypes as discussed in Section 8. Hence, the question arises if we can design a better grammar that only generates the desired class of terms. For the language `expr` at least the answer is yes.

```
expr' ::= G(e,
            e  -> True | False | IsZero ae
                | Zero | Succ(ae)
                | If(be,ae,ae) | If(be,be,be)
            ae -> Zero | Succ(ae) | If(be,ae,ae)
            be -> True | False | IsZero ae
                | If(be,be,be))
```

The idea is to divide the grammar into two disjoint sets of expressions: arithmetic expressions represented by the nonterminal `ae` and boolean expressions represented by the nonterminal `be`. The whole grammar is defined as the union of arithmetic and boolean expressions. The productions for the start symbol `e` are the result of normalizing the grammar. `expr'` is a subtype of `expr` which means that all terms of `expr'` are also terms of `expr`. But the grammar `expr'` allows to only construct terms which can be evaluated to a value. We prove this by implementing a total evaluator.

Before implementing the evaluator for the whole language, we implement two mutually recursive sub-evaluators for arithmetic and boolean expressions. The sub-evaluators have more specific target domains. Arithmetic expressions evaluate to numbers, and boolean expressions evaluate to boolean values represented by the grammar

```
bool ::= G(b, b -> True | False)
```

To simplify the code we define the following two grammars.

```
ae ::= G(ae, (ae -> ...,
              be -> ...))
be ::= G(be, (be -> ...,
              ae -> ...))

eval-ae(e : ae) : num =
  match e : ae
    case Zero    -> Zero
    case Succ(x) -> Succ(eval-ae(x))
    case If(test,then,else) ->
      match eval-be(test) : bool
        case True  -> eval-ae(then)
        case False -> eval-ae(else)

eval-be(e : be) : bool =
  match e : be
    case True  -> True
    case False -> False
    case IsZero(y) ->
      match eval-ae(y) : num
        case Zero    -> True
        case Succ(y) -> False
    case If(test,then,else) ->
      match eval-be(test) : bool
        case True  -> eval-be(then)
        case False -> eval-be(else)
```

Together with these two sub-evaluators it is now straight forward to implement an evaluator for the whole language of expressions.

```
eval-e(e : expr') : val =
  match e : expr'
    case True      -> eval-be(True)
    case False     -> eval-be(False)
    case IsZero(x) -> eval-be(IsZero(x))
    case Zero      -> eval-ae(Zero)
    case Succ(x)   -> eval-ae(Succ(x))
    case If(test,then,else) ->
      match eval-be(test) : bool
        case True  -> eval-e(then)
        case False -> eval-e(else)
```

In order to make the structure of the code more explicit the evaluator delegates to the respective sub-evaluators as soon as possible. In most cases the topmost constructor unambiguously identifies the class of expressions the term belongs to. The only exception is the constructor `If`. In this case the function `eval-e` has to do the evaluation itself until it can identify the term to either be an arithmetic or a boolean expression.

Hence, for the language of expressions `expr` we succeeded in writing a total evaluator. This was due to the fact that we could design a grammar representing only well-typed expressions. Put otherwise, we embedded the language of expressions in a type-safe way into our meta language. Is this always possible? The answer is clearly no. As discussed in section 1 most programming languages need to ensure context sensitive properties. But like context free grammars, tree grammars are not able to capture context sensitive constraints. For example, if we enrich our small language of expressions by first class functions, there is no way to capture the constraints of variable bindings within the grammar. There is ongoing research about type-safe embedding of programming languages into others [11, 25, 1, 26, 10]. An opportunity for further research is to explore whether tree grammars can be extended in a way to support a broader range of programming languages to be embedded in a type-safe way. Some ideas are discussed in Section 12.

When comparing the implementations of `eval-e` and the sub-evaluators `eval-ae` and `eval-be` we can see that they all contain very similar code for the `If` case. Higher order functions alone would not help to abstract over the common parts in this case. To see why let's try to implement a function `if` capturing the common behavior of evaluating an `If` expressions.

```
if(test : be, then : ?, else : ?, action : ? -> ?) : ?
  match eval-be(test) : be
    case True  -> action(then)
    case False -> action(else)
```

The problem is that we do not know what types to insert in the places of the question marks. If we want to call the function from `eval-ae` we would have to annotate `then` and `else` with `ae` and `action` with `ae -> num`. And the result of the function would have to be `num`. If we want to call the function from `eval-be` we would get different requirements on the types.

In Section 9 we discuss how to extend our language with generic types. This enables us to write code that is type independent, i.e. that can be used in different contexts with different types.

7 First Extension: Higher Order Functions

Higher order functions are functions that take other functions as arguments. Furthermore, higher order functions can also return functions as result. Put differently, in systems with higher order functions functions are first class entities.

So far, the simply typed language can only work with trees. Functions consume and produce trees. And we have language constructs to deconstruct trees and build new trees. Extending the simply typed language with higher order functions means enriching the data by function values.

Higher order functions provide us with a new kind of abstraction mechanism. First order functions abstract over values. This enables us to use the same code with different values. Higher order functions lift this abstraction mechanism to functions. This means we can now write code that abstracts over behavior.

The case study in Section 6 showed a motivating example for a context where it was important to abstract over concrete behavior. We defined the type

```
maybe-num ::= G(mn, (mn -> None | Just(n),
                     n  -> Zero | Succ(n)))
```

describing values that are either a number or `None` representing "no result". This type was used to encode partial functions, i.e. functions that are not defined for all values of the argument type. In the course of writing such a partial function we often had to write code of the form

```
match some-expr : maybe-num
  case None    -> None
  case Just(n) -> ...
```

as to propagate the failure. To abstract over the common parts in this kind of code fragments we need to abstract over the right hand side of the second case expression (indicated by ... in the example). This right hand side depends on the pattern variable `n`. Hence, we need to abstract over a function. In a system that supports higher order functions we can write the function `bind` abstracting over the common parts of the code fragments.

```
bind(x : maybe-num,
     f : number -> maybe-num) : maybe-num =
  match x : maybe-num
    case None    -> None
    case Just(y) -> f(y)
```

In the reminder of this Section we extend the simply typed language by higher order functions. Therefor we first extend the syntax and the semantics of the language. After that we extend the type system to incorporate higher order functions and finally we prove the soundness of the resulting system.

7.1 Syntax

In the simply typed language functions are defined separately at the top level. Hence, functions play a special role in this system. Especially, function definitions are not part of the expression language. The only possibility to make use of the definitions is via function application. In contrast, a system with higher order functions treats functions as first class entities. Hence, we need syntax to write down function values.

Since functions are now part of the expression language we don't need function definitions as top level construct any more. Hence, a program in the new system consist of a single expression. We will later see that we can treat the explicit function definitions from the simply typed languages as syntactic sugar.

Figure 12 shows the extension of the expression language by function literals. We adopt the common notation from the simply typed lambda calculus. A function literal of the form `λx : t. e` defines a function with an argument `x` of type `t` and the body `e`. We allow to use additional parentheses to group subexpressions. Furthermore, we adopt the convention that the function body extends as far to the right as possible. Hence, `λx : t. e(y)` means `λx : t. (e(y))` and not `(λx : t. e)(y)`. Finally, we define function applications to be left associative. Hence, `f(g)(h)` means `(f(g))(h)`. In addition to the new syntax for function literals, the syntax for function applications need to be changed slightly. Since we now have expressions that can *evaluate* to a function we allow arbitrary expressions at the function position instead of only function names.

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \ldots \\
 & & \mid \lambda x : t.\,\mathit{expr} \\
 & & \mid \mathit{expr}(\mathit{expr})
\end{array}
$$

Figure 12: Expressions with Function Literals

Example 7.1
The following example defines a function taking a number as argument and returning `True` if the number is `Zero`.

```
λx : num.
  match x : num
    case Zero    -> True
    case Succ(y) -> False
```

With this extension of the expression language we can now supply functions as arguments to functions. Since we require to annotate types for the arguments, we also need to extend the type language with function types. The extension is shown in figure 13. The old types from the simply typed language become base types in the extended system. A type is then defined to be either a base type or a function type written as t_1 `->` t_2. Thereby the type t_1 represents the type of the function's argument and t_2 the result type. We adopt the convention that "`->`" is right associative. Hence, t_1 `->` t_2 `->` t_3 means t_1 `->` `(`t_2 `->` t_3`)`. Parenthesis can be used to group types.

$$\begin{array}{lcl} t & ::= & \mathit{base} \mid t \texttt{ -> } t \\ \mathit{base} & ::= & \texttt{G(}n\texttt{, (}\mathit{prod}\texttt{,}\ldots\texttt{,}\mathit{prod}\texttt{))} \end{array}$$

Figure 13: Function Types

Note, that formally we only allow functions with one argument. Since functions can return functions as result this is no restriction. A function with more than one argument can be encoded as multiple nested functions each taking only one argument. For example, we encode a function of the form

```
f(x:t1, y:t2) = body
```

by

```
λx:t1. λy:t2. body
```

As discussed in Section 2, it is essential that a language working on inductively defined trees offers recursive functions as a unit of abstraction. So far, we have no possibility to express function literals of recursive functions. The problem is that literal functions are anonymous. But for recursive calls we need to name that very function we are defining. In the simply typed language without higher order functions this problem did not occur, because there we give names to functions when defining them. In a system with higher order functions the problem is commonly solved by introducing a fixpoint construct `fix`.[24]. `fix` takes a function as argument and produces a new function in the following way. `fix(λf : t1 -> t2. e)` evaluates to `e` where `f` is replaced by the whole expression `fix(λf : t1 -> t2. e)`. The syntax is shown in figure 14 We explain the functionality by a small example.

$$\begin{array}{lcl} \mathit{expr} & ::= & \ldots \\ & & \mid \texttt{fix(}\mathit{expr}\texttt{)} \end{array}$$

Figure 14: Syntax for Fixpoints

Example 7.2
We can approximate a recursive function by abstracting over the function we want to use for the recursive call. The following code shows such an approximation for the function `even`.

```
λeven : num -> bool.
  λx : num.
    match x : num
      case Zero           -> True
      case Succ(Zero)     -> False
      case Succ(Succ(y)) -> even(y)
```

The fixpoint construct ties the knot to get true recursion. The recursive function `even` is defined as

```
fix(λeven : num -> bool.
      λx : num.
        match x : num
          case Zero           -> True
          case Succ(Zero)     -> False
          case Succ(Succ(y)) -> even(y))
```

which evaluates to

```
λx : num.
  match x : num
    case Zero           -> True
    case Succ(Zero)     -> False
    case Succ(Succ(y)) -> fix(...)(y)
```

When applying this result for example to the value `Succ(Succ(Zero))` it reduces to

```
fix(...)(Zero)
```

which again reduces to

```
(λx : num.
   match x : num
     case Zero           -> True
     case Succ(Zero)     -> False
     case Succ(Succ(y)) -> fix(...)(y))
(Zero)
```

Hence, the `fix` construct ensures that we have as many instances of the recursive function's body available as we need.

The fixpoint construct `fix` also allows to encode sets of mutually recursive functions. We illustrate this by the following example.

Example 7.3
We consider the mutually recursive functions `even` and `odd`.

```
even(x : num): bool
  match x : num
    case Zero    -> True
    case Succ(y) -> odd(y)

odd(x : num): bool
  match x : num
    case Zero    -> False
    case Succ(y) -> even(y)
```

We define the grammar

```
s ::= G(s, (s -> Even | Odd))
```

containing one constructor per function. With that we can define the following generator function.

```
g ::= λgen : s -> (num -> bool).
        λf : s.
          match f : s
            case Even -> λx : num.
                           match x : num
                             case Zero    -> True
                             case Succ(y) -> gen(Odd)(y)
            case Odd  -> λx : num.
                           match x : num
                             case Zero    -> False
                             case Succ(y) -> gen(Even)(y)
```

With the help of this generator function we can now encode `even` and `odd` as

```
even ::= fix(g)(Even)
odd  ::= fix(g)(Odd)
```

As we have seen in the previous examples we can encode functions with more than one argument as well as (mutually) recursive functions in our new system. Hence, a program in the simply typed language containing to level function definition can be seen as syntactic sugar. The desugaring works as follows. We start with a program of the form

$$\begin{array}{l} f_1(x_{1_1} : t_{1_1}, \ldots, x_{1_{n_1}} : t_{1_{n_1}}) : t_1 = e_1 \\ \quad \ldots \\ f_k(x_{1_k} : t_{1_k}, \ldots, x_{k_{n_k}} : t_{k_{n_k}}) : t_k = e_k \\ e : t \end{array}$$

First, we define an anonymous function literal for each function definition f_i as follows.

$$f_i \quad ::= \quad \lambda x_{1_i} \texttt{ : } t_{1_i} \texttt{.} \ldots \texttt{.} \lambda x_{i n_i} \texttt{ : } t_{i n_i} \texttt{.}\ e$$

If necessary we encode sets of mutually recursive function with the help of a generator function as explained in example 7.3. With the help of these definitions we can desugar the whole program to the following program in the higher order system.

$$\begin{array}{l} \texttt{(}\lambda f_1 \texttt{ : } t_{1_1} \texttt{->} \ldots \texttt{->}\ t_{1_{n_1}} \texttt{.} \\ \quad \ldots \texttt{.} \\ \ \lambda f_k \texttt{ : } t_{1_k} \texttt{->}\ \ldots \texttt{->}\ t_{k_{n_k}} \texttt{.}\ e\texttt{)(}f_1\texttt{)} \ldots \texttt{(}f_n\texttt{)} \texttt{ : } t \end{array}$$

Regarding top level definitions as syntactic sugar allows us to use this more readable syntax for examples. The formal analysis of the language with higher order function sticks to the smaller core language.

7.2 Semantics

In order to give a meaning to the new syntactic constructs we need to extend the semantics of the language. Therefor, we first have to define which expressions are values. Clearly, all tree values should be values. The same holds for function literals. But what about expressions of the form $C\texttt{(}\lambda x \texttt{ : } t\texttt{.}\ e\texttt{)}$? In a system that works with trees and functions on trees this kind of expression does not make any sense, since it is neither a tree nor a function. Therefore we choose to define the set of values to not contain these "mixed" forms. It will be the task of the type system to reject such terms as ill-typed.

Figure 15 shows the extended reduction relation. In contrast to the old reduction relation (see Section 4) we don't need the function context F any more. As mentioned above, a program now consists only of a single expression.

$$\text{CONG}\ \frac{e_1 \longrightarrow e_2}{E[e_1] \longrightarrow E[e_2]} \qquad \text{FAPP}\ \frac{}{\texttt{(}\lambda x \texttt{ : } t\texttt{.}e\texttt{)(}v\texttt{)} \longrightarrow e[x \mapsto v]}$$

$$\text{FIX}\ \frac{}{\texttt{fix(}\lambda x \texttt{ : } t\texttt{.}e\texttt{)} \longrightarrow e[x \mapsto \texttt{fix(}\lambda x \texttt{ : } t\texttt{.}e\texttt{)}]}$$

$$\text{MATCH}\ \frac{\begin{array}{c}\text{tree}(v) \qquad \neg\,\text{matches}(pat_j, v), j \in \{1, \ldots, i-1\} \\ \text{match}(pat_i, v) = \{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}\end{array}}{\begin{array}{c}\texttt{match}\ v \texttt{ : } t\ \texttt{case}\ pat_1 \texttt{->}\ e_1 \ldots\ \texttt{case}\ pat_n \texttt{->}\ e_n \\ \longrightarrow e_i[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\end{array}}$$

Figure 15: Extended Reduction Relation

The congruence rule CONG can be adopted directly from the reduction relation of the simply typed language (see Figure 4). As for this rule to also apply to the new syntactic constructs we extend the definition of evaluation contexts as follows.

$$\begin{array}{rcl} E & ::= & \bullet \\ & & \mid E\texttt{(}\mathit{expr}\texttt{)} \\ & & \mid \texttt{(}\lambda x\ \texttt{:}\ t\texttt{.}\mathit{expr}\texttt{)(}E\texttt{)} \\ & & \mid c\texttt{(}\mathit{value}, \ldots, \mathit{value}, E, \mathit{expr}, \ldots, \mathit{expr}\texttt{)} \\ & & \mid \texttt{match}\ E\ \mathit{caseexpr}^{+} \\ & & \mid \texttt{fix(}E\texttt{)} \end{array}$$

The rule FAPP covers function applications. It says that whenever a function value $\lambda x\ :\ t.e$ is applied to a value v is can reduce to the body of the function e where the argument x is replaced by the value v. In order for this rule to be well defined we need to extend to definition of substituting values in expressions. Figure 16 shows the new version. The cases for variables, constructor applications and match expressions remain the same. In the case for function applications we now need to also apply the substitution to the function expression in addition to applying it to the argument expression. The cases for `fix` and for functions are new. In the case of `fix` the substitution is propagated into the argument. The case for functions requires more attention. Since the function $\lambda x\ :\ t.e$ binds the variable x, occurrences of x in the body e should not be replaced. Hence, similar to the case for pattern matching expressions, we apply a substitution σ' to e. σ' is defined as σ without the mapping for the variable x.

$$\begin{array}{lcl}
\sigma(x) & = & \begin{cases} \sigma(x) & \text{if } x \in \text{dom}(\sigma) \\ x & \text{otherwise} \end{cases} \\
\sigma(C\texttt{(}e_1, \ldots, e_n\texttt{)}) & = & C\texttt{(}\sigma(e_1), \ldots, \sigma(e_n)\texttt{)} \\
\sigma(e_1\texttt{(}e_2\texttt{)}) & = & \texttt{(}\sigma(e_1)\texttt{)(}\sigma(e_2)\texttt{)} \\
\sigma(\lambda x\ \texttt{:}\ t\texttt{.}e) & = & \lambda x\ \texttt{:}\ t\texttt{.}\sigma(e) \\
 & & \text{where } \sigma' = \sigma \setminus \{x\} \\
\sigma(\texttt{fix(}e\texttt{)}) & = & \texttt{fix(}\sigma(e)\texttt{)} \\
\sigma\left(\begin{array}{l} \texttt{match}\ e\ \texttt{:}\ t \\ \quad \texttt{case}\ \mathit{pat}_1\ \texttt{->}\ e_1 \\ \qquad \vdots \\ \quad \texttt{case}\ \mathit{pat}_n\ \texttt{->}\ e_n \end{array}\right) & = & \sigma\left(\begin{array}{l} \texttt{match}\ \sigma(e)\ \texttt{:}\ t \\ \quad \texttt{case}\ \mathit{pat}_1\ \texttt{->}\ \sigma_1(e_1) \\ \qquad \vdots \\ \quad \texttt{case}\ \mathit{pat}_n\ \texttt{->}\ \sigma_n(e_n) \end{array}\right) \\
 & & \text{where } \sigma_i = \sigma \setminus \text{vars}(\mathit{pat}_i)
\end{array}$$

Figure 16: Applying a Substitution to an Expression

The rule FIX implements the behavior of the fixpoint construct as explained in example 7.2. Whenever `fix` is applied to a function value it reduces to the body of that function thereby replacing the argument with the whole expression.

The rule MATCH differs only slightly from its original counterpart. In the old system all values where tree values. In the extended system the range of values also comprises function values. Since pattern matching is a construct working on tree values only we have to ensure that the value v matched against is indeed a tree value. This is done by the additional premise tree(v).

Example 7.4
To demonstrate the reduction relation we use the `bind` function from the beginning of this Section. `bind` was defined as follows.

```
bind(x : maybe-num,
     f : num -> maybe-num) : maybe-num =
  match x : maybe-num
    case None    -> None
    case Just(y) -> f(y)
```

The intention is to lift functions of the type `number -> maybe-num` to work on values of type `maybe-num`. As an example we lift the predecessor function. The predecessor function is defined as follows.

```
λx : num.
  match x : num
    case Zero    -> None
    case Succ(y) -> Just(y)
```

Let's first look at the application

```
bind(None,
     λx : num.
       match x : num
         case Zero    -> None
         case Succ(y) -> Just(y))
```

Both arguments are values, hence to whole expression reduces to

```
match None : maybe-num
  case None    -> None
  case Just(y) -> (λx : num.
                     match x : num
                       case Zero    -> None
                       case Succ(y) -> Just(y))
                   (y)
```

by FAPP and this again reduces to

```
None
```

by MATCH. Thus, the values `None` is propagated automatically to the result without bothering the lifted function.

As a second example we use an argument different from `None`.

```
bind(Just(Succ(Zero)),
     λx : num.
       match x : num
         case Zero    -> None
         case Succ(y) -> Just(y))
```

This expression evaluates to

```
match Just(Succ(Zero)) : maybe-num
  case None    -> None
  case Just(y) -> (λx : num.
                     match x : num
                       case Zero    -> None
                       case Succ(y) -> Just(y))
                  (y)
```

in one step.

Since this time the second pattern matches the value `Just(Succ(Zero))`. Therefore the rest of the computation is delegated to the lifted function.

```
(λx : num.
   match x : num
     case Zero    -> None
     case Succ(y) -> Just(y))
(Succ(Zero))
```

Evaluating this expression finally yields the value `Just(Zero)`.

7.3 Type System

The goal of the type system is to define the set of well-typed expressions. As in the simply typed language we define an expression to be well typed if evaluating this expression can never reach a state that is neither a value nor can reduce further.

In the simply typed language subtyping showed to be an essential feature for the usability of the system. In the course of extending the language with higher order functions we also extended the range of types by function types. Hence, we need to lift the notion of subtyping to function types. Subtyping on base types is defined as inclusion of languages. From the point of view of typing this means that we can use an expression of a subtype in all contexts where an expression of the supertype is expected. For example, we can safely pass the expression `Zero` with type `(z, (z -> Zero))` to a function that expects an expression of type `num`, since `(z, (z -> Zero))` is a subtype of `num`.

Functions are contravariant in the argument type and covariant in the result type.[24] This means a function type t_1->t_1' is a subtype of a function type t_2->t_2' under two conditions.

- t_1' is a subtype of t_2'.
- t_2 is a subtype of t_1.

This duality arises from the fact that we want to be able to safely use a function f_1 of type t_1->t_1' in any context where a function f_2 of type t_2->t_2' is expected. There are two aspects of using a function. The first aspect is obtaining a result. If we call a function of type t_2->t_2' we expect a result of type t_2'. Since the function f_1 produces a result of type t_1' which is a subtype of t_2', f_1 can be safely used to produce this result. The second aspect of using a function is applying it, i.e. passing an argument to the function. The constraints on types for applying a function are dual to the constraints for using the result. If we are in a context that expects a function of type t_2->t'_2 we know that we can safely pass any argument of type t_2 to that function. Hence, the function f_1 needs to be able to handle arguments of type t_2. On the other hand it is no problem if it even accepts a broader range of values as input. Therefore, the subtype relation on the argument types is reversed. Figure 17 shows the formal definition of subtyping for function types.

$$t_1 \texttt{->}\, t'_1 <: t_2 \texttt{->}\, t'_2 :\Leftrightarrow t_2 <: t_1 \wedge t'_1 <: t'_2$$

Figure 17: Subtyping for Function Types

Figure 18 shows the typing rules for the extended system. T-FABS, T-FAPP and T-FIX correspond to the standard typing rules for the lambda calculus and related systems. A function definition of the form $\lambda x\texttt{:}t_1\texttt{.}e$ is well-typed with the type t_1->t_2 if the body e is well-typed with type t_2 under the extended typing context that maps the argument x to the annotated type t_1. On the other hand, a function application of the form $e_1\texttt{(}e_2\texttt{)}$ is well-typed with type t_2 if we can show that e_1 has the function type t_1->t_2 and the argument e_2 has the matching type t_1. A fixpoint construct of the form `fix(e)` is well-typed with type t if the expression e is well-typed with type t -> t.

The rules T-VAR and T-SUB are adopted unchanged from the simply typed language. The rule T-CAPP did not change either but still needs some explanation. We already discussed that constructor applications are only valid if the arguments are trees. Especially, we do not allow functions as arguments to constructors. This requirement is ensured by the rule T-CAPP by constraining the types of the arguments to be grammars. Since functions have function types this rules out the possibility to use functions as arguments to constructors.

The rule T-MATCH covers the case of pattern matching expressions. In the reduction relation we specified that pattern matching is only possible on tree values. Hence, the task of the rule T-MATCH is to ensure that in any well-typed pattern matching expression the expressions matched against will actually evaluate to a tree value at runtime. This is done by the premise base(t_e) which requires that the annotated type t_e is a base type, i.e. a tree grammar. Apart from this additional premise the rule T-MATCH coincides with the corresponding rule for the simply typed language.

Example 7.5
We show that the function `bind` from example 7.4 is well-typed with type `maybe-num -> (num -> maybe-num) -> maybe-num`. As to better see the correspondence from the syntax to the typing rules we desugar the function definition which contains two arguments into two nested functions each

$$\text{T-SUB}\ \frac{\Gamma \vdash e : t_{sub} \qquad t_{sub} <: t}{\Gamma \vdash e : t} \qquad\qquad \text{T-VAR}\ \frac{(x : t) \in \Gamma}{\Gamma \vdash x : t}$$

$$\text{T-FABS}\ \frac{\Gamma \cup \{x : t_1\} \vdash e : t_2}{\Gamma \vdash \lambda x\ :\ t_1 . e : t_1 \texttt{->} t_2}$$

$$\text{T-FAPP}\ \frac{\Gamma \vdash e_1 : t_1 \texttt{->} t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : t_2}$$

$$\text{T-FIX}\ \frac{\Gamma \vdash e : t \texttt{->} t}{\Gamma \vdash \texttt{fix(}e\texttt{)} : t}$$

$$\text{T-CAPP}\ \frac{\pi(s_t) = C\texttt{(}n_1\texttt{,} \ldots \texttt{,}n_n\texttt{)} \qquad \Gamma \vdash e_i : \texttt{G(}n_i\texttt{,}\pi\texttt{)}}{\Gamma \vdash C\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} :\ \texttt{G(}s_t\texttt{,}\pi\texttt{)}}$$

$$\text{T-MATCH}\ \frac{\begin{array}{c}\text{base}(t_e) \qquad \Gamma \vdash e : t_e \qquad \text{match}(pat_i, t_e) = \Gamma_i \\ \text{exhaustive}(\{pat_1, \ldots, pat_n\}, t_e) \qquad \Gamma_i \cup \Gamma \vdash e_i : t\end{array}}{\Gamma \vdash \texttt{match}\ e\ :\ t_e\ \texttt{case}\ pat_1 \texttt{->}\ e_1 \ldots \texttt{case}\ pat_n \texttt{->}\ e_n : t}$$

Figure 18: Typing Relation for the System with Higher Order Functions

taking only one argument.

```
bind ::=
  λx : maybe-num.
    λf : num -> maybe-num.
      match x : maybe-num
        case None    -> None
        case Just(y) -> f(y)
```

The outermost construct is a function abstraction, hence the rule T-FABS applies. It tells us that in order to show that the whole expression has the type `maybe-num -> (num -> maybe-num) -> maybe-num` under the empty typing context we have to prove that the body of the expression

```
λf : num -> maybe-num.
  match x : maybe-num
    case None    -> None
    case Just(y) -> f(y)
```

has type `(num -> maybe-num) -> maybe-num` under the typing context

$$\Gamma = \{\texttt{x} : \texttt{maybe-num}\}$$

.

The body of the outermost function is again a function definition. By the same arguments we thus have to show that

```
  match x : maybe-num
    case None     -> None
    case Just(y)  -> f(y)
```

has type `maybe-num` under the typing context

$$\Gamma' = \{\texttt{x} : \texttt{maybe-num}, \texttt{f} : \texttt{num -> maybe-num}\}$$

.

Now, the rule T-MATCH applies. `maybe-num` is a base type, hence the check base(`maybe-num`) succeeds. Both pattern `None` and `Just(y)` match the type `maybe-num`. Matching the pattern `None` against the type `maybe-num` yields the empty typing context $\Gamma_1 = \emptyset$. Matching the pattern `Just(y)` against the type `maybe-num` yields the typing context $\Gamma_2 = \{\texttt{y} : \texttt{num}\}$. According to T-MATCH we need to conduct the two subproofs showing $\Gamma' \cup \Gamma_1 \vdash \texttt{None} : \texttt{maybe-num}$ and $\Gamma' \cup \Gamma_2 \vdash \texttt{f(y)} : \texttt{maybe-num}$. We do this by writing down the derivation trees.

$$\dfrac{\dfrac{\pi(\texttt{n}) = \texttt{None}}{\Gamma' \cup \Gamma_1 \vdash \texttt{None} : \texttt{G(n, (n -> None))}}\;\text{T-CAPP} \qquad \texttt{G(n, (n -> None))} <: \texttt{maybe-num}}{\Gamma' \cup \Gamma_1 \vdash \texttt{None} : \texttt{maybe-num}}\;\text{T-SUB}$$

$$\dfrac{\dfrac{(\texttt{f} : \texttt{num -> maybe-num}) \in \Gamma' \cup \Gamma_2}{\Gamma' \cup \Gamma_2 \vdash \texttt{f} : \texttt{num -> maybe-num}}\;\text{T-VAR} \qquad \dfrac{(\texttt{y} : \texttt{num}) \in \Gamma' \cup \Gamma_2}{\Gamma' \cup \Gamma_2 \vdash \texttt{y} : \texttt{num}}\;\text{T-VAR}}{\Gamma' \cup \Gamma_2 \vdash \texttt{f(y)} : \texttt{maybe-num}}\;\text{T-FAPP}$$

7.4 Soundness

The soundness proof for the type system of the simply typed language can be extended in a straight forward way to yield a soundness proof for the extended type system. Since the proof shows nothing new we omit the details and only give an overview of the necessary steps.

The soundness proof for the simply typed language was separated into three parts: the substitution lemma (Lemma 4.2) and the theorems about progress (Theorem 4.1) and preservation (Theorem 4.1). All three proofs were carried out as a structural induction on derivation trees. The main part of the proof hence consisted of a case analysis on the last rule used in the derivation tree. What remains to be done is to extend this case analysis with cases for the new typing rules.

8 Case Study II: Lists, Maps and Folds

In the world of functional programming, algebraic data structures and their corresponding higher order traversal functions are an important tool for writing elegant and concise programs. In this case study we explore how these well-known techniques can be expressed in our simply typed language with higher order functions using the example of lists. We show how to encode lists as tree grammars and demonstrate how to implement the higher order list functions "map" and "fold". While algebraic data types and tree grammars have a great deal in common, there are also some differences. In the last part of this case study we discuss these differences.

We start by giving a grammar for describing lists of numbers.

```
num-list ::= (l, (l -> Nil | Cons(n,l)
                  n -> Zero | Succ(n)))
```

This grammar corresponds to the standard definition of lists in programming languages featuring algebraic data types like Haskell or ML.

Example 8.1
Examples for lists are

```
Nil
```

representing the empty list or

```
Cons(Zero,
     Cons(Succ(Succ(Zero)),
          Nil))
```

representing the list with elements 0 and 2.

Lists are inductive data structures. Hence, most functions on lists have the recursive form

```
f(l : num-list) : res
  match l : num-list
    case Nil         -> base-case
    case Cons(n,l') -> some-fun(n,f(l'))
```

Example 8.2
The following function computes the length of a list.

```
length(l : num-list) : num
  match l : num-list
    case Nil         -> 0
    case Cons(n,l') -> Succ(length(l'))
```

Example 8.3
The following function increments each element of the list by one.

```
map-inc(l : num-list) : num-list
  match l : num-list
    case Nil         -> Nil
    case Cons(n,l') -> Cons(Succ(n),map-inc(l'))
```

The `length` function from example 8.2 is an instance of a more general recursion scheme called a *fold*. A fold aggregates the elements of a list into one value.

The `map-inc` function from example 8.3 is an instance of a more general recursion scheme called a *map*. A map constructs a new list out of a given list by applying a function to each element.

In the following we show how to abstract over the common behavior using higher order functions.

Folds

Folds on lists of numbers aggregate the elements of the lists into one value. Since we are in a mono-typed setting, we first restrict our discussion to folds where the result of the aggregation is a number.

We need to consider two cases.

- The first case applies when the input list is `Nil`. This corresponds to the base case of the recursion. Since we cannot know in general which value to return in case of the empty list, we make this into an argument of the fold function.
- The second case applies when the input list has the form `Cons(n,l')`. Recursively applying the fold function to the sublist `l'` yields a number as result. Hence, we need a function that combines the number `n` with the result of the recursive call. Since the way of combining these two numbers will be different in different instances of the fold function, we also make this function into an argument.

Thus, the fold function has the type

```
(num -> num -> num) -> num -> num-list -> num
```

The implementation of the fold function is straight forward.

```
fold(f : num -> num -> num, base : num, l : num-list) : num =
  match l : num-list
    case Nil         -> base
    case Cons(n,l') -> f(n,fold(f,base,l'))
```

In case of `l = Nil` we return `base`. Otherwise we recursively fold the sublist and use the function `f` to combine the result with the element `n`.

We can encode the function `length` from example 8.2 as an instance of a fold as follows.

```
length(l : num-list) : num =
  fold(λx : num.
         λy : num. Succ(y),
       Zero,
       l)
```

The length of the empty list is 0, hence we supply the value `Zero` as base case to the fold. Since the length of the list is independent from the concrete values in the list, the argument function ignores its first argument `x` and increments its second argument `y` by one.

Another example for a concrete fold is computing the sum of the elements of the list.

```
sum ::= λl : num-list.
          fold(plus,Zero,l)
```

Maps

Analogous to the folds we can define a general map function on lists of numbers as follows.

```
map(f : num -> num, l : num-list) : num-list =
  match l : num-list
    case Nil         -> Nil
    case Cons(n,l')  -> Cons(f(n),map(f,l'))
```

In case of the empty list we return the empty list. Otherwise we recursively map the function `f` over the sublist `l'` and add the result of `f(n)` at the beginning.

With the help of this map function we can encode the function `map-inc` from example 8.3 as an instance of a map as follows.

```
map-inc(l : num-list) : num-list =
  map(λx : num. Succ(x),
      l)
```

Discussion

The implementation of functions like `map` and `fold` shows how the well-known techniques on algebraic datatypes carry over to tree grammar types in a straight forward way.

However, the simply typed language with higher order functions is more expressive than the mono-typed counterpart of functional languages with algebraic datatypes. The reason is that our system uses structural subtyping on tree grammars. Furthermore, we lift the notion of subtyping to function types. This leads to quite powerful abstraction mechanisms.

Let's for example assume the constant function

```
c(x : num) : zero =
  Zero
```

with

```
zero ::= G(z, (z -> Zero))
```

that maps all numbers to `Zero`.

Thanks to subtyping on functions we can lift this function to a function from lists of numbers to lists of numbers as follows.

```
c-lifted(l : num-list) : num-list =
  map(c,l)
```

In a system without subtyping on functions we would have to change the result type of `c` to the more imprecise type `num` in order for the function `c-lifted` to be well-typed. But this implies that we cannot use the function `c` any more in a context where a result of type `zero` is expected.

On the other hand, let's assume a function `f` of type `arith-expr -> num`, where `arith-expr` is defined as follows.

```
arith-expr ::= G(ae, (ae -> Zero | Succ(ae) | Plus(ae,ae)))
```

The type `num` is a subtype of `arith-expr`. Hence, the function type `arith-expr -> num` is a subtype of the function type `num -> num`. This implies that we can lift the function `f` to a function from lists of numbers to lists of numbers in the following way.

```
f-lifted(l : num-list) : num-list =
  map(f,l)
```

While in the first case we could change the result type of the function `c` to the more imprecise type `num`, in this case it is not possible to change the argument type of `f` to `num`: the function `f` is not well-typed with type `num -> num`.

While higher order functions are a very powerful abstraction mechanism they don't solve the problem of how to abstract over types. For example, our map function from above works only on lists of numbers. If we want to map a function over a lists with different element types, say boolean values, we have to implement another instance of the map function, namely

```
map(f : bool -> bool, l : bool-list) : bool-list = ...
```

The same holds if we want to lift a function of type `num -> bool` or `bool -> num`. All instances share the same code, the only difference are the type annotations.

In the next Section we introduce polymorphic types to our language. This allows us to implement the map function with the polymorphic type `(X -> Y) -> list(X) -> list(Y)`. Note, that with polymorphic types we can even express the map function as an instance of the fold function as follows.

```
map(f : X -> Y, l -> list(X)) : list(Y) =
  fold(λx : X.
         λy : list(Y).
           Cons(f(x),y),
       Nil,
       l)
```

9 Second Extension: Parametric Types

Both case studies (see Section 6 and Section 8) clearly showed that the simply typed language lacks an important abstraction mechanism, namely the possibility to abstract over types. So far, the simply typed language features two different kinds of abstraction mechanisms. The first abstraction mechanism are functions which abstract over values. This enables us to use the same piece of code with different values. The second abstraction mechanism are higher order functions which abstract over behavior. This means we can factor out pieces of code that are similar except for context specific behavior at certain points. But there also exist situations where we have multiple pieces of code that vary only in their type annotations. In the previous case study we showed how to encode lists and higher order list functions in the simply typed language. In this course we implemented the map function `map-num` as follows.

```
map-num(f : num -> num, l : num-list) : num-list
  match l : num-list
    case Nil        -> Nil
    case Cons(n,l') -> Cons(f(n),map-num(f,l'))
```

If we want to apply a function with a different result type to all elements of the list, we need to write a new instance of the map function. For example, a map function that lifts a function of type `num -> bool` to lists looks as follows.

```
map-bool(f : num -> bool, l : num-list) : bool-list
  match l : num-list
    case Nil        -> Nil
    case Cons(n,l') -> Cons(f(n),map-bool(f,l'))
```

Note that the function `map-bool` *exactly* corresponds to the function `map-num` except for the type annotations.

Many programming languages offer parametric polymorphism [24] as a solution to abstract over type independent code. Parametric polymorphism essentially allows to use type variables instead of types with the following semantics: the code is valid for any instantiation of the type variables with concrete types. Hence, it is possible to abstract over types. For example, the map function in Haskell has the type `(a -> b) -> [a] -> [b]`. This means: for all types `a` and `b`, given a function of type `a -> b` and a list of type `a` the map function yields a list of type `b`.

In the reminder of this Section we discuss how to incorporate the notion of type parametricity into our calculus for metaprogramming. Therefor, we first need to clarify what parametricity means in the context of regular tree grammars. After that, we extend the simply typed language with System F [24] like polymorphism. This includes extensions to the syntax and semantics of the language and additional typing rules. And we also prove the soundness of the resulting system.

9.1 Parametric Tree Grammars

In Section 3 we specified the language a grammar generates by defining how trees can be derived by the grammar. In a nutshell, a tree is derived by successively replacing nonterminals by right hand sides of productions starting with the start symbol. As a small example we look at the grammar for numbers.

```
num := G(n, (n -> Zero | Succ(n)))
```

We start a derivation with the start symbol `n`. The productions tell us that we can replace `n` by `Zero` or by `Succ(n)`. If we choose to replace `n` by `Zero` we are done. There are no nonterminals left and hence `Zero` is value of the grammar `num`. But we can also choose to replace `n` by `Succ(n)`. `Succ(n)` still contains the nonterminal `n`. We continue this process of replacing `n` by one right hand side of the productions until we finally reach a value.

But what is the meaning of a grammar that contains type variables?

Example 9.1
As an example we consider a grammar for parametric lists.

```
list := G(l, (l -> Nil | Cons(X,l)))
```

Thereby `X` is a language variable that abstracts over the concrete types of the list elements. According to the rules of derivation we can for example derive the trees `Nil` or `Const(X,Cons(X,Nil))`. These trees are not yet values, because they still contain the language variable `X`. But for each *instantiation* of the language variable `X` with a concrete grammar `G` we can continue the derivation process by replacing `X` with the start symbol of that grammar `G`.

Hence, parametric grammars can be seen as *generators* for grammars or functions from grammars to grammars: Given an input grammar, a parametric grammar generates a new grammar by instantiating the respective language variable with the input grammar.

Figure 19 shows the syntax extension of grammars with language variables. Differences are highlighted with boxes. We assume the set of language variables to be distinct from the set of constructor names.

$$
\begin{array}{lcl}
\mathit{base} & ::= & \texttt{G}(n\texttt{, }(\mathit{prod}, \ldots, \mathit{prod})\texttt{)} \\
\mathit{prod} & ::= & n \texttt{ -> } \mathit{rhs} \mid \ldots \mid \mathit{rhs} \\
\mathit{rhs} & ::= & c(\boxed{N}, \ldots, \boxed{N}) \\
N & ::= & n \mid X \\
X & ::= & \boxed{\text{type variable names}}
\end{array}
$$

Figure 19: Syntax Extensions for Grammars

Figure 20 shows the formal definition of instantiating a language variables with a grammar. We purposely reuse the syntax for applying a substitution to an expression (see Section 2.2.3), because the concept of replacing a variable with a concrete value is similar to replacing a language variable with a concrete grammar, only at a different abstraction level. The instantiation works as follows: we want to replace the language variable `X` within the grammar $\texttt{G(}s\texttt{,}\pi\texttt{)}$ by the grammar $\texttt{G(}s'\texttt{,}\pi'\texttt{)}$. To this end we need to replace the variable `X` by the start symbol s' of the second grammar. In addition we need to add the productions π' to the productions π. If necessary, we rename the nonterminals in the grammar $\texttt{G(}s'\texttt{,}\pi'\texttt{)}$ appropriately before applying the substitution as to avoid name clashes.

$$\texttt{G(}s\texttt{,}\pi\texttt{)}[X \mapsto \texttt{G(}s'\texttt{,}\pi'\texttt{)}] \quad = \quad \texttt{G(}s\texttt{,}\pi[X \mapsto s'] \cup \pi'\texttt{)}$$

Figure 20: Instantiation of Language Variables

Example 9.2
We consider the case of instantiating the language variable `X` in the list grammar of Example 9.1 with the grammar `num`. According to the definition of instantiation we need to replace the language variable `X` with the start symbol `n` of the grammar `num` in the productions of the grammar `list`.

The resulting grammar arises from adding the productions of the grammar `num` to the modified productions of the grammar `list`. Hence, the result is

```
G(l, (l -> Nil | Cons(n,l),
      n -> Zero | Succ(n)))
```

which corresponds exactly to our definition of number lists in Section 8.

There is one subtle problem that we need to be aware of. The nonterminals of a grammar are implicitly bound within the definition of the grammar. This also means that we can rename them at will as to avoid name clashes. So far, we have not yet discussed the scope of language variables. But in the case where both the grammar we substitute in and the grammar we substitute with contain the same language variable `X`, the process of substitution establishes a connection between the two language variables from different grammars. In the course of extending the simply typed language with language constructs for type abstraction we introduce binding mechanisms for type variables. It turns out that we never need to substitute grammars containing free language variables. Hence, the definition for instantiation as is produces the expected results.

9.2 Syntactic Extensions

So far, we have introduced the notion of language variables ranging over grammar types. But grammar types are not the only kind of types in our language: there are also function types. The example of the polymorphic identity function

```
id(x : X) : X = x
```

shows that it is overly restrictive to constrain type variables to range over grammar types. Hence, we like to extend type language as follows.

$$t \quad ::= \quad \mathit{base} \mid t \texttt{->} t \mid \boxed{X}$$

This allows us to use a type variable X in place of any type, including function types.

9.2.1 Type Abstractions

With the extension of the type language as discussed above we are able to write down types that contain type variables. Pushing this extension into the syntax of our language implies that we can use type variables inside of type annotations. But this is not enough to yield parametric polymorphism. While we have a mechanism to use type variables we still lack a corresponding binding mechanism. In the context of values this compares to offering syntax for variables without including a language construct like functions that can bind them.

Example 9.3
We consider an implementation of the polymorphic map function.

```
λf : A -> B.
  λf : list(A).
    ...
```

In this context it is clear that both type variable `A` should belong to the same scope. The type of the list elements needs to correspond to the argument type of the function `f`. Hence, the scope of the type variables `A` and `B` consists of the whole expression.

As second example we consider the following function.

```
map-const :=
λf : X -> num.
  λl : list(Y).
    map(f)(l)
```

This function `map-const` takes a function as argument that turns any input into a number. One example for such a function is the constant function `λ.x : X. Zero`. The second argument of the function `map-const` is a list with element type `Y`. In this case the type of `map-const` should be something like `(∀X.X -> num) -> list(Y) -> list(num)`. Hence, the scope of the type variable `X` does not extend into the function definition.

The last example showed that we need a language construct for binding type variables. There exists two well-known approaches that we discuss in the following.

Let-Polymorphism Systems with let-polymorphism have top-level language constructs which constrain the scope of type variables. One prominent example for a language featuring let-polymorphism is Haskell. Haskell has top-level definitions of the form

```
f x = ...
```

In the signature for a top-level definition it is possible to use type variables. For example we can write the polymorphic identity function as follows.

```
id :: a -> a
id x = x
```

Top-level definition implicitly bind type variables. Hence, the signature essentially is a shorthand notation for `f :: ∀a. a -> a`.

Let-Polymorphism hence allows to abstract over types and use the same piece of code with different types in different contexts. For example we can apply the identity function to a number as in

```
id 10
```

which implicitly instantiates the type variable with the concrete type `Int`. But we can also apply the function to a boolean values as in

```
id True
```

A more verbose notation would write the applications as

```
id<Int> 10
```

and

```
id<Bool> True
```

respectively, making the instantiation with the concrete type `Int` explicit. In Haskell this in not necessary, because the type instantiations are done automatically using type inference.

However, only top-level definitions can bind type variables. This means that the most fine grained scope for type variables corresponds to these top-level definitions. Especially, we cannot express a function like the function `map-const` from example 9.3 that requires a more fine grained scoping mechanism. Since this restriction turned out to be a serious limitation in practice, there is ongoing research about extending Haskell with more powerful type abstraction mechanisms [23].

Universal Types Due to the limitations of let-polymorphism, we choose to extend our language with a more powerful means of parametric polymorphism, called universal types [24]. The general idea is to make type abstractions and type applications part of the expression language.

In the same way as abstraction over values is encoded by functions from values to values we can abstract over types by the means of functions from types to values, called type abstractions. We write a type abstraction with the type parameter `X` and the body e as Λ`X`.e. The upper case lambda Λ syntactically distinguishes normal functions from type abstractions. Correspondingly, we extend the language of types with generator types, called universal types, of the form $\forall$`X`.t. The dual to type abstraction is providing a type abstraction with a concrete type, hence type application. We write type applications as e<t>.

Example 9.4
The polymorphic identity function can be implemented as follows.

```
id :=
ΛX. λx : X. x
```

We can use this function by instantiating the type variable with a concrete type.

```
id<num>(Zero)
```

First, we apply the function `id` to the type `num` yielding the identity function on numbers. This resulting function can then be applied to the number `Zero`.

Figure 21 summarizes the syntax of the extended language. The new language constructs are highlighted with boxes.

9.3 Values and Semantics

With the syntactic extension of the language by type abstractions and type applications we introduce a new class of values, namely type abstractions. As discussed in Section 7 we don't allow to use function values as arguments to constructors, since the result cannot be described by a tree grammar.

We need to discuss how to deal with type abstractions as arguments to constructors.

Example 9.5
Suppose we are in a context where `x` has the type

```
t := ∀X. G(s, (s -> Zero | Succ(X))
```

Now, suppose we allowed expressions of the form `C(x)`. With the hypothetical notation using the forall quantifier inside of grammars we could assign the type

```
t' := G(a, (a -> C(b),
            b -> ∀X. G(s, (s -> Zero | Succ(X)))))
```

to the expression `C(x)`.

If we deconstruct this expression by pattern matching as in

$$\begin{array}{lcl}
\mathit{expr} & ::= & x \\
 & & \mid \lambda x \texttt{ : } t\texttt{.}\mathit{expr} \\
 & & \mid \mathit{expr}\texttt{(}\mathit{expr}\texttt{)} \\
 & & \mid \texttt{fix(}\mathit{expr}\texttt{)} \\
 & & \mid \boxed{\Lambda X\texttt{.}\mathit{expr}} \\
 & & \mid \boxed{\mathit{expr}\texttt{<}t\texttt{>}} \\
 & & \mid c\texttt{(}\mathit{expr}\texttt{,}\ldots\texttt{,}\mathit{expr}\texttt{)} \\
 & & \mid \texttt{match } \mathit{expr} \texttt{ : } t \ \mathit{caseexpr}^{+} \\
\mathit{caseexp} & ::= & \texttt{case } \mathit{pat} \texttt{ -> } \mathit{expr} \\
\mathit{pat} & ::= & x \mid c\texttt{(}\mathit{pat}\texttt{,}\ldots\texttt{,}\mathit{pat}\texttt{)} \\
t & ::= & \mathit{base} \mid t\texttt{->}t \mid X \mid \boxed{\forall X\texttt{.}t} \\
\mathit{base} & ::= & \texttt{G(}n\texttt{, (}\mathit{prod}\texttt{,}\ldots\texttt{,}\mathit{prod}\texttt{))} \\
\mathit{prod} & ::= & n \texttt{ -> } \mathit{rhs} \texttt{ | } \ldots \texttt{|} \mathit{rhs} \\
\mathit{rhs} & ::= & c\texttt{(}N\texttt{,}\ldots\texttt{,}N\texttt{)} \\
N & ::= & n \mid X \\
x & ::= & \text{variable names} \\
c & ::= & \text{constructor names} \\
n & ::= & \text{nonterminal names} \\
X & ::= & \text{type variable names}
\end{array}$$

Figure 21: Syntax of the Language with Parametric Types

```
  match C(x) : t'
    case C(y) -> ...
```

we forward the parametricity of `x` to the right hand sides of the case expression. This means we could instantiate the pattern variable `y` with different types at different places. In contrast to that, an expression of the form

```
  ΛX. C(x<X>)
```

forces the instantiation of the subexpression with only one concrete type.

The additional flexibility of allowing type abstractions as arguments to constructors raises some problems.

- We need to extend regular tree grammars in a way that supports quantification inside of productions.
- Since we don't allow functions as arguments to constructors, we need to distinguish between type abstractions with a function body and type

abstractions with a tree body.

We believe that this additional flexibility does not increase the expressivity of the language for the following reason. The only possible value with type

```
∀X.G(s, (s -> Zero | Succ(X)))
```

is the value `ΛX.Zero`, because the second production `s -> Succ(X)` cannot be used to construct values as long as we don't know the concrete instantiation for the type variable `X`. Hence, values of type `∀X.t` where `t` is a grammar cannot exploit the parametricity but always need to treat the type variable as a "black box".

Hence, we can define the values of the language as shown in figure 22. There are three classes of values: tree values, functions and type abstractions. As can be seen from the definition of tree values, arguments to constructors have to be tree values.

$$\begin{array}{lcl} \mathit{val} & ::= & \mathit{treeval} \mid \Lambda X\,.\,\mathit{expr} \mid \lambda x\!:\!t\,.\,\mathit{expr} \\ \mathit{treeval} & ::= & c\texttt{(}\mathit{treeval}\texttt{,}\ldots\texttt{,}\mathit{treeval}\texttt{)} \end{array}$$

Figure 22: Values of the Language with Parametric Types

With the help of the definition of values it is now straight forward to formalize the operational semantics. The reduction rules are shown in Figure 24. The only new rule is the rule TAPP. It says that a type application of the form $\texttt{(}\Lambda X\,.\,e\texttt{)<}t\texttt{>}$ reduces to the body e where the type variable X is replaced by the concrete type t.

As for the rule CONG to also cover the case of type applications we need to extend the definition of the evaluation contexts by the following clause.

$$E \;\; ::= \;\; \ldots \mid E\texttt{<t>}$$

Furthermore, we need to lift the notion of substituting a type inside a grammar to expressions. Figure 9.3 shows the definition. The interesting parts are the cases for type application and type abstraction. In the case of type abstractions we only apply the substitution to the body of the type abstraction if the bound variable is different from the variable to be replaced. If we look at the reduction relation in figure 24, the only place where a substitution of types in initiated is in the rule TAPP for type applications. When specifying the type system, we will see that in any well-typed type application the supplied type does not contain free type variables. Hence, it is not possible that type variables are accidentally captured in the process of substitution.

We illustrate the extension of the language by some examples.

$$
\begin{array}{lcl}
x[X \mapsto t] & = & x \\
(\lambda x{:}t_1.e)[X \mapsto t] & = & \lambda x{:}t_1[X \mapsto t].e[X \mapsto t] \\
(e_1(e_2))[X \mapsto t] & = & e_1[X \mapsto t](e_2[X \mapsto t]) \\
\texttt{fix}(e)[X \mapsto t] & = & \texttt{fix}(e[X \mapsto t]) \\
(\Lambda Y.e)[X \mapsto t] & = & \begin{cases} \Lambda Y.e & \text{if } X = Y \\ \Lambda Y.e[X \mapsto t] & \text{otherwise} \end{cases} \\
(e\texttt{<}t_1\texttt{>})[X \mapsto t] & = & e[X \mapsto t]\texttt{<}t_1[X \mapsto t]\texttt{>} \\
C(e_1,\ldots,e_n)[X \mapsto t] & = & C(e_1[X \mapsto t],\ldots,e_n[X \mapsto t]) \\
(\texttt{match } e_m{:}t_m \texttt{ case } p_i \texttt{->} e_i)[X \mapsto t] & = & \texttt{match } e_m[X \mapsto t]{:}t_m[X \mapsto t] \\
 & & \quad \texttt{case } p_i \texttt{->} e_i[X \mapsto t]
\end{array}
$$

Figure 23: Substitution of Types

Example 9.6
We start with a simple example using the following parametric type for pairs.

```
pair(X,Y) := G(p, (p -> Pair(X,Y)))
```

As before, the definition `pair(X,Y) := ...` is only syntactic sugar to give names to pieces of code in order to not have to inline the definition all the time. Formally, these kind of definitions are not part of the language.

With this definition we can write polymorphic functions for accessing the first and second element of the pair respectively.

```
fst := ΛX.ΛY.
          λp : pair(X,Y).
            match p : pair(X,Y)
              case Pair(x,y) -> x

snd := ΛX.ΛY.
          λp : pair(X,Y).
            match p : pair(X,Y)
              case Pair(x,y) -> y
```

The function `fst` and `snd` first abstract over the types `X` and `Y`.

To use these function we have to supply concrete types. For example the expression

```
fst<num><bool>(Pair(Zero,True))
```

instantiates the type abstractions in `fst` with the concrete types `num` and `bool` yielding a function of type `pair(num,bool) -> num`. This resulting function is then applied to the value `Pair(Zero,True)` yielding the final result `Zero`.

$$\text{CONG}\ \frac{e_1 \longrightarrow e_2}{E[e_1] \longrightarrow E[e_2]}$$

$$\text{TAPP}\ \frac{}{(\Lambda X\,.\,e)\texttt{<}t\texttt{>} \longrightarrow e[X \mapsto t]}$$

$$\text{FAPP}\ \frac{}{(\lambda x\ :\ t.e)(v) \longrightarrow e[x \mapsto v]}$$

$$\text{FIX}\ \frac{}{\texttt{fix}(\lambda x\ :\ t.e) \longrightarrow e[x \mapsto \texttt{fix}(\lambda x\ :\ t.e)]}$$

$$\text{MATCH}\ \frac{\begin{array}{c}\text{tree}(v) \qquad \neg\,\text{matches}(pat_j, v), j \in \{1, \ldots, i-1\} \\ \text{match}(pat_i, v) = \{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}\end{array}}{\begin{array}{c}\textbf{match}\ v\ :\ t\ \textbf{case}\ pat_1 \texttt{->}\ e_1 \ldots \textbf{case}\ pat_n \texttt{->}\ e_n \\ \longrightarrow e_i[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\end{array}}$$

Figure 24: Semantics of the Language with Parametric Types

The next example shows the interplay between type abstractions and recursive functions.

Example 9.7
We can define the type for parametric lists as follows.

```
list(X) := G(l, (l -> Nil | Cons(X,l)))
```

Now, there are two different possibilities for implementing the the polymorphic map function mentioned at the beginning of this Section.

The first possibility is to place the type abstractions outside the fixpoint construct.

```
ΛA. ΛB.
  fix(λmap : (A -> B) -> list(A) -> list(B).
        λf : A -> B.
          λl : list(A).
            match l : list(A)
              case Nil         -> Nil
              case Cons(n,l’) -> Cons(f(n),map(f)(l’)))
```

The second possibility is to push the type abstractions inside the fixpoint construct.

```
fix(λmap : ∀A.∀B.(A -> B) -> list(A) -> list(B).
      ΛA.ΛB.
        λf : A -> B.
          λl : list(A).
            match l : list(A)
              case Nil -> Nil
              case Cons(n,l')
                -> Cons(f(n),map<A><B>(f)(l')))
```

While in the first case we compute the fixpoint of a monomorphic function, in this case the result is the fixpoint of a polymorphic function. This means we have to explicitly instantiate the function with type arguments for the recursive call. In this case we just forward the type arguments `A` and `B`. But this is not necessary in general. Instead we could use different type arguments for the recursive call yielding polymorphic recursion.

9.4 Parametric Type System

The goal of the type system is to reject invalid programs, i.e. programs that lead to an error when executed. With the introduction of type abstractions and type applications to the language there are new sources of errors which the type system needs to deal with.

Example 9.8
We get back to our previous example of pairs (see Example 9.6). There, we defined the function accessing the first element of a pair as follows.

```
fst := ΛX.ΛY.
          λp : pair(X,Y).
            match p : pair(X,Y)
              case Pair(x,y) -> x
```

Suppose we try to instantiate the type variable `X` with a function type.

```
fst<num -> num>
```

This means that we would need to substitute `X` with `num -> num` within the grammar

```
pair(X,Y) := G(p, (p -> Pair(X,Y)))
```

But this is invalid, because the substitution is only defined on grammars and not on arbitrary types. Hence, the rule TAPP does not apply and the expression `fst<num -> num>` cannot reduce further. Since the expressions is also not a value, the whole program should be rejected as ill-typed by the type system.

The last exampled showed that there exist type abstractions that are only defined for a certain set of instantiations and not for instantiations with arbitrary types.

A similar problem can be found in the context of functions and their arguments. Functions are usually only defined for certain input values. For example, the function

```
pred(x : positive-num) : num =
  match x : positive-num
    case Succ(y) -> y
```

is only defined for arguments of type `positive-num`. This restriction of valid input values is made explicit by the type annotation `x : positive-num` so that the type system can check the corresponding function applications.

This suggest that we need a similar annotation mechanism for type abstractions. Classification of types is usually called *kinding* [24]. In our case, the kind system needs two different kinds separating grammar types from the rest. We call them `base` and `any`. With kind annotations we can rewrite the function `fst` from example 9.6 as follows.

```
fst := ΛX : base.ΛY : base.
           λp : pair(X,Y).
             match p : pair(X,Y)
               case Pair(x,y) -> x
```

With this additional information the type checker is able to reject the expression

```
fst<num -> num>
```

as ill-typed since the type `num -> num` is not of kind base, i.e. it is not a grammar.

To incorporate kind annotation into the language we have to extend the syntax as follows.

$$\begin{array}{lll} expr & ::= & \dots \mid \Lambda X \boxed{:k}.\, expr \mid \dots \\ t & ::= & \dots \mid \forall X \boxed{:k}.t \\ k & ::= & \texttt{base} \mid \texttt{any} \end{array}$$

With the help of the kind annotations we can phrase the typing rules for type application and type abstraction as follows.

$$\text{T-TABS}\ \frac{\Gamma, T \cup \{X : k\} \vdash e : t}{\Gamma, T \vdash \Lambda X{:}k.e : \forall X{:}k.t}$$

$$\text{T-TAPP}\ \frac{\Gamma, T \vdash e : \forall X{:}k.t_1 \qquad T \vdash \text{kind}(t) = k}{\Gamma, T \vdash e\texttt{<}t\texttt{>} : t_1[X \mapsto t]}$$

In addition to the typing context Γ we use a kinding context T that maps type variables to kinds. The rule T-TABS covers the case of type abstractions. A type abstraction of the form $\Lambda X{:}k.e$ is well typed with type $\forall X{:}k.t$ if the body e is well typed with type t in using a kinding context that maps the type variable X to the kind k. The typing rule T-TAPP accordingly checks that the argument of a type application has the correct kind. Since the type t can contain free type variables, we need the kinding context T to determine the kind of t.

The definition of kinding of types is shown in figure 25. The kinding rules resemble closely the typing rules of the simply typed language. The difference is that the kind system is much more coarse-grained than the type system of the simply typed language. We have only two kinds `base` and `any`. The main goal of the kind system is to check whether all type variables are defined and whether a grammar only uses type variables of kind `base`. Note that the rule K-SUB establishes a subkind relation between the kinds `base` and `any`.

$$\text{K-BASE}\ \frac{\forall X \in \text{typevars}(\pi) : (X : \texttt{base}) \in T}{T \vdash \text{kind}(\texttt{G(}s\texttt{,}\pi\texttt{)}) = \texttt{base}}$$

$$\text{K-VAR}\ \frac{(X : k) \in T}{T \vdash \text{kind}(X) = k} \qquad \text{K-SUB}\ \frac{T \vdash \text{kind}(t) = \texttt{base}}{T \vdash \text{kind}(t) = \texttt{any}}$$

$$\text{K-FUN}\ \frac{T \vdash \text{kind}(t_1) = \texttt{any} \qquad T \vdash \text{kind}(t_2) = \texttt{any}}{T \vdash \text{kind}(t_1 \texttt{->} t_2) = \texttt{any}}$$

$$\text{K-ALL}\ \frac{T \cup \{X : k\} \vdash \text{kind}(t) = k}{T \vdash \text{kind}(\forall X{:}k.t) = \texttt{any}}$$

Figure 25: Kind System

With the typing rules T-TABS and T-TAPP it is now possible to tell that the application

```
fst<num>
```

is well-typed while rejecting the application

```
fst<num -> num>
```

as ill typed. But we still need typing rules that enforce the kind annotation `base` in the function definition of `fst`. Otherwise we could simply annotate the type variable with the kind `any` yielding an invalid program.

There are three places in the type checking process where we need to ensure kinding constraints.

Type Annotations in Functions We have to ensure that type annotations in function definitions are well-kinded. This means especially that there are no free type variables and that type variables are used according to their kinds.

Example 9.9
The type annotation `pair(X,Y)` in the function `fst` from Example 9.6 uses the type variables `X` and `Y` inside of a grammar. Therefore, the type can only be well-kinded if the type variables both have the kind `base`. This forces us to use the kind annotation `base` in the definition of `fst`.

This leads to the following typing rule for function definitions.

$$\text{T-FABS}\ \frac{T \vdash \text{kind}(t_1) = k \qquad \Gamma \cup \{x : t_1\}, T \vdash e : t_2}{\Gamma, T \vdash \lambda x\ :\ t_1 . e : t_1 \texttt{->} t_2}$$

Pattern Matching A pattern matching expression also uses a type annotation. For the same reason as for function definitions this type annotation should be well-kinded.

But in this case it is not enough that the type annotation is well-kinded with an arbitrary kind. Since pattern matching works only on trees we need to ensure that the expression matched against has a grammar type. This means we have to require the kind `base` for the type annotation.

The difficult part in typing pattern matching expressions is to compute appropriate typing contexts for checking the right hand side of the case expressions. Since we require that the type annotation is a grammar, the only difference to pattern matches in the simply typed language is the existence of type variables inside of the grammar used as type annotation.

The algorithm for computing the typing contexts for pattern variables can be generalized to grammars with type variables in a straight forward way. The only problem is that the algorithm needs to check for the emptiness of types at certain points (see the discussion in Section 4.1). The type variables inside the grammar abstract over types. All we know is that the type variables will be instantiated with a grammar. But we cannot know if the grammar describes the empty type or not.

Example 9.10
We look at the grammar for pairs

```
pair(X,Y) := G(s, (s -> Pair(X,Y)))
```

and the pattern

```
Pair(x,y)
```

If we treat the type variables `X` and `Y` as non-empty types, we get the typing context $\Gamma_1 = \{\texttt{x} : \texttt{X}, \texttt{y} : \texttt{Y}\}$ according to the algorithm from Section 4.1.

On the other hand, if we treat the type variables as empty types, we get the typing context $\Gamma_2 = \{\mathtt{x} : \emptyset, \mathtt{y} : \emptyset\}$.

It is important to note that each expression, that is well typed under any typing context Γ with $\mathrm{dom}(\Gamma) = \{\mathtt{x}, \mathtt{y}\}$ is also well type under the typing context Γ_2 but not vice versa. This implies that for any instantiation of `X` with t_x and `Y` with t_y, if an expression is well-typed under the typing context $\Gamma = \{\mathtt{x} : t_x, \mathtt{y} : t_y\}$ it is also well typed under the typing context Γ_2. On the other hand, if an expression is well-typed under the typing context Γ_2 it is not necessarily well-typed under the typing context Γ.

The last example has shown that we need to treat type variables as non-empty types when computing the typing context for pattern variables. Otherwise we don't get a sound type system.

Figure 26 shows the adaption of the algorithm from Section 4.1 for our extended setting. Thereby we define type variables to be non-empty. Furthermore, the function type is defined as follows.

$$\begin{aligned} \mathrm{type}(X, \pi) &= X \\ \mathrm{type}(n, \pi) &= \mathtt{G(}n\mathtt{,}\pi\mathtt{)} \end{aligned}$$

$$\mathrm{contexts}'(x, t) = \begin{cases} \{\emptyset\} & \text{if } \mathrm{empty}(t) \\ \{\{x : t\}\} & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathrm{contexts}'(C\mathtt{(}p_1\mathtt{,}\dots\mathtt{,}p_n\mathtt{)}, G\mathtt{(}s\mathtt{,}\pi\mathtt{)}) = {} & \bigcup G'_j \\ & \text{where} \\ & \quad C\mathtt{(}N_{1_j}\mathtt{,}\dots\mathtt{,}N_{n_j}\mathtt{)} \in \pi(s) \\ & \quad G_{i_j} = \mathrm{contexts}'(p_i, \mathrm{type}(N_{i_j}, \pi)) \\ & \quad G_j = G_{1_j} \times_{\mathrm{uni}} \cdots \times_{\mathrm{uni}} G_{n_j} \\ & \quad G'_j = \text{filter-nonempty}(G_j) \end{aligned}$$

Figure 26: Computation of Typing Contexts for Pattern Variables

In order to typecheck pattern matching expression we also have to ensure that the pattern are exhaustive with respect to the annotated type. The formulation of the exhaustiveness check for the simply typed language makes use of the subtyping relation on grammars. However, we have not yet discussed what subtyping means in the context of parametric grammars. We defer this discussion to section 9.5.

Assuming a subtyping relation on parametric tree grammars, we can define the typing rule for pattern matching expressions as follows.

$$\text{T-MATCH}\ \dfrac{\begin{array}{c} T \vdash \text{kind}(t_e) = \texttt{base} \qquad \Gamma, T \vdash e : t_e \qquad \text{context}(pat_i, t_e) = \Gamma_i \\ \text{exhaustive}(\{pat_1, \dots, pat_n\}, t_e) \qquad \Gamma_i \cup \Gamma, T \vdash e_i : t \end{array}}{\Gamma, T \vdash \texttt{match}\ e\ \texttt{:}\ t_e\ \texttt{case}\ pat_1\texttt{->}\ e_1 \dots \texttt{case}\ pat_n\texttt{->}\ e_n : t}$$

Constructor Application When typing constructor applications we have to make sure that all arguments have a grammar type. Hence, the typing rule for constructor applications looks as follows.

$$\text{T-CAPP}\ \dfrac{\begin{array}{c} \pi(s_t) = C\texttt{(}N_1\texttt{,} \dots \texttt{,}N_n\texttt{)} \\ \text{type}(N_i, \pi) = t_i \qquad \Gamma, T \vdash e_i : t_i \qquad T \vdash \text{kind}(t_i) = \texttt{base} \end{array}}{\Gamma, T \vdash C\texttt{(}e_1\texttt{,} \dots \texttt{,}e_n\texttt{)}\ \texttt{:}\ \texttt{G(}s_t\texttt{,}\pi\texttt{)}}$$

Figure 27 summarizes the typing rules for the language with parametric types.

There is a very subtle problem with the typing rule for type abstractions. We demonstrate it by the following example.

We consider the expression

```
ΛX : any. λx : X.
  ΛX : base. (λy : X. Succ(y))(x)
```

The outer type abstraction assigns the kind `any` to the type variable `X`. However, the inner type abstraction constrains the type variable `X` to the kind `base`. This means that inside the inner type abstraction the type variable `X` ranges over grammars types. The function `λy : X. Succ(y)` exploits this fact, since it supplies the argument `y` as argument to the constructor application `Succ(y)`. Since `x` was bound outside of the inner type abstraction it may be instantiated with any value, not only with tree values. Hence, the whole program is invalid and should consequently not be well-typed.

However, with the original formulation of the rule T-TABS it is possible to construct the following derivation tree. We abbreviate the grammar `G(s, (s -> Succ(X)))` by `G`. The problematic parts are highlighted with boxes.

$$\text{T-TABS}\ \dfrac{\text{T-FABS}\ \dfrac{\text{T-TABS}\ \dfrac{\text{T-FAPP}\ \dfrac{\dots}{\{\boxed{\texttt{x : X}}\}, \{\boxed{\texttt{X : base}}\} \vdash \texttt{(λy:X.Succ(y))(x)} : G}}{\{\texttt{x : X}\}, \{\texttt{X : any}\} \vdash \texttt{ΛX:base.(λy:X.Succ(y))(x) : ∀X.G}}}{\emptyset, \{\texttt{X : any}\} \vdash \texttt{λx:X.ΛX:base.(λy:X.Succ(y))(x) : X->∀X.G}}}{\emptyset, \emptyset \vdash \texttt{ΛX:any.λx:X.ΛX:base.(λy:X.Succ(y))(x) : ∀X.X->∀X.G}}$$

The upper instance of the rule T-TABS overwrites the binding for the type variable `X` within the kinding context. However, the mapping $x : X$ in the typing context still refers to the type variable which was bound by the outer type

$$\text{T-VAR}\ \frac{(x : t) \in \Gamma}{\Gamma, T x : t} \qquad \text{T-SUB}\ \frac{\Gamma, T \vdash e : t_{sub} \qquad T \vdash t_{sub} <: t}{\Gamma, T \vdash e : t}$$

$$\text{T-TABS}\ \frac{X \notin T \qquad \Gamma, T \cup \{X : k\} \vdash e : t}{\Gamma, T \vdash \Lambda X{:}k.e : \forall X{:}k.t}$$

$$\text{T-TAPP}\ \frac{\Gamma, T \vdash e : \forall X{:}k.t_1 \qquad T \vdash \text{kind}(t) = k}{\Gamma, T \vdash e\texttt{<}t\texttt{>} : t_1[X \mapsto t]}$$

$$\text{T-FABS}\ \frac{T \vdash \text{kind}(t_1) = k \qquad \Gamma \cup \{x : t_1\}, T \vdash e : t_2}{\Gamma, T \vdash \lambda x{:}t_1.e : t_1 \texttt{->} t_2}$$

$$\text{T-FAPP}\ \frac{\Gamma, T \vdash e_1 : t_1 \texttt{->} t_2 \qquad \Gamma, T \vdash e_2 : t_1}{\Gamma, T \vdash e_1\texttt{(}e_2\texttt{)} : t_2}$$

$$\text{T-FIX}\ \frac{\Gamma, T \vdash e : t \texttt{->} t}{\Gamma, T \vdash \texttt{fix(}e\texttt{)} : t}$$

$$\text{T-CAPP}\ \frac{\begin{array}{c}\pi(s_t) = C\texttt{(}N_1\texttt{,} \ldots \texttt{,}N_n\texttt{)} \\ \text{type}(N_i, \pi) = t_i \qquad \Gamma, T \vdash e_i : t_i \qquad T \vdash \text{kind}(t_i) = \texttt{base}\end{array}}{\Gamma, T \vdash C\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} : \texttt{G(}s_t\texttt{,}\pi\texttt{)}}$$

$$\text{T-MATCH}\ \frac{\begin{array}{c}T \vdash \text{kind}(t_e) = \texttt{base} \qquad \Gamma, T \vdash e : t_e \qquad \text{context}(pat_i, t_e) = \Gamma_i \\ \text{exhaustive}(\{pat_1, \ldots, pat_n\}, t_e) \qquad \Gamma_i \cup \Gamma, T \vdash e_i : t\end{array}}{\Gamma, T \vdash \texttt{match}\ e\ \texttt{:}\ t_e\ \texttt{case}\ pat_1 \texttt{->}\ e_1 \ldots \texttt{case}\ pat_n \texttt{->}\ e_n : t}$$

Figure 27: Typing Rules for the Language with Parametric Types

abstraction. This discrepancy allows us to close the derivation tree with the following two subderivations.

$$\text{T-FABS}\ \frac{\text{T-CAPP}\ \dfrac{\ldots}{\{\texttt{x} : \texttt{X}, \texttt{y} : \texttt{X}\}, \{X : \texttt{base}\} \vdash \texttt{Succ(y)} : G}}{\{\texttt{x} : \texttt{X}\}, \{\texttt{X} : \texttt{base}\} \vdash \lambda\texttt{y:X.Succ(y)} : \texttt{X->G}}$$

$$\text{T-VAR}\ \frac{(\texttt{x} : \texttt{X}) \in \{\texttt{x} : \texttt{X}\}}{\{\texttt{x} : \texttt{X}\}, \{\texttt{X} : \texttt{base}\} \vdash \texttt{x} : \texttt{X}}$$

The solution we propose is to rename the type variable bound by a type abstraction appropriately in order to avoid name clashes. This renaming is implicitly enforced in the version of the typing rule shown in figure 27 by the requirement $X \notin T$.

9.5 Subtyping

In the context of type variables the definition of subtyping becomes more complex. In the following, we propose an approach to define and compute the subtype relation for parametric types. The intention is to give a starting point for further exploration rather than to present a complete solution. Therefore, we leave the formal correctness proof to future work.

Semantically, we want the following definition of subtyping: A type A is a subtype of type B if and only if for all instantiations $\sigma = \{X_i \mapsto t_i\}$ for free type variables X_i, $\sigma(A)$ is a subtype of $\sigma(B)$. But since the universe of types is not finite, there is no possibility to literally check this requirement for all instantiations. Hence, we need a structural way of computing the subtype relation. Furthermore, we need to discuss what subtyping means in the context of quantified types.

We first deal with the case of grammars with type variables. There is one very important observation. Suppose we have two type variables `X` and `Y`. In this case we can both find an instantiation $\{\texttt{X} \mapsto t_1, \texttt{Y} \mapsto t_2\}$ such that $t_1 <: t_2$ and an instantiation $\{\texttt{X} \mapsto t_3, \texttt{Y} \mapsto t_4\}$ such that $t_4 <: t_3$. Since a polymorphic type t can only be a subtype of another polymorphic type t' if the relation holds for all possible instantiations, a type variable can never be in the subtype relation with a different type variable.

As discussed in Section 3, the tree grammars we use are not closed under complement. This comes from the fact that we are not able to express a most general grammar, i.e. a grammar that describes all possible values. This also implies that we cannot write down two grammars G_1 and G_2 such that the union $G_1 \cup G_2$ contains all values, since tree grammars are closed under union. This has interesting consequences for the subtyping of parametric grammars.

Example 9.11
We consider the grammar

```
G1 := G(s, (s -> C(X)))
```

where `X` is a type variable. It is not possible to construct a grammar `G2` such that `G1` $<:$ `G2` and `G2` does not literally use the type variable `X` at the corresponding position. This means it is not possible to define `G2` as

```
G2 := G(s, (s -> C(n1) | C(n2),
            n1 -> ..., n2 -> ...))
```

since we cannot define two grammars `n1` and `n2` such that the union `n1` $\cup$ `n2` is a supertype of any possible type.

The last example shows that we have to treat type variables in grammars as named “black boxes”. The only thing we know about these black boxes it that two boxes with the same name are equal. All other types except for the empty type are not comparable to a black box in terms of the subtype relation. This observation implies that we can reduce the subtyping check for parametric tree grammars to a subtyping check without type variables by using a unique and fresh constructor name in place of each type variable. Using a constructor without arguments in place of type variables ensures that the resulting language cannot be treated as empty language. And the freshness of the constructor name ensures that the resulting language is incomparable to all other non-empty languages in terms of the subtype relation.

Example 9.12
We consider the parametric grammars

```
G1 := G(s, (s -> Base(X) | Succ(s)))
```

and

```
G2 := G(s, (s -> Base(X) | Succ(s) | Plus(s,s)))
```

where `X` is a type variable. In order to check the relation `G1` <: `G2` we introduce a fresh constructor name `Foo` for the type variable `X` and transform the grammars as follows.

```
G1' := G(s, (s -> Base(f) | Succ(s),
             f -> Foo))
G2' := G(s, (s -> Base(f) | Succ(s) | Plus(s,s),
             f -> Foo))
```

After this transformation the resulting grammars do not contain any type variables any more and we can proof the relation `G1'` <: `G2'` with our usual algorithm. According to our discussion above, this result implies the subtype relation `G1` <: `G2`.

On the other hand, neither of the following two grammars `G3` and `G4` is in a subtype relation with the grammar `G1'`.

```
G3 := G(s, (s -> Base(f) | Succ(s),
            f -> Zero))

G4 := G(s, (s -> Base(f) | Succ(s),
            f -> ))
```

It still remains to discuss how to lift this notion of subtyping to quantified types. The main goal of subtyping in the context of a type system is to ensure that any expression of type t_1 can be used in a context where an expression of type t_2 is expected whenever $t_1 <: t_2$.

Expressions with quantified types have the form $\Lambda X{:}k.e$. Suppose we have the type abstraction `ΛX:k1.e` with type `∀X:k1.t1` and want to use it in a context that expect an expression of type `∀X:k2.t2`. Using a type abstraction means applying it to a concrete type. This implies two things.

- Since the context expects an expression of type `∀X:k2.t2`, the expression can be instantiated with any type of kind `k2`. Hence, as to be able to use the expression `ΛX:k1.e` the subkind relation `k2` $<:_k$ `k1` must hold. This shows that type abstractions are contravariant in their argument, just like functions.

- The result of the type application is expected to have the type `t2`[`X`$\mapsto$`t`] for whatever type `t` has been supplied. Hence, for any type t we need that `t1`[`X`$\mapsto$`t`] <: `t2`[`X`$\mapsto$`t`] holds. With that condition we are back to our original formulation of subtyping for parametric types.

The complete definition of subtyping for parametric types in summarized in Figure 28. Thereby, the operation "tv2const" turns each type variable in a grammar into a unique fresh constructor name and uses the normal subtyping definition for non-parametric grammars. The rule S-FUN is the same we used in the language with higher order functions (see Section 7). Similar to function types, quantified types are contravariant in the argument type. Therefore we have to check that the kind k_2 is a subkind of the kind k_1. The subkinding relation consist of the reflexive closure of `base` <: `any`. Note, that the subtype relation is only defined on well-kinded types.

$$\text{S-VAR}\ \frac{}{X <: X}$$

$$\text{S-BASE}\ \frac{\text{tv2const}(\texttt{G(}s_1\texttt{,}\pi_1\texttt{)}) <: \text{tv2const}(\texttt{G(}s_2\texttt{,}\pi_2\texttt{)})}{\texttt{G(}s_1\texttt{,}\pi_1\texttt{)} <: \texttt{G(}s_2\texttt{,}\pi_2\texttt{)}}$$

$$\text{S-FUN}\ \frac{t_1' <: t_1 \qquad t_2 <: t_2'}{t_1 \texttt{->} t_2 <: t_1' \texttt{->} t_2'}$$

$$\text{S-ALL}\ \frac{k_2 <:_k k_1 \qquad t_1[X_1 \mapsto X] <: t_2[X_2 \mapsto X] \qquad X \text{ fresh}}{\forall X_1{:}k_1.t_1 <: \forall X_2{:}k_2.t_2}$$

Figure 28: Subtyping for Parametric Types

9.6 Soundness

The soundness proof for the polymorphic type system is more complex than the simply typed counterpart. It goes beyond the scope of this thesis to provide the full soundness proof. In particular, the correctness proof of the subtype relation defined in the previous Section is needed. However, in the following we sketch the necessary proof steps.

For the soundness proof two important lemmas are needed. The first one concerns the canonical forms of values.

Lemma 9.1 : Canonical Forms of Values
A value of a grammar type is a tree generated by this grammar.

A value of type t_1->t_2 has the form $\lambda x{:}t_1.e$, where $\{x : t_1\}, \emptyset \vdash e : t_2$.

A value of type $\forall X{:}k.t$ has the form $\Lambda X{:}k.e$, where $\emptyset, \{X : k\} \vdash e : t$.

In order to show that type preservation holds in the case of type applications we need a substitution lemma for types.

Lemma 9.2 : Substitution Lemma for Types
$\Gamma, T \cup \{X : k\} \vdash e : t$, $X \notin \Gamma, T$ and $T \vdash \mathrm{kind}(t_x) = k$ implies
$\Gamma, T \vdash e[X \mapsto t_x] : t[X \mapsto t_x]$

Furthermore, the Substitution Lemma for values (see Lemma 4.2) needs to be extended for the new language constructs.

10 Implementation

We have implemented a prototype version of the simply typed language on top of PLT Redex [20]. This implementation includes the implementation of the operational semantics and the type system. Also the case study from Section 6 has been implemented on top of that framework. The code is available at `https://github.com/haselhorst/Tree-Grammars`.

The goal of this Section is twofold. First, we describe some of the algorithms on regular tree grammars that we use in our implementation. And second, we discuss our experience of implementing our language withing PLT Redex. Since like our language, PLT Redex is also a domain specific language for term transformations it is very interesting to compare them to each other.

10.1 Algorithms on Tree Grammars

The main operations on tree grammars that we use in the formalization of our language are union, intersection, inclusion and check for emptiness. Furthermore, we assume the grammars to be normalized, hence we need an algorithm which normalizes a given grammar. In the following we discuss the general ideas of the algorithms. For more details we refer the reader to our implementation.

Normalization

A regular tree grammar is defined as a start symbol s and a set of productions π, written as G(s,π). While in general, the right hand sides of productions can consist of any tree, normalized grammars only allow productions of the form n->C(n_1,...,n_k).

The algorithm for normalization works in two steps.

- First, all trees are flattened. This is done by introducing a new nonterminal for each subtree. For example, the production

  ```
  n -> C(D,E(m,F))
  ```

  is transformed into the set of productions

  ```
  n  -> C(n1,n2)
  n1 -> D
  n2 -> E(m,n3)
  n3 -> F
  ```

- Second, all productions of the form n_1 -> n_2 need to be eliminated. This is done by computing the transitive closure of right hand sides for each nonterminal. As an example we consider the grammar

  ```
  G(n, (n    -> even | odd,
        even -> Zero | Succ(odd),
        odd  -> Succ(even)))
  ```

The productions for the nonterminal `even` and `odd` are already in normalized form. The production `n -> even` tells us that anything we can derive from `even` can also be derived from `n`. Hence, the transitive closure for this production is `n -> Zero | Succ(odd)`. The same argument gives us the production `n -> Succ(even)` as transitive closure of the production `n -> odd`. Hence, the resulting grammar looks as follows.

```
G(n, (n    -> Zero | Succ(odd) | Succ(even),
      even -> Zero | Succ(odd),
      odd  -> Succ(even)))
```

After these two steps, the resulting grammar is in normalized form.

Union

The union of two grammars G_1 and G_2 is defined as the grammar than contains exactly all trees that can be derived by G_1 or by G_2. Hence, we can compute the union of $G_1 = \texttt{G(}s_1\texttt{,}\pi_1\texttt{)}$ and $G_2 = \texttt{G(}s_2\texttt{,}\pi_2\texttt{)}$ as $G_1 \cup G_2 = \texttt{G(}s\texttt{,}\{s\texttt{->}\,\pi_1(s_1) \mid \pi_2(s_2)\} \cup \pi_1 \cup \pi_2\texttt{)}$, where s is a fresh nonterminal name.

This works well as long as the set of nonterminals in G_1 is distinct from the set of nonterminals in G_2. Otherwise we get name clashes when unifying π_1 and π_2. Since nonterminals are bound within a grammar they can be renamed without changing the meaning of the grammar. Hence, before computing the union of two grammars we need to rename nonterminals appropriately as to avoid name clashes.

Example 10.1
We consider the grammar

```
G(s, (s -> A(x,y),
      x -> X, y -> Y))
```

and the grammar

```
G(s, (s -> B(n),
      n -> N))
```

Both grammars contain the nonterminal `s`, hence we have to rename it first. We choose to rename the nonterminal `s` to `s1` in the second grammar which yields

```
G(s1, (s1 -> B(n),
       n  -> N))
```

Now, we can compute the union of the two grammars as follows.

```
G(s_fresh, (s_fresh -> A(x,y) | B(n),
            s       -> A(x,y),
            x       -> X
            y       -> Y
            s1      -> B(n)
            n       -> N))
```

Note, that the union of two grammars G_1 and G_2 is normalized if both G_1 and G_2 are normalized.

Check for Emptiness

A grammar generates the empty language if no trees can be derived. A derivation always starts with the start symbol of the grammar. Hence, the grammar is empty if either there are no productions for the start symbol or if none of the right hand sides of the productions generates a tree.

A right hand side has the form C`(`n_1`, ... ,`n_k`)`. We cannot derive any trees from this right hand side if any of the nonterminals n_1 to n_k describes the empty language. Hence, we have to recursively check for emptiness of the grammars starting with the nonterminals n_1 to n_k respectively.

The only problem are infinite derivations. For example, the grammar

```
G(s, (s -> Succ(s)))
```

is empty, but the algorithm sketched above would get into an infinite loop. The solution is to keep track of the nonterminals we have already seen. If we arrive at the same nonterminal for the second time we can conclude that the language is empty.

Example 10.2
We consider the grammar

```
G(s, (s -> A(x,y),
      x -> X,
      y -> A(x,y)))
```

We start with the start symbol `s`. The only right hand side is `A(x,y)`. Hence, `s` describes the empty language if any of `x` or `y` describes the empty language. We thus recurse into `x` and `y`.

The production `x -> X` tells us that `x` is not empty. On the other hand, the production `y -> A(x,y)` requires us to recurse into `x` and `y` again. As above, `x` is not empty. Since we are in course of visiting the nonterminal `y` for the second time, we can conclude that `y` is empty. This implies the emptiness of the whole grammar.

Inclusion and Intersection

For testing inclusion of two grammars we need to check whether all trees that can be derived by the first grammars can also be derived by the second grammar. It is well-known that the worst case complexity of the inclusion problem for regular tree grammars is EXPTIME-complete [5]. This comes from the fact that a production for the same nonterminal may contain multiple alternatives with the same constructor and the same arity.

We first consider the simplified setting, where each constructor can only be used once with the same arity within the productions for the same nonterminal. In this case the inclusion problem can be implemented in a structural way. Suppose we have grammars $G_1 = \texttt{G(}s_1\texttt{,}\pi_1\texttt{)}$ and $G_2 = \texttt{G(}s_2\texttt{,}\pi_2\texttt{)}$ and want to check if $L(G_1)$ is included in $L(G_2)$. Then for each constructor application $C\texttt{(}n_1\texttt{,} \dots \texttt{,}n_k\texttt{)}$ in $\pi_1(s_1)$ we need a corresponding constructor application $C\texttt{(}m_1\texttt{,} \dots \texttt{,}m_k\texttt{)}$ in $\pi_2(s_2)$. Furthermore, for each i we have to recursively check if the inclusion holds for the grammars $\texttt{G(}n_i\texttt{,}\pi_1\texttt{)}$ and $\texttt{G(}m_i\texttt{,}\pi_2\texttt{)}$. Like in the algorithm for checking for emptiness of grammars, we can conclude that the inclusion holds if we visit the same pair of nonterminals n_i and m_i for the second time during the computation.

Matters are different if multiple constructors with the same arity are allowed withing the productions for the same nonterminal.

Example 10.3
We consider the grammar

```
G1 := G(s, (s -> C(x),
            x -> A | B))
```

and the grammar

```
G2 := G(s, (s -> C(a) | C(b),
            a -> A, b -> B))
```

Both grammars generate the same language, namely the set $\{\texttt{C(A)},\texttt{C(B)}\}$. Hence, the inclusion $\texttt{G1} \subseteq \texttt{G2}$ should hold. However, the grammar with startsymbol `x` is neither included in the grammar with startsymbol `a` nor in the grammar with startsymbol `b`. Rather, the values generated by `x` are partly generated by `a` and partly by `b`.

Note, that in the last example it would have been possible to transform `G2` into an equivalent grammar `G2'` with only one alternative of the form `C(n)` by introducing a new nonterminal `n` defined as the union of `a` and `b`. However, this is not always possible. A counterexample is the grammar

```
G(s, (s -> A(b,c) | A(d,e),
      b -> B, c -> C,
      d -> D, e -> E))
```

This grammar generates the language $\{\texttt{A(B,C)},\texttt{A(D,E)}\}$. If we introduce the new nonterminals `bd` defined as the union of `b` and `d` and the nonterminal `ce` defined as the union of `c` and `e` we get the following grammar.

```
G(s, (s  -> A(bd,ce),
      bd -> B | D,
      ce -> C | E))
```

This grammar is not equivalent to the first one, since it generates the language $\{$`A(B,D),A(B,E),A(D,C),A(D,E)`$\}$.

We get back to the grammars from example 10.3. To conclude that the grammar `G1` is included in the grammar `G2` we need first to subtract all values from `C(x)` that are generated by `C(a)`. Then, we can check that the resulting set of values is included in `C(b)`. Hence, we need a means to compute the difference between two grammars.

Checking the inclusion of a grammar G_1 in a grammar G_2 can then be defined by computing the difference $G_2 - G_1$ and checking for emptiness of the result. In a similar way, the intersection of two grammars can be expressed by the difference operator: $G_1 \cap G_2 = G_1 - (G_1 - G_2)$.

In the following we present the general ideas of how to compute the difference between two grammars.

Suppose we have two grammars $G_1 =$ `G(`s_1`,`π_1`)` and $G_2 =$ `G(`s_2`,`π_2`)` and want to compute the difference $G_1 - G_2$. We have to subtract right hand sides C`(`m_1`, ... ,`m_k`)` in $\pi_2(s_2)$ from all corresponding right hand sides of the form C`(`n_1`, ... ,`n_k`)` in $\pi_1(s_1)$.

We first consider the case of subtracting the one right hand side C`(`m_1`, ... ,`m_k`)` from another right hand side C`(`n_1`, ... ,`n_k`)`. If we recursively apply the algorithm to all pairs (n_i, m_i) we get the grammars $n_i - m_i$. To get a grammar that describes all values that are in C`(`n_1`, ... ,`n_k`)` $-$ C`(`m_1`, ... ,`m_k`)` we have to observe the following. Suppose we have a value $v_1 \in n_1 - m_1$. This value is generated by n_1 but not by m_1. This implies that any value of the form C`(`v_1`, ... ,`v_k`)` cannot be generated by the grammar C`(`m_1`, ... ,`m_k`)`, independently of the values v_2 to v_k. Hence, all values in C`(`$n_1 - m_1$`,`n_2`, , ... ,`n_k`)` are in C`(`n_1`, ... ,`n_k`)` $-$ C`(`m_1`, ... ,`m_k`)`. The same holds for all other arguments of the constructor. Thus subtracting constructors with k arguments from each other yields k right hand sides in the result.

Additional right hand sides in π_2 of the corresponding form have to be subtracted from all resulting right hand sides which leads to an exponential blowup.

Although the worst case complexity of the inclusion problem is exponential, we believe that this is not a severe problem in practice. The problem of exponential complexity only arises if a grammar contains multiple productions with multiple instances of the same constructor. Our experience from the case studies is that such multiple constructor instances are only necessary in very rare cases. However, this needs to be further explored.

10.2 PLT Redex as Term Transformation Language

PLT Redex is a domain specific language for specifying languages and their operational semantics.

Languages in Redex are described by a restricted form of context free grammars. The restriction is that right hand sides of productions have to be well-formed s-expressions [21]. For example, our definition of the language of types looks as follows.

```
(define-language Type
  (x omitted)
  (c omitted)
  (raw-t (start raw-prod ...))
  (t (start prod ...))
  (start x)
  (nonterm x)
  (raw-prod (nonterm raw-right ...))
  (prod (nonterm right ...))
  (raw-right nonterm
             (c nonterm ...))
  (right (c nonterm ...)))
```

The ellipsis mean zero or more repetitions of the preceding element. For example, the line `(t (start prod ...))` means that the nonterminal `t` can be replaced by an opening bracket followed by the nonterminal `start`, any number of instances of the nonterminal `prod` and a closing bracket.

Hence, language definitions are very similar to our regular tree grammars. Since in the language definitions all right hand sides of productions need to be correctly bracketed, terms of the language are well-formed s-expression. S-expression can be seen as a linear representation of tree structured data. The main difference to regular tree grammar is that the language definition does not fix one distinguished nonterminal as start symbol. Rather, each nonterminal itself defines a language, namely the set of terms that can be derived from this nonterminal.

One main tool for working with terms are metafunctions. Metafunctions allows to define domain specific functions working on terms of the embedded languages. For example a metafunction of the form

```
(define-metafunction Type
  ...)
```

defines a function that works on terms of the type language we defined above.

Metafunctions use pattern matching as their main deconstruction mechanism. For example, the following definition computes the start symbol of a tree grammar.

```
(define-metafunction Type
  [(startsym (start raw-prod ...)) start])
```

`startsym` is the name of the function and `(start raw-prod ...)` is the pattern which is matched against the input term. Patterns in Redex are much more expressive than the pattern in our language. A nonterminal matches all values that can be derived by that nonterminal. In the example above, the nonterminal `start` matches against any value that can be derived by `start` in the context of the language definition for `Type`. The ellipsis `raw-prod ...` matches any numbers of terms that can be derived by the nonterminal `raw-prod`. While in our language, patterns are trees that may contain variables, patterns in Redex correspond to right hand sides of productions. The task of the nonterminals in patterns is twofold. First they are used to check if the input has the expected form. And second they give names to the corresponding subparts of the input. In the example above we return the start symbol of the grammar by using the nonterminal `start` as result.

Metafunctions can be optionally given a signature. For example, the `startsym` function from above can be rephrased as follows.

```
(define-metafunction Type
  startsym : raw-t -> start
  [(startsym (start raw-prod ...)) start])
```

Signatures of functions can be compared to type annotations in our language: they specify the range of valid input values and the range of possible output values. However, signatures in Redex are not statically verified but only checked at runtime. When either the input or the output of a function does not match its contract a runtime error is produced. In the course of implementing our language in Redex we have experienced the following. The signatures have been a lot of help in debugging the code. Every time a runtime error was thrown due to contract violations it clearly showed whom to “blame”. If the input contract was violated, the error was located at the calling position. And if the output contract was violated, the function implementation had to be wrong. The debugging process would have been much harder without dynamic type checking. However, with a static type system, most of the errors would have been caught before even executing the code. This experience was an affirmation for the usefulness of static type checking.

But apart from not being statically checked, the function signatures in Redex are closely related to our type annotations. Each nonterminal represents a language, namely the set of terms it generates. Hence, a language definition in Redex can be seen as a scoping mechanism for grammar definitions. This is also the reason why a metafunction needs to know the name of the language it works on. Otherwise the system works in a complete structural way: a term belongs to a language represented by a nonterminal if it can be derived from this nonterminal. This corresponds to our notion of typing.

There are two severe shortcomings that we experienced in the course of implementing our language. The first shortcoming is that metafunctions are first order. There were many situations where we had to apply a certain function to a list of terms or were we needed to accumulate a list of terms in a certain way. Since metafunction are first order, we ended up in manually writing concrete maps and folds for different functions. All of these function have the same structure but with first order functions we had no way to abstract over this boilerplate code.

The second shortcoming is the lack of polymorphic annotations. In our implementation there are many functions that work the same of different types of input values. For example we have functions that add an element to a list or check if a list contains a certain element. Redex offers the symbol `any` which matches any value. Hence, we could for example give the signature `any (any ...) -> (any ...)` to the function `add` that adds an element to a list. However, this does not capture exactly our intention. A signature of the form `X (X ...) -> (X ...)` for all `X` would have been more appropriate. The more general signature using `any` allows to use lists containing elements of different types. This has lead to errors that were very hard to debug, because the dynamic type checker was not able to find them. Of course we could have written one instance of the function `add` for each type. But since `add` is not the only function with this problem, this would have implied an explosion of the size of the implementation caused by this additional boilerplate code. A polymorphic type system is a much better solution to this problem.

These two shortcoming, namely the lack of higher order functions and of polymorphism, have to a great part influenced and motivated the kind of extensions we incorporated into our language.

11 Related Work

Program transformations are an important domain in the universe of metaprogramming. The applications include static analysis of programs, code optimization, desugaring, etc., to only mention a few of them. The common ground of all these applications is that they work on an abstract syntax tree representation of programs. There is much research centered around the question how to design good languages and tools for program transformations. In the following we discuss the most relevant pieces of related work.

11.1 Data Types for Syntax Trees

Traditionally, term transformations are implemented within general purpose languages like C, Java or Haskell. Hence, the syntax trees need to be represented by available language constructs.

Regular tree grammars are closely related to algebraic datatypes as can be found for example in Haskell or ML. Syntax trees can be encoded by algebraic datatypes in a convenient way, since algebraic datatypes describe tree structured data. However, there is one main shortcomings with this approach. Algebraic datatypes are closed. This means that the set of constructors cannot be extended by new constructors within a program. However, in the area of term transformations it is often important to be able to add new nodes to syntax trees. Swierstra [29] shows that it is possible to encode extensible data types on top of algebraic data types. The main problem with the solution it that it is not convenient to use. Extensible data types cannot be used with the same syntax as algebraic data types but require very explicit construction and destruction operations.

Axelsson [2] uses the extensible data types mentioned above to develop a generic abstract syntax model for embedded languages. The abstract syntax trees are typed and support generic traversals. The core idea is to encode abstract syntax trees by a data type with two constructors. One constructor produces AST nodes while the second constructor encodes the application of AST nodes to arguments. One challenge thereby is to distinguish partially applied syntax trees from function-valued expression.

In our system the problem of closed datatypes does not occur, because constructor names are not tied to types. Any set of trees that can be defined by a regular tree grammar is a type. Furthermore, our system features structural subtyping. This enables us to express types containing only a subset of the values of another type which is not possible with algebraic datatypes. The case study in Section 8 discusses the differences and commonalities of algebraic data types and regular tree grammar types.

Mishra et. al. [22] approach the problem of closed data types in a different way. Instead of requiring data *definitions* they the programmer to use constructor names in their programs and automatically infer the corresponding types. Types in their system are represented as regular tree expressions, i.e.

trees with free variables and fixpoints.

Other very powerful mechanisms for expressing data types for syntax trees are generalized algebraic datatypes (GADTs) [28, 27] and the combination of traits and abstract types in the programming language Scala [26]. These advanced concepts allow to capture context sensitive constraints of the embedded language in a type-safe way. For example, it is possible to embed the simply typed lambda calculus in a type-safe way using GADTs in Haskell [33] or traits in Scala [11]. In Section 12 we discuss some ideas on how to extend regular tree grammars in order to be able to express such type-safe embeddings.

11.2 Domain Specific Term Transformation Languages

As an alternative to using general purpose languages to implement term transformations, several domain specific languages for term transformations have been developed recently.

One example is Stratego [31]. Stratego is a very expressive domain specific language for implementing transformations on tree shaped data. The most powerful language feature are the so called generic traversals. These generic traversals allow to apply a certain transformation to all nodes of a tree recursively. For example, Stratego offers the generic transformations “bottomup” or “topdown” which apply a given transformation recursively to all subtrees in a topdown or bottomup fashion respectively.

While Stratego offers very powerful mechanisms for term transformations it does not provide static guarantees about the results of transformations, since it is not statically typed. In contrast to Stratego, our language is type safe but is not yet able to express generic tree traversals.

Another example for a domain specific term transformation language is PLT Redex [20]. PLT Redex was mainly developed to study semantic properties of programming languages. PLT Redex allows to annotate term transformations with types describing the input and output domains. However, these type annotations are only checked at runtime. We have implemented a prototype version of our simply typed language on top of PLT Redex. In Section 10 we discuss how PLT Redex compares to our language.

11.3 Generic Traversals

We already mentioned the Stratego [31] as a programming language featuring generic traversals of trees. When working with tree data a programmer often needs to change some specific parts of the tree while leaving the rest unchanged [19, 3]. Without generic traversals this task leads to much boilerplate code which deconstructs and reconstructs the parts of the tree that don’t need to be changed. This motivates a lot of research in the area of generic traversals.

Lämmel et. al. [18, 19] propose generic traversal combinators as a way to generate tree traversal functions for arbitrary data types in the programming language Haskell. The core idea is to lift a function that transforms certain subtrees to a function that transforms complete trees as follows: if the subtree has the right type, the function is applied. Otherwise, a default operation is used. There are two kinds of combinators: generic transformations and generic queries. In generic transformations the default operation is the identity function. This means that transformations are able to change certain parts of a tree while leaving the remaining part unchanged. Generic queries return a fixed default value instead of using the identity function for the parts of the tree which are not affected by the function to be lifted. These traversal combinators are strongly typed, however the possible types are constrained to types that can be expressed by Haskell types. Since constructors are typed, transformations cannot change the type of a subtree. Hence, generic transformations have the type $\forall\alpha.\alpha \rightarrow \alpha$, while generic queries have the type $\forall\alpha.\alpha \rightarrow \tau$ for a fixed τ. Since Haskell supports parametric polymorphism, it is possible to write polymorphic traversals.

Brand et. al. [3, 30] extend the algebraic specification formalism of ASF+SDF [6] with generic tree traversals very similar the the work of Lämmel et. al. [19, 18]. They offer two different traversals, namely transformers and accumulators that are closely related to the generic transformations and generic queries in [19]. Transformers have to be type preserving, i.e. they cannot change the type of the tree during the traversal. Accumulators don't change a tree but only extract information. In contrast to [19] this system is mono-typed, i.e. does not support parametric polymorphism.

Dolstra [7] proposes a programming language closely related to Stratego featuring generic traversals. The main innovation is a type system for the strict subpart of the language. The type system is able to prove the safety of type preserving and type unifying generic transformations similar to the generic transformations in [3] and [19].

As discussed in Section 12 our goal is to extend our language with the possibility to express generic traversals. We believe that our language is a good starting point for exploring how to express a much broader range of tree traversals for the following reasons. First, constructors are not typed, i.e. they can be used essentially everywhere. And second, due to structural subtyping it is possible to assign more precise types to terms. This is especially important for implementing desugarings, since usually the goal is to remove certain constructors from the abstract syntax trees. Hence, in order for the type system to ensure that the result does not contain any instances of that constructor any more, it is necessary to express this fact with a type. In both systems mentioned above [19, 30] this cannot be expressed. Instead it is necessary to use the more imprecise type $\forall\alpha.\alpha \rightarrow \alpha$ for such a transformation.

11.4 Grammars as Types

The most related work we know of that also exploits the idea of using languages as types is the work of Hosoya et. al. [14] in the context of XML. Their goal is to design a polymorphic type system for XML processing. The language is very similar to our first order calculus. It includes first order functions and pattern matching as main language constructs. While we use regular tree grammars as types, they define so called regular expression types [16] which are equivalent to regular tree grammars from the expressiveness point of view.

The main difference to our first order calculus is the definition of patters. While we define patterns as trees that can contain variables, in their system patterns are types. The semantic is as follows: a tree matches a pattern if it has the type of the pattern. In addition patterns can give names to subparts of the types which are bound to the matching part of the corresponding value during evaluation. This more flexible definition of patterns allows them to conveniently express the kind of patterns which are often needed during XML transformations. However, this additional flexibility has a price. Formalizing the semantics and the typing rules for pattern matches becomes much more complex. There are alone two publications [15, 13] concerning the semantics of pattern matches. Furthermore, they only support linear patterns, i.e. patterns where each pattern variable occurs only once while our language can cope with multiple instances of pattern variables.

The second difference concerns the definition of parametric polymorphism. They support a weaker form of polymorphism, namely let-polymorphism, while we extended our language with the full flexibility of universal types. Furthermore, they employ a syntactic definition of parametricity: type parameters are treated as syntactic markers instead of as a semantic abstractions. The key motivation for using this approach is the high complexity of computing the subtype relation in the context of quantification. This high complexity comes from the fact that their types are closed under complement. Especially, it is possible to define a most general type. In contrast to that, we assume an infinite set of constructor names which implies that the tree grammars are not closed under complement. As discussed in Section 9.5 we are able to reduce the subtyping of parametric tree grammars to the subtyping of non-parametric tree grammars without the need to switch to a syntactic definition of parametricity. In addition, it is not clear if the marking approach can be used in a system with higher order functions.

12 Future Work

The results of this theses raise several opportunities for future work. In the following we discuss some ideas.

Improvements to the Existing System

In the context of the language with parametric types there are several open questions. First, the soundness proof is not yet complete. In addition, much of the complexity of the type system stems from introduction of the kind system separating grammar types from the other types. Therefore, it would be interesting to investigate if a different formulation of the system leads to a simpler formalism.

In our system, type abstractions and type applications are part of the expression language. However, functional languages featuring parametric polymorphism like Haskell or ML don't require the programmer to explicitly write them down. Rather, they infer the correct instantiations for type variables automatically. We think that this inference if crucial to the usability of a language. Thus, we need to examine if and how type inference can be incorporated into our language.

There are two different directions for further extending the existing system. The first direction is to rise the expressiveness of types by using more powerful classes of grammars. The second direction is to add more powerful language constructs like generic traversals to the language and investigate how the type system needs to be extended to cope with these constructs.

Expressiveness of Types

In our first case study (see Section 6) we showed how to embed a small example language into our metalanguage in a type-safe way by defining a grammar that only allowed to express well-typed programs. This type-safe embedding enabled us to write a total evaluator for programs written in the embedded language. However, with regular tree grammars as types is not always possible to express type-safe embeddings, since regular tree grammars are not able to capture context sensitive constraints.

One idea is to use arbitrary tree grammars instead of regular tree grammars as types. Tree grammars in general allow productions of the form $\alpha \to \beta$ where α and β are tree expressions over a set of nonterminal names and a set of variable names [5]. Hence each production can be seen as a context-sensitive rewriting rule. For example we can express the following language consisting of arithmetic and boolean expressions, if-statements and pairs in a type-safe way.

```
G(s, (s(Num)        -> Zero | Succ(s(Num)),
      s(Bool)       -> True | False | IsZero(s(Num)),
      s(Pair(x,y)) -> Pair(s(x),s(y)),
      s(x)          -> If(s(Bool),s(x),s(x))
                        | First(s(Pair(x,t)))
                        | Second(s(Pair(t,x))),
      t             -> Num | Bool | Pair(t,t)))
```

Tree grammars are very powerful. For example they can simulate a Turing machine [5]. But this expressiveness probably makes them unsuited for use in a type system, because many interesting problems like inclusion of grammars are not decidable any more. A compromise both in expressiveness and complexity between regular tree grammars and general tree grammars is the class of context-free tree grammars [5]. The question is whether context-free tree grammars are enough to get an expressiveness comparable to generalized algebraic datatypes.

Another idea for extending the type language is to go away from "pure" tree grammars and allow to use arbitrary types in productions. This means that we allow to use values of arbitrary type inside constructor applications. For example, we could use list of functions instead of only lists with tree elements. We think that this generalization leads to a very powerful calculus for functional programming with inductive datatypes. The main advantage of these inductive datatypes based on tree grammars over algebraic datatypes is their flexibility. Constructors are not tied to data definitions, especially they are not typed. This allows us to define structural subtyping on the inductive datatypes.

Generic Traversals

One very important tool for program transformations are generic tree traversals (see Section 11). Our calculus cannot yet express generic traversals, because it lacks certain abstraction mechanism as for example the possibility to apply a function to all children of an arbitrary constructor.

There are two alternatives to incorporate generic traversals into the language.

- We add generic strategies like "bottomup" or "topdown" that recursively apply functions to all subtrees as language primitives. The main challenge of this approach it to find appropriate types for these primitives. The disadvantage of traversals as language primitives is that the set of possible traversals is fix and thus a programmer cannot define custom traversals. He can only use the predefined ones.

- We add language constructs which enable to *express* generic traversals within the language. The following are some ideas for basic language constructs which are helpful for expressing generic traversals.

  Abstraction over Constructor Names This allows us to deconstruct arbitrary trees with pattern matching as long as we know the arity of the constructors. For example, the following program transforms any binary tree into a tree of the language `G(s, (s -> Leaf | Node(s,s)))`. Thereby `X` is a constructor variable.

```
f(x) = match x
          case X(a,b) -> Node(f(a),f(b))
          case X      -> Leaf
```

All Another useful construct is the language primitive “all” which applies a given function to all children of a constructor. The type of “all” would be something like

```
(L1 -> L2)
  -> G(t, (t -> X(L1,...,L1)))
  -> G(t, (t -> X(L2,...,L2)))
```

This shows that we need constructor polymorphism in types to express the type of “all”.

Try It is often the case that programmers only need to change certain parts of the tree while leaving the rest unchanged [19]. To this end, we can use a primitive language construct “try” which applies a function to a value only if the value has the right type and otherwise acts as identity function. Try takes as input a function of type `L1 -> L2` and an expression of type `L3` and applies the function to the expression if `L3 <: L1`. Hence, “try” has the result type `if L3 <: L1 then L2 else L3` which demands for a much more powerful type language.

13 Conclusion

The goal of this thesis was to design a core calculus for program transformations and investigate its formal properties. To this end we first identified a minimal set of language constructs which enables to express reasonably complex program transformations, namely pattern matching and recursive functions. On this basis we defined the syntax and semantics of the core calculus.

Furthermore, we specified a domain specific type system for our calculus using regular tree grammars as types. The type system ensures that well-typed programs do not lead to runtime errors. We formally proved the soundness of the type system with respect to the semantics of the calculus.

In order to explore the applicability of the calculus to express complex program transformations, we conducted a case study comprising the implementation of an evaluator for a small programming language. The result of the case study was twofold. First, it clearly showed that we can express complex program transformations. We even succeeded in writing a total evaluator for a language for which in functional programming languages with algebraic datatypes only partial evaluators can be expressed. But second, the case study also showed that our calculus lacks important abstraction mechanisms which leads to much boilerplate code.

This observation motivated further extensions of the calculus. In the course of this thesis, we explored two different language extensions, namely higher order functions and parametric types. To this end, we extended both the syntax and semantics of the core calculus with the corresponding language constructs and the type system. For the language with higher order functions we formally proved the soundness of the system. While the extension with higher order functions turned out to be very straightforward, parametric types essentially increased the complexity of the type system. Especially, the definition of subtyping in the context of parametric types and the introduction of a kind system proved to be challenging. Therefore, we proposed one possibility to define the type system and the computation of subtyping intended as a starting point for further exploration. A complete correctness proof went beyond the scope of this thesis and we left it as future work.

We believe that our calculus constitutes a solid basis for further research on program transformation languages. Especially due to its simplicity, the calculus is well-suited as a good tool for studying formal properties.

References

[1] Atkey, R., Lindley, S., and Yallop, J. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell* (New York, NY, USA, 2009), Haskell '09, ACM, pp. 37–48.

[2] Axelsson, E. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2012), ICFP '12, ACM, pp. 323–334.

[3] Brand, M., Klint, P., and Vinju, J. Term rewriting with Type-safe Traversal Functions. In *Workshop on Rewriting Strategies (WRS2002)* (2002), B. Gramlich and S. Lucas, Eds.

[4] Carette, J., Kiselyov, O., and Shan, C.-c. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming 19*, 5 (Sept. 2009), 509–543.

[5] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[6] Deursen, A. V., Heering, J., and Klint, P., Eds. *Language Prototyping: An Algebraic Specification Approach: Vol. V.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[7] Dolstra, E. First Class Rules and Generic Traversals for Program Transformation Languages. Tech. rep., Utrecht University, 2001.

[8] Erdweg, S., Rendel, T., Kästner, C., and Ostermann, K. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, Oct. 2011), ACM Press, pp. 391–406.

[9] Erdweg, S., Rieger, F., Rendel, T., and Ostermann, K. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of the 2012 symposium on Haskell symposium* (New York, NY, USA, 2012), Haskell '12, ACM, pp. 149–160.

[10] Hofer, C., and Ostermann, K. Modular Domain-Specific Language Components in Scala. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)* (Oct. 2010), ACM.

[11] Hofer, C., Ostermann, K., Rendel, T., and Moors, A. Polymorphic Embedding of DSLs. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)* (2008), ACM.

[12] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[13] Hosoya, H. Regular Expression Pattern Matching - A Simpler Design. Tech. rep., 2003.

[14] Hosoya, H., Frisch, A., and Castagna, G. Parametric polymorphism for XML. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005), POPL '05, ACM, pp. 50–62.

[15] Hosoya, H., and Pierce, B. Regular expression pattern matching for XML. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2001), POPL '01, ACM, pp. 67–80.

[16] Hosoya, H., Vouillon, J., and Pierce, B. C. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems 27*, 1 (Jan. 2005), 46–90.

[17] Jones, S. P., Washburn, G., and Weirich, S. Wobbly Types: Type Inference for Generalised Algebraic Data Types. Tech. rep., 2004.

[18] Lämmel, R. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming 54*, 1-2 (2003), 1 – 64.

[19] Lämmel, R., and Jones, S. P. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation* (New York, NY, USA, 2003), TLDI '03, ACM, pp. 26–37.

[20] Matthews, J., Findler, R. B., Flatt, M., and Felleisen, M. A visual environment for developing context-sensitive term rewriting systems. In *In Proceedings of the International Conference on Rewriting Techniques and Applications (RTA* (2004), Springer, pp. 301–311.

[21] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM 3*, 4 (Apr. 1960), 184–195.

[22] Mishra, P., and Reddy, U. S. Declaration-free type checking. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles*

of programming languages (New York, NY, USA, 1985), POPL '85, ACM, pp. 7–21.

[23] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. Practical type inference for arbitrary-rank types. *Journal of Functional Programming 17*, 1 (Jan. 2007), 1–82.

[24] Pierce, B. C. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[25] Rendel, T., Ostermann, K., and Hofer, C. Typed Self-Representation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2009).

[26] Rompf, T., and Odersky, M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering* (New York, NY, USA, 2010), GPCE '10, ACM, pp. 127–136.

[27] Schrijvers, T., Peyton Jones, S., Sulzmann, M., and Vytiniotis, D. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2009), ICFP '09, ACM, pp. 341–352.

[28] Sheard, T., Hook, J., and Linger, N. GADTs + Extensible Kinds = Dependent Programming A Programming Pearl for the 21st Century.

[29] Swierstra, W. Data types a la carte. *Journal of Functional Programming 18*, 4 (July 2008), 423–436.

[30] van den Brand, M. G. J., Klint, P., and Vinju, J. J. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology 12*, 2 (Apr. 2003), 152–190.

[31] Visser, E. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)* (May 2001), A. Middeldorp, Ed., vol. 2051 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 357–361.

[32] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. *Information and Computation 115* (1992), 38–94.

[33] Xi, H., Chen, C., and Chen, G. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), POPL '03, ACM, pp. 224–235.

Selbständigkeitserklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Katharina Haselhorst

Marburg, den 28.09.2012