Master Thesis
Master of Science Informatik

A Language for the Specification and Efficient Implementation of Type Systems

Pascal Wittmann

Technische Universität Darmstadt
Fachbereich Informatik
Software Technology Group

Prüfer: Prof. Dr. Mira Mezini
Betreuer: Dr. Sebastian Erdweg

Abgabetermin: 16. Oktober 2014

Erklärung

Hiermit versichere ich gemäß der Allgemeinen Prüfungsbestimmungen der Technischen Universität Darmstadt (APB) §23 (7), die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	(Pascal Wittmann)

Abstract

Type systems are important tools to detect semantic inconsistencies, to establish abstractions and to guide programmers in the development process. However, there is currently a lack of established tools supporting the development of type systems, tools like lexer and parser generators but for type systems. We introduce a declarative specification language for type systems that allows to specify type systems in a natural deductive style. We generate two products out of a type system specification: A first-order formula representation to facilitate the use of automated theorem provers and a type checker. The type checker generator uses results proven by automated theorem provers to check the applicability of optimization strategies and the first-order formula representation is also a reference implementation. Both results aim to accelerate the development cycle for type systems and to narrow the gap between theory and practice.

Contents

List of Figures

Chapter 1

Introduction

This chapter motivates the thesis, summarizes the contributions and gives an overview of the structure of the thesis.

1.1 Motiviation

Type systems ensure that programs are well-behaved. In other words, they ensure that programs have meaning in the sense of the semantics of the programming language. The type systems we focus on are static type systems and can be thought of as a static approximation of the program semantics. Besides ensuring that programs are well-behaved, type systems are means to establish abstractions and to enforce adherence to these abstractions. Type systems can provide explicit type annotations that serve as program documentation, which cannot become obsolete as type annotations are verified by the type system. During type checking the program under consideration can also be annotated with optimization hints for the compiler. All in all type systems can help to develop software more efficiently (cf. [PHR14] and [MHR$^+$12]).

Type systems are useful tools, particularly if they fit to the programming language and the application scenario. If that is not the case type systems can event get in the way. To ensure a type system fits to a programming language, e.g. in the context of a Domain Specific Language (DSL), it makes sense to adapt and modify existing type systems or to create new specialized type systems. Those specializations can lead to better error messages, more expressive type systems, and the detection of more errors (cf. [Thi02] and [vdBvdMSH10]). Currently there are only some tools that support the creation and adaption of type systems by generating type checkers from a type system specification ([MME$^+$10], [OZQG14], [Gas05], [TF05], and [Ber07]), but they are not established. Such generators accelerate the development of type checkers, make development less error prone and fit well into the language development workbench besides the long established lexer and parser generators.

Type checker generators that use a high-level specification language can also provide type theorists with a tool that allows them experiment with new type systems without long development delays. High-level specifications that are close to the formalism used by type theorists have also the advantage of narrowing the gap between type system specifications on paper and their implementation. Results shown for the paper version can be adapted for the implementation (depending on the correctness of the type checker generator). Such adaptions are not possible with handcrafted implementations. High-level specification languages are also suited to specify type systems from other areas, like language-based information-flow security [SM06].

A declarative specification can be translated into a first-order formula representation, which can be used to conduct proofs of these properties and thus raise the confidence regarding these properties. As the proofs of propositions about type systems have to change when the type system changes, it is desirable to make most of these changes automatically. Automated theorem provers can try to conduct those proves automatically. If the proof search fails one can give hints for proof search, e.g. by providing induction hypothesis. This establishes a direct connection between the specified type system, the proofs, and the generated type checker.

The translation of type system specifications into first-order formula representations also enables to provably check for and remove redundancies within the type system specification. As redundancies have a potential to slow down the generated type checker, the removal leads to different optimization strategies for the resulting type checker.

The goal of this thesis is to narrow the gap between theory and implementation of type systems. A declarative specification language is developed, in which type systems are represented close to the standard formalisms. Two things are generated from those specifications. First, a representation of the type system as first-order formulas, intended to support proving properties of the type system. Second, an efficient type checker that exploits facts proven by automated theorem provers for optimization.

1.2 Contributions

The main contributions of this thesis are:

1. A declarative specification language for static type systems, with support for natural deductive style typing rules and error messages. The specification language is implemented in Syntax Definition Formalism (SDF) and capable of using most SDF definitions of programming languages.

2. A transformation of type system specifications into equivalent first-order formula representations in the Thousands of Problems for Theorem Provers (TPTP) format. The first-order formula representations are suitable for type checking using theorem provers, theorem proving, and serve as a reference implementation for the type checker generator.

3. A generator that creates type checkers from type system specifications. This generator uses an intermediate representation for typing rules suitable for constraint generation and optimizes the intermediate representation using automated proofs, which check the applicability of optimizations.

Figure 1.1 shows how these contributions are connected. From a type system specification a first-order formula representation and a type checker is generated. The first-order formula representation is used in conjunction with an automated theorem prover to check the applicability of optimizations for the type checker.

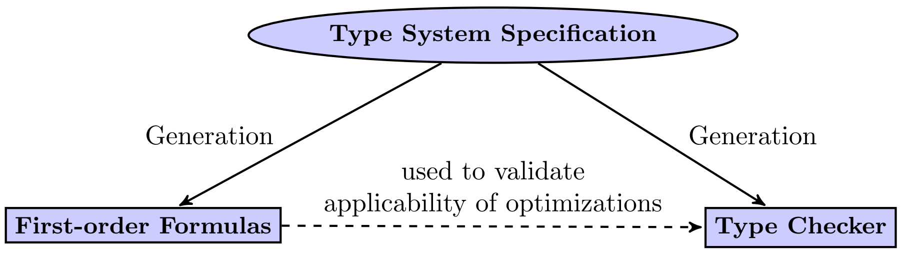

Figure 1.1: Connection between theorem proving and type system optimization

The source code of the specification language, the first-order formula transformation, and the type checker generator including all type system specifications is available as a Spoofax project at

`https://www.github.com/pSub/master-thesis`

1.3 Structure

The thesis is structured as follows. In Chapter 2 we introduce preliminaries and briefly discuss their alternatives. Then we introduce in Chapter 3 the specification language in detail and take a look at the implementation. Before we introduce in Chapter 5 the type checker generator and its optimization strategies in depth, we introduce the first-order model of the specifications in Chapter 4. In Chapter 6 we evaluate the type checker generator with two programming language type systems and one security type system. Related work is discussed in Chapter 7 and the thesis is concluded in Chapter 8 together with a prospect on future work.

Chapter 2

Preliminaries

2.1 Tools

This section introduces the tools used in this thesis. We argue what makes the tools suitable for achieving the goals of the thesis and give a short overview of the alternatives.

2.1.1 SDF

Syntax Definition Formalism (SDF) [HHKR89] is a formalism to define the syntax of formal languages in the tradition of the Backus-Naur Form (BNF). SDF allows to define lexical and context-free grammars and is, in contrast to BNF, modular. This modularity allows to compose SDF syntax definitions. There are also parser generators for SDF (see [Rek92]). In contrast to formalisms for traditional parser generators like Yacc [Joh75], SDF specifications are purely declarative, which increases the reusability of the specifications.

The modularity of SDF is a consequence of the fact that SDF is *generalized LR parsable* [Rek92]. Generalized parsable means that parsing might be indeterminate, i.e. the parser produces all possibilities for syntactic ambiguities. This might not seem to be an advantage as we have to deal with ambiguities now, but generalized parsing has some nice implications. First, the class of supported grammars is larger and therefore enforces less restrictions on the programmer and might allow more natural definitions of formal languages. Second, it enables the modularity of SDF, because the composition of generalized LR grammars is a generalized LR grammar. This does not hold in general for LR grammars.

Another property of SDF is that it is scannerless parsable [vdBSVV02], i.e. tokenization and parsing can be done in a single step. The advantages are that only one meta-language is needed and non-regular lexical structure are handled easily.

SDF supports specification of layout sensitive languages [ERKO12] like Python or Haskell. It enforces context insensitive layout constraints at parse time and context sensitive constraints at disambiguation time, i.e. all ambiguities that violate layout constraints are removed.

Example 1 shows a grammar that allows to write boolean expressions. Conjunctions are represented in the Abstract Syntax Tree (AST) with the constructor

Example 1.

```
module Bool
exports
  context-free start-symbols Bool
  sorts Bool
  context-free syntax
    "true"  -> Bool
    "false" -> Bool
    "~" Bool -> Bool {cons("Not")}
    Bool "&" Bool -> Bool {cons("And"),
                           layout("2.first.col < 2.left.col")}
```

And, written as an infix &. The layout constraint enforces, that if conjunctions span over multiple lines, the right-hand side of & needs to be indented. Negation is represented in the AST with the constructor Not and written as a prefix ~. True and false are written as true and false respectively. In this example all expressions belong to the same syntactic sort Bool. Sorts represent non-terminals in SDF. The start symbol is Bool as we only have a single sort.

SDF is used in this thesis to define the syntax of the type system specification language. Programming languages used in specifications are defined in SDF, too. Features like layout constraints make SDF a good tool for defining complex syntax for natural deduction rules. Further modularity and composability make it possible to define the syntax of the specification language independent of the target language.

2.1.2 Stratego/XT

Stratego [Vis01b] is a framework for the development of transformation systems. It consists of the transformation language Stratego and a set of tools (called XT) for tasks like parsing and pretty printing. The approach of Stratego is to use user-definable strategies for rewriting. Stratego distinguishes the following abstraction levels.

Transformation rules are basic rewrite rules on the structure of the AST.

Transformation strategies are the glue between the transformation rules. They combine rules, define the order of application and the traversal order of the AST. These strategies can be defined generically. *scoped dynamic rewrite rules* [Vis01a] allow to pass context information during the traversal, because transformation rules and strategies are context-free.

Transformation tools allow to compile transformation strategies into a stand-alone program. The interface between such programs is the ATerm format [VdBdJKO00] for ASTs.

Transformation systems describe a set of programs created by the transformation tools. A transformation system for a source-to-source transformation usually includes a parser and pretty printer.

Example 2.

```
module Bool
signature
  sorts Bool
  constructors
    Not   : Bool -> Bool
    And   : Bool * Bool -> Bool
rules
  Eval : Not(True)      -> False
  Eval : Not(False)     -> True
  Eval : And(True, x)   -> x
  Eval : And(x, True)   -> x
  Eval : And(False, x)  -> False
  Eval : And(x, False)  -> False
strategies
  eval = bottomup(repeat(Eval))
```

Example 2 shows a Stratego module that declares the constructors from Example 1. It has a rule `Eval` that reduces a term one step. The strategy `eval` applies rule `Eval` repeatedly from bottom to top.

We choose Stratego to implement the transformation into first-order formulas and the type checker generator as it interacts well with SDF, has facilities to integrate into tool-chains and it allows to write abstract and generic transformations.

2.1.3 Spoofax

Spoofax [KV10] started with the goal to provide an integrated development environment (IDE) for SDF and Stratego. It was then developed into a language workbench for Eclipse that allows language development with editor support for both, the meta-languages and the developed language. Spoofax allows smooth switching between both editor services and allows to deploy the editor for the developed language standalone. The editor services provide syntactic and semantic analysis based on live parses, with error recovery and origin tracking. Those facilities are implemented language pragmatic, which allows developers to focus on language specific parts.

We use Spoofax in this thesis to provide editor support for the specification language and as glue between SDF and Stratego in the development of the specification language, the formula generator and the type checker generator.

2.1.4 Vampire

We use the automatic theorem prover Vampire [Vor95] to prove classical first-order logic propositions about type system specifications. Vampire is able to parse formulas in the TPTP format and provides detailed output for the conducted proofs. We use this output to extract information about the applied axioms, which are in our case typing rules. However, the choice of Vampire was rather arbitrary every first order theorem prover that has support for TPTP will do the job. Of course the results might vary depending on the performance of the theorem prover.

2.1.5 Alternatives

We chose Spoofax and its components, because its feature set fits well to our goals and there was some previous related work based on it. However there are alternative *language workbenches* that could have been a good fit as well. For example Rascal [KvdSV09] a meta-programming language that has among others support for context-free grammars, algebraic data-types, relations, relational calculus operators, advanced patterns matching, generic type-safe traversal, comprehensions, and string templates for code generation. The paper [EvdSV$^+$13] gives an overview of the state of the art in language workbenches.

2.2 Type Systems

Type theory started as an attempt by Gottlob Frege to solve Russel's paradox, which shows that naïve set theory is inconsistent. Frege argued that a predicate requires an object as argument and cannot have itself as an argument, as it is the subject. So the initial motivation for type theory was to avoid paradoxes and contradictions in logics and rewrite systems. The term *type system* refers to type theories whose rewrite systems are programming languages. Type systems address the problem to ensure that programs have meaning, whereas in type theory the problem is the ensure the consistency of a logic.

What does it mean to "ensure that programs have meaning"? It means that one wants to filter the useful programs. It is not useful to have a syntactically valid program that has no semantics as this program will behave unexpectedly at some point. A type system assigns types to the expressions of a programming language. A *type checker* can then verify, whether the types of the programs expressions match according to predefined typing rules. If the type checker succeeds the program is *well-typed* and has meaning, i.e. the program will not misbehave due to undefined semantics. Depending on the expressiveness of the type system also the correctness of the computation that the program performs can be shown.

Chapter 3

Specification Language

Our type system specification language is a DSL for people familiar with type systems or type theory. While developing it, we had the following goals in mind. The specification language should be

- close to text-book formalisms used in the type system community
- purely declarative
- modular
- usable with existing syntax definitions of programming languages

Those characteristics fit well with the goals of making it easy to experiment with type systems and to create type systems from a high-level specification.

3.1 Language Design

In this sections we introduce the architecture of the specification language with references to a small examples and argue how this architecture reflects the goals mentioned above.

Specifications in our specification language are divided into eight sections, of which two are optional. In a specification the sections have to be present in the order they are introduced here.

Module name The first section declares the name of the specification module. This name is an unique identifier and used for imports. The module name may contain an arbitrary combination of numbers, letters, and the symbols `.` (dot), `-` (dash), and `/` (slash).

Example 3. `module target-language/typesystem`

Example 3 shows the declaration of a module with the unique identifier `target-language/typesystem`.

Imports The second section declares which other type system specifications are imported and is optional. It is possible to prevent sections of a module from being imported with the keyword `hiding`.

Example 4.

```
imports some-specification hiding (language contexts)
        another-specification
```

The specification in Example 4 imports the module `some-specification` without the sections `language` and `contexts` and it imports the module `another-specification`.

Language The third section declares the language for which a type system should be specified. This language needs to be present as a SDF module located at the specified path. We call this language from now on *target language*. If other modules are imported the target language must be defined only once in the closure of the module.

Example 5.

```
language specifications/simply-typed-lambda-calculus/syntax
```

In Example 5 the SDF definition at `specifications/simply-typed-lambda-calculus/syntax.sdf` is used; This path is relative to the `syntax` folder of the Spoofax project. A SDF definition file (a file with the suffix `def`) located at the same path needs to be present for analysis of the target language, see Section 5.4.

Contexts The fourth section declares the contexts that can be used in the judgments and rules. In specifications contexts are used to track or collect information during type checking. Informally these contexts are declared as cross-products of non-terminals, i.e. a context instance is a set consisting of tuples of terminals that can be produced by the non-terminals in the declaration. Every context has a name, which can be used as a non-terminal in the specification.

Contexts serve in most use cases as bindings for variables. Therefore it is necessary to be able to look up terminals in a context instance. In the context declaration each non-terminal is annotated with an input (`{I}`) and output (`{O}`) tag. Those tags specify the key and value positions of the context declaration. Depending on those annotations we generate look up functions. If all non-terminals are tagged as inputs the generated look up function will perform a normal membership test. However, a context declaration with only output tags is not useful as it is not possible to add elements to it.

Given a context declaration `Z := A{I} x B{I} x C{O} x D{O}` an instance of this context looks like `(a : b : c : d) ; z` where `a`, `b`, `c`, and `d` are terminals produced from the non-terminals `A`, `B`, `C`, `D`, respectively. `z` is another instance of context `Z` or the empty context, which is written `()`. Different empty contexts can be disambiguate with the context name, e.g. `(Z)`.

Example 6. `contexts Binding := ID{I} x Type{O}`

Example 6 shows a context called `Binding` that consists of a non-terminal `ID` tagged as input and a non-terminal `Type` tagged as output. This contexts models a type binding for identifiers. We generate from this context declaration a look up function of the form `i : t in c` where `i` represents a terminal produced from `ID`, `t` a terminal produced from `Type` and `c` represents an instance of context `Binding`.

Meta-variables The fifth section declares meta-variables. In specifications meta-variables are used to refer to expressions of the target language and to contexts. The declaration of a meta-variable consists of a class, a prefix, and a set of non-terminals from the target language and from the specification. We use the class to distinguish different kinds of meta-variables. Currently we use classes only for merging meta-variable declarations when resolving imports. For details see Section 3.3.

The set of non-terminals defines the scope of a meta-variable. Every non-terminal contained in this set is extended with productions for meta-variables. In other words, a meta-variable is a substitute for every terminal that can be produced from one of the non-terminals. We explain the extension of the target language in detail in Section 3.3.

Syntactically a meta-variable is a string of numbers and letters that is prefixed with the prefix of the meta-variable declaration. The sole purpose of the prefix is to avoid syntactic ambiguities. The prefix itself can consist of numbers, letters and the following symbols `~`, `$`, `%`, `&`, and `?`. There is currently no way to add new symbols to that list, besides editing the syntax definition of the specification language. Usually a prefix is chosen such that it is not a prefix of a construct of the target language, to reduce the chance of encountering ambiguities.

Example 7.

```
meta-variables  Term "~" { Type Exp }
                Ctx "$" { Context }
                Id "%" { ID }
```

Example 7 declares three classes of meta-variables. The first class is called `Term` and each meta-variable of this class has the prefix `~`. Productions for meta-variables of this class are added to the non-terminals `Type` and `Exp`.

Judgments The sixth section of a type system specification module declares judgments. In our specification language judgments are the basic building blocks of the rules defined in the next section. This is common in the specification of deduction systems in general. Judgments can be thought of as the “syntax” of the type system, the semantics is defined by the rules.

A judgment can be defined rather arbitrarily from a combination of strings of letters and numbers, non-terminals of the target language and the names of the contexts. Those can be mixed freely, as long as a string separates the non-terminals and context names. This restriction is only needed to reduce the number of syntactic ambiguities in the language. Judgments do not have a name as it is currently not possible to refer to them in any other way than in instances of them. To separate judgment declarations from each other, each declaration must be finalized with a dot.

Non-terminals of the target language and context names need to be annotated with input/output-tags in the same way as for contexts. Those tags describe which parts of the judgment need to be computed by rule applications and which are provided as input parameters to the rule application. Non-terminals of contexts currently only support the input tag.

Equality and inequality are predefined built-ins, but have to be introduced as judgments. This enables to define precisely for which non-terminals equality/inequality should be available and prevents other uses at parse time. Currently it is not possible to define (in-)equalities between contexts. Equality and inequality judgments are defined by appending `is Eq` or respectively `is Neq` to the judgment declaration.

Example 8.

```
judgments Context{I} "|-" Exp{I} ":" Type{O}.
          Type{I} "<:" Type{I}.
          Exp{I} "=" Exp{I} is Eq.
```

Example 8 shows three judgments. The first could be the typing judgment of a variant of the simply typed lambda calculus with a context, an expression of the target language as input and a type of the target language as output. The second judgment defines a relation between the types of the target language, i.e. it has only input positions. This judgment could represent a subtyping relation. The last judgment declares an equality between expressions.

Rules The seventh section of the module declares the (typing) rules. These rules define the semantics for the judgments declared in the previous section. The syntax of the rules replicates the form of inference rules: A (possibly empty) list of premises separated from a conclusion by a horizontal line. Premises and conclusion are instantiated judgments. All meta-variables that occur free in rules are implicitly all-quantified. This means that all meta-variables are all-quantified, as there is currently no mechanism to bind variables in typing rules.

Rules can be annotated with a name. Rule names increase the readability of the specification and allow to create human readable derivation traces. The rules have also support for custom error messages. There are two kinds of error annotations. Premises can be annotated with `@error msg`, where `msg` can contain meta-variables and holes (written `{}`) interleaved with arbitrary strings. Those errors are thrown if the premise could not be derived or if the calculated output does not match the expected output. The meta-variables are instantiated with the appropriate terms and the hole with the expected output. The other kind of errors are prefixed with `@implicit` and are thrown if an implicit equality between two meta-variables cannot be satisfied. The meta-variables of implicit equalities are distinguished in the error message by `@[number]` annotations, where `[number]` is a natural number. The conclusion can also be annotated with error messages for implicit equalities. Section 5.5 and Section 5.6 explain how error messages are implemented in the type checker generator. Note that error messages are always attached to the preceding premise.

Example 9.

```
judgments Context{I} "|-" Exp{I} ":" Type{O}.

rules

%x : ~T in $C @error %x "should have type" ~T "but has" {}.
============= T-var
$C |- %x : ~T

(%x : ~T1 ; $C) |- ~e : ~T2
@error ~e "should have type" ~T2 "but has" {}.
===================================== T-abs
$C |- fun %x : ~T1 (~e) : ~T1 -> ~T2

$C |- ~e1 : ~T1 -> ~T2
$C |- ~e2 : ~T1 @error ~e2 "should have type" ~T1 "but has" {}.
==================== T-app
$C |- ~e1 ~e2 : ~T2
```

Example 9 shows the typing rules of Programming Computable Functions (PCF) for variables (`T-var`), function abstraction (`T-abs`), and function application (`T-app`). Rule `T-var` models that some variable `%x` has type `~T` if it has type `~T` in context `$C`.If this check fails the annotated error message is thrown, where `~T` is replaced by the expected type and `{}` by the actual type of variable `%x`.

Rule `T-abs` in Example 9 expresses that function with argument `%x` of type `~T1` and a function body `~e` has type `~T1 -> ~T2` if the function body has type `~T2` under a context that is extended with the function argument. Function application is covered by rule `T-app`: if `~e1` is a function of type `~T1 -> ~T2` and the type `~e2` of matches its argument, then the application of `~e2` to `~e1` has type `~T2`.

Example 10.

```
judgments
Context{I} "|-" Exp{I} ":" Type{O}.

rules

====================== Subst-Eq
~S = [ %x -> ~S ] %x@1    @implicit %x "does not equal" %x@1.
```

Example 10 shows the rule `Subst-Eq` from the SystemF specification in Appendix A.1. `Subst-Eq` has no premises and an implicit equality annotated with an error message in the conclusion. Here `%x` and `%x@1` refer to the same meta-variable. The annotation `@1` is used to distinguish the variables in the error message.

Conjectures Tests for a specification are called *conjectures*. Their syntax is similar to the syntax of rules, with two exceptions. It is not possible to annotate premises or conclusions with error messages and a conjecture can be marked as *not derivable* by prepending the separating line with a slash. Marking conjectures not derivable allows to formulate negative tests.

Example 11.

```
============================
() |- let fac : int -> int =
  fix f : int -> int (
    fun n : int (
      ifz n then 1
      else n * (f (n - 1))
    )
  )
 in (fac 3) : int

/===========================
() |- fun x : int (x) : int
```

Example 11 shows two conjectures of the PCF implementation. They use the judgment shown in Example 8. The first conjecture asserts that the type of the faculty function applied to `3` is `int` and the second conjecture asserts that the identity function for integers (`fun x : int (x)`) has type `int` is not derivable.

Comments Line comments (`//`) and block comments (`/* ... */`) can be inserted everywhere in the module.

3.2 Design Assessment

How does this design reflect the characteristics from the beginning of this chapter? We will address this point by point.

Usage Formal definitions of type systems usually consist of judgments, rules, and auxiliary definitions for contexts. All those components can be represented in a natural way in the specification language. Judgments can be defined as an arbitrary combination of non-terminals and separation symbols, as long as the syntax does not create ambiguities. Rules are written in a natural deduction style, because this is the most common formal representation of typing rules and it acknowledges the deductive nature of typing rules in general. Contexts can be defined using a set like notation. This is close to the intuitive semantics of contexts and allows to generate commonly used syntax.

Declarative Nothing that can be defined in the specification language has side effects (i.e. breaks referential transparency) or possibilities to embed executable code. Therefore the specification language is purely declarative, it focus on *what* should be done rather then *how* it should be done.

Modularity Type system specifications are organized in composeable modules. A module can import other modules or even only parts of other modules. This enables the reuse of existing type specifications and the separation of orthogonal features.

Integration Type systems can be defined for every programming language for which a SDF syntax definition exists. These definitions can be used in most cases without further modifications. One of the requirements is that they introduce a constructor for every context-free production. Modifications are also needed if one wants to use a language concept inductively in the type system, but has not implemented this concept with explicit induction in the syntax. The implementation of records in the simply typed lambda calculus in Appendix A.2 is an example for this.

3.3 Implementation

In this section we describe how the specification language is implemented and how the new syntactic constructs that can be defined in a specification are integrated.

The specification language itself is defined in SDF and consists of four SDF modules. The module `Common` containts lexical definitions that are used in multiple modules, the module `BaseLanguage` which defines the syntax of the specification language, the module `Generated` which contains new syntax for meta-variables, contexts, and jugmens that was defined in a specification and the main module `SLTC` that combines the previous modules.

The module `Common` only defines character classes and lexical restrictions, therefore we will not explain it in detail.

The syntax of the specification language, which is independent of the target language, is defined in the module `BaseLanguage`, and is parameterized by the non-terminals `TypingJudgment` and `MetaVariable`. These non-terminals depend on the actual target language and are therefore defined in `Generated`.

In a new project the module `Generated` does not exist in the syntax folder, as no specification is in use. Building a project copies a dummy `Generated` module from the resource folder into the syntax folder. The dummy `Generated` module contains empty productions for the non-terminals `TypingJudgment` and `MetaVariable` to ensure that a compilation of the whole project is possible without a specification (e.g. to run tests). We describe the generation of a specification specific `Generated` module in the following.

The strategy `toSdf` transforms a specification into a SDF AST which is then pretty printed and saved in the syntax folder. The generated module `Generated` imports module `Common` and the SDF file of the target language. In addition it contains context-free grammars for context, meta-variables, and judgment declarations.

For each context declaration the strategy `make-contexts` generates productions for the empty context, context bindings, and context lookups. Figure 3.1 shows the resulting productions, where *Name* is the name of the context declaration and `n` the position of the context declaration in the specification.

Figure 3.2 shows the productions created by `make-variable` for meta-variables. A meta-variable consists of its prefix, a name and an optional error annotation. There are two kinds of productions for meta-variables. The first kind extends every non-terminal (in the production called *Scope*) listed in the meta-variable definition of the target language. The second kind (the non-terminal *MetaVariable*) enables the use of meta-variables in error messages. In the constructor of meta-variables `m` is either the context number in case *Scope* is a context and otherwise the meta-variable class.

Strategy `make-judgment` creates productions for judgment declarations. For each judgment we generate a production that consists of all non-terminals separated

$\langle \mathit{Name} \rangle$ ::= '`ContextEmpty-n`'
  | '`ContextBind-n`' $\langle \mathit{Elem} \rangle$ $\langle \mathit{Name} \rangle$
  | '`ContextLookup-n`' $\langle \mathit{Elem} \rangle$ $\langle \mathit{Name} \rangle$
  | '`(`' $\langle \mathit{Name} \rangle$ '`)`'

$\langle \mathit{Elem} \rangle$ ::= $\langle \mathit{String} \rangle$ | $\langle \mathit{String} \rangle$ '`:`' $\langle \mathit{Elem} \rangle$

Figure 3.1: Context productions

$\langle \mathit{Scope} \rangle$ ::= '`MetaVariable-m`' $\langle \mathit{Prefix} \rangle$ $\langle \mathit{MetaVariableName} \rangle$ $\langle \mathit{Anno} \rangle$

$\langle \mathit{MetaVariable} \rangle$ ::= '`MetaVariable-m`' $\langle \mathit{Prefix} \rangle$ $\langle \mathit{MetaVariableName} \rangle$ $\langle \mathit{Anno} \rangle$

$\langle \mathit{Anno} \rangle$ ::= ϵ | '`@`' $\langle \mathit{ErrorNumber} \rangle$

Figure 3.2: Meta-variable productions

by the separators from the declaration. The constructor of judgments is either composed from the string `TypingJudgment` or in the case of a built-in from the built-in name and the position of the judgment in the specification.

The module `SLTC` plugs all modules together. It imports the module `Generated` and instantiates the parameters of module `BaseLanguage`.

Imports are implemented using the Name Binding Language (NaBL) that is integrated in Spoofax. NaBL ensures that the module names are unique, that imports can be resolved and annotates modules with meta-information about modules. The Stratego strategy `resolve-imports` does the actual resolving of the imports before a module gets used.

First `resolve-imports` fetches all module definitions that are imported. Then it merges the fetched modules into the current module. This results in a module that contains the declarations of the current module plus all not excluded declarations from the imported modules. All sections are merged separately and redundancies such as duplicate declarations are removed.

Chapter 4

Formula Generation

Type systems in programming languages are treated as black boxes from a programmers point of view. The programmer interacts with the type system for example via type annotations and receives feedback from the type system in form of error messages. Errors in the type system are hard to detect for the programmer, because he cannot be sure if it is an error in his program or in the type system (or its implementation). In addition he can only debug his program, because of the black box view on the type system. Therefore it is desirable to ensure that the type system has the intended semantics.

Mathematical proofs are used to ensure that a type system has the intended semantics. A basic property of type systems is safety, which roughly means "a well-typed term can never reach a stuck state during evaluation"[Pie02]. Those properties can be proven by hand or with the help of proof assistants. Both methods require substantial manual effort. In accordance to the goal of the automated generation of type checkers, we explore how well automated theorem provers can solve simple propositions and how this can be exploited in the type checker generation. We generate formulas from type system specifications to interface with automated theorem provers.

4.1 Goals

The generation of formulas from type system specifications pursues two goals. The first goal is to represent type system specifications as formulas suitable for automated theorem provers and to use those as a basis to prove simple propositions about type systems. In Section 5.4 we prove propositions that check the applicability of an optimization. The second goal is to explore how well automated theorem provers can check if a program is well-typed. Being able to type check programs using automated theorem provers provides us with free reference implementations for our type checker generator. The correctness of those reference implementations depends only on the translation of type system specifications into first-order formulas and on the correct implementation of the automated theorem provers.

4.2 Translations

In this section we explain how we translate type system specifications into first-order formula. Just context declarations, rules, and conjectures have an explicit representation as first-order formulas. The rest of the specification is not directly translated into first-order formulas and it is only needed to ensure that the generated formulas are well formed.

We represent contexts as a list like structure. Every context c has a constant empty_c that represents the empty context and predicate bind_c that constructs contexts from the inputs and outputs of the context declaration and a context. The lookup of elements in a context c is modeled by the predicate lookup_c. This predicate is defined inductively by two formulas shown in the following and checks whether an element is contained in a context. In the base case the element to look up is the top element of the context. In the step case the top element is different from the element we search, therefore we have to proceed with the rest of the context.

$$\begin{aligned}&\forall e, x_1, \dots, x_n, y_1, \dots, y_m.\\ &\qquad(\mathit{lookup}_c(x_1, \dots, x_n, y_1, \dots, y_m, \mathit{bind}_c(x_1, \dots, x_n, y_1, \dots, y_m, e)))\end{aligned} \tag{4.1}$$

$$\begin{aligned}&\forall e, x_1, \dots, x_n, x'_1, \dots, x'_n, y_1, \dots, y_m, y'_1, \dots, y'_m.\\ &\quad(x_1 \neq x'_1) \wedge \cdots \wedge (x_n \neq x'_n) \wedge (\mathit{lookup}_c(x_1, \dots, x_n, y_1, \dots, y_m, e) \implies\\ &\qquad\mathit{lookup}_c(x_1, \dots, x_n, y_1, \dots, y_m, \mathit{bind}_c(x'_1, \dots, x'_n, y'_1, \dots, y'_m, e))\end{aligned} \tag{4.2}$$

Formula 4.1 shows the base case of the lookup in context c. We translate all non-terminals in the context declaration tagged as input into variables $x_1 \dots x_n$ and all tagged as output into variables $y_1 \dots y_m$, respectively. The variable e represents context. In the base case we apply lookup_c to a context whose outer most bind_c binds an element that is exactly the element we search. Therefore the element is trivially contained in the context and the lookup succeeds.

Formula 4.2 models the step case of the lookup in context c. We introduce two variables, $x_1, x'_1, \dots, x_n, x'_n$ for non-terminals tagged as input and $y_1, y'_1, \dots, y_m, y'_m$ for non-terminals tagged as output. The intuition of the formula is that if it is possible to look up an input/output pair in a context e then we can also look it up in a context that contains an additional input/output pair. If read with evaluation in mind the intuition is that if the outermost element is not the element we look for, i.e. all inputs differ, then we have to look into the rest of the context. We only check the input positions in order to test whether we have a match to lookup the last element first. This corresponds to the scoping behavior of most programming languages.

Example 12. The following context declaration models a standard identifier to type binding. It has two components, identifier as inputs and types as outputs.

```
contexts C := ID{I} x Type{O}
```

For this declaration we generate the following two formulas to model the behavior of lookup_C.

$$\forall i, t, e\,.\,(\mathit{lookup}_C(i, t, \mathit{bind}(i, t, e))) \tag{4.3}$$

$$\forall i, i', t, t', e\,.\,(\mathit{lookup}_C(i, t, e) \wedge i \neq i' \implies \mathit{lookup}_C(i, t, \mathit{bind}_C(i', t', e)))) \tag{4.4}$$

We translate the AST nodes of the programming language directly into predicates that resemble the program structure. For the translation we use the following scheme. AST nodes of the following form $Cons(e_1, \dots, e_n)$ are translated into a predicate of the form $cons(p_1, \dots, p_n)$, where we translate all constituents e_i into predicates p_i recursively. We create for each of those predicates the following injectivity and univalence axioms.

$$\begin{aligned}&\forall p_1, p_1', \dots, p_n, p_n' \,. \\ &\quad (cons(p_1, \dots, p_n) = cons(p_1', \dots, p_n') \implies (p_1 = p_1') \wedge \dots \wedge (p_n = p_n'))\end{aligned} \tag{4.5}$$

$$\begin{aligned}&\forall p_1, p_1', \dots, p_n, p_n' \,. \\ &\quad ((p_1 \neq p_1') \vee \dots \vee (p_n \neq p_n') \implies cons(p_1, \dots, p_n) \neq cons(p_1', \dots, p_n'))\end{aligned} \tag{4.6}$$

$$\begin{aligned}&\forall x, y, p_1, \dots, p_n \,. \\ &\qquad ((cons(p_1, \dots, p_n) = x \wedge cons(p_1, \dots, p_n) = y) \implies x = y)\end{aligned} \tag{4.7}$$

Injectivity and univalence holds by definition for those predicates as we create those predicates from syntax. These axiom can help theorem provers to conduct proofs, for examples see Section 5.4.

The most important part is the translation of the typing rules. Depending on whether the typing rule has premises we use either of the following schema:

$$\forall FV(c) \,.\, c \tag{4.8}$$

$$\forall FV(p_1, \dots, p_n, c) \,.\, p_1 \wedge \dots \wedge p_n \implies c \tag{4.9}$$

The predicates p_i represent the premises and c is the conclusion of the typing rules, FV computes the free variables. What a typing rule intuitively expresses is that the conclusion can be derived if all premises can be derived. In terms of first-order logic "derived" means that there exists a proof for the proposition. Therefore we translate a typing rule without premises into a formula that consists of the conclusion and all-quantifies all free variables of the conclusion. Free variables are all-quantified, because we want that all possible variants of the conclusion are derivable. Typing rules with premises translate into a single implication. The premise of the implication is the conjunction of all premises of the typing rule. This ensures that all premises need to be derivable/satisfied. The conclusion of the implication is the conclusion of the typing rule. This is a safe fact, because the conclusion of the typing rule is derivable if all premises are derivable, which is exactly the semantics of this implication.

Example 13. The following we have the typing rules `T-var` and `T-abs` from the PCF specification.

```
%x : ~T in $C          (%x : ~T1 ; $C) |- ~e : ~T2
============= T-var    =================================== T-abs
$C |- %x : ~T          $C |- fun %x : ~T1 (~e) : ~T1 -> ~T2
```

Those are translated into the following first-order formulas.

$$\forall x, t, e\,.\,(\mathit{lookup}(x, t, e) \implies \mathit{tcheck}(e, \mathit{var}(x), t))) \tag{4.10}$$

$$\begin{aligned}\forall c, x, e, t, s\,.\,(&\mathit{tcheck}(\mathit{bind}(x, t, c), e, s)\\ &\implies \mathit{tcheck}(c, \mathit{fun}(\mathit{param}(x, t), e), \mathit{funtype}(t, s)))\end{aligned} \tag{4.11}$$

We translate judgments in rules into predicates. The built-in judgments for equality and inequality are translated into the primitives of TPTP.

4.3 Implementation

This section describes the implementation details of the translation from the specification language into first-order formulas.

The implementation is organized in the following steps. At fist, the module is split up into its components, then we transform contexts, typing rules, and conjectures into first-order formulas. After that we create a file for each conjecture which contains all generated formulas. In the following those steps will be explained in detail.

The strategy `make-context-formulas` generates axiom formulas for each context declaration. For each distinct non-terminal in a context definition we create a fresh variable name to ensure variable names are compatible with TPTP. We transform every context declaration into Formula 4.1 and Formula 4.2 in our internal representation of TPTP formulas. To adhere to the structure of the two formulas, we split the non-terminals of a context declaration by its input/output tags and put those tagged as input before those tagged as outputs. In the generated formulas we replace the non-terminals by the fresh variables. In case of Formula 4.2 we create additional fresh variables for input positions.

To rewrite typing rules into first-order formulas is a bit more involved than the rewriting of context declarations. The strategy `make-formula` transforms rules into first-order formulas in the TPTP format. This strategy rewrites premises and conclusions into first-order terms using the strategy `rewrite`. Premises are, due to limitations of the `layout`-rules, not represented as ordinary lists after parsing. Therefore premises are transformed into ordinary lists by the generic `to-list` strategy. Now that we have first-order terms for the premises and the conclusion, we collect all free variables that occur in them. In rules variables are not bound, thus all variables are free. However it is important to collect only free variables, as conjectures may contain quantifiers. At last, we put premises, conjecture, and quantification together to construct a formula in the TPTP format which has the structure of Formula 4.9.

Now we present the details of the `rewrite` strategy, which transforms premises and conclusions. This strategy is defined as a sequence of two top down traversals. The first traversal is the actual rewrite of the typing judgments and the second is a special treatment for strings. The second traversal is necessary to wrap all target language constructs into the `Term` constructor.

We translate parts of the specification language into new SDF syntax definitions as described in Section 3.3. Therefore not all nodes contained in the AST of a specification are known at the time of implementation. However all those nodes have a regular structure. That is why it is possible to use the `cons#(args)` pattern to extract the relevant parts in a generic manner. The rule `make-aux-cons` wraps all generic nodes[1] into the auxiliary constructor `AuxCons` with three parameters, the static part of the constructor name, the generic part of the constructor name and the arguments. After this transformation normal pattern matching on `AuxCons` is possible. The strategy `rewrite-aux-cons` transforms those auxiliary constructors into first-order terms.

All nodes in the AST of a specification that are not wrapped into auxiliary constructors are constructors of the target language. Therefore we attempt to transform each node into an auxiliary node, in case that succeeds, we rewrite the node into a first-order term otherwise we wrap the node into the `Term` constructor. It is important to wrap the nodes of the target language to implement pretty printing.

The second top down traversal wraps all strings that occur in the parameters of nodes into `Term` constructors. As the specification language has no string nodes, it is safe to transform all those strings into `Term` nodes. If there is no second traversal a direct transformation leads to infinite recursion as every string within a `Term` constructor would be wrapped again in a `Term` constructor, therefore we need a second traversal.

Conjectures are essentially transformed following the same scheme as rules, the only difference is that we tag the resulting formulas as `conjecture` and not as `axiom`.

4.4 Editor Support

It is possible to use automated theorem provers to type check conjectures from within the editor. For type checking with automated theorem provers we transform the specification into first order formulas and create a file per conclusion, as described in the previous section. Then we call the automated theorem prover Vampire on each resulting file and parse the results. For each conjecture we visualize in the editor whether the verification succeeded and in case of success provide the names of the used first-order formulas.

We contribute also a consistency test for specifications. This check attempts a proof of `1 = 0` with vampire using the first-order formulas generated from the specification. This method allows us to possibly find inconsistencies in the type system specification. However, due to the embedding into first-order logic and Gödel's second incompleteness theorem it is not possible to proof within first-order logic that the type system is consistent.

[1]Generic nodes of the specification language, i.e empty context, context binds, context lookups, typing judgment and meta-variable nodes.

Chapter 5

Type Checker Generation

In this chapter we briefly describe and motivate the design and goals of the type checker generator.

5.1 Goals

The first goal is to generate type checkers from high level type system specifications that are not geared to type checking. Mainly this means to try to deal with non-syntax directed typing rules without backtracking. The motivation for this is that non-syntax directed typing rules are often more readable and that changes have mainly local effects.

The second goal is to design a modular type checker generator, particularly one that can be easily adapted and that facilitates the exchange of components. This modularity is desirable because it increases the reusability and makes it possible to combine previously unrelated projects.

The third goal is to generate type checkers that emit readable error messages if a program is not well-typed. This is essential to make the generated type checker usable in production in any way.

5.2 Architecture

The type checker generator has two phases: Template generation and template optimization. The template generation phase transforms the type system specification with modifications into templates. A template is a different representation for a typing rule that is better suited for type checking. We introduce templates in detail in Section 5.3. The template optimization phase checks which optimizations apply to the generated templates and applies them. The optimizations aim to reduce the amount of non-determinism in the type system and therefore reduce the amount of backtracking in the type checker. The final product after those two phases is a file that contains the optimized templates.

The template optimization phase is the key part of the type checker generator. It is the link between the type checker generator on the one hand and the formula generation and automated theorem provers on the other hand. We generate from a single type system specification a first-order formula representation suitable for theorem proving and a template representation suitable for constraint generation. In the optimization phase we combine them by using the automated theorem prover to check if optimizations are applicable to the templates.

Instead of generating a type checker directly from the specification, we implement a generic constraint-based type checker that takes templates as input. This generic type checker has two phases: Constraint generation and constraint solving. The constraint generation phase takes the templates and the expression that shall be type checked as input. We then build a derivation tree according to the expression by pattern matching the conclusion of the templates. The building and traversing of the derivation tree emits constraints. In the constraint solving phase we then try to unify the emitted constraints. If that succeeds it reports the result otherwise it reports the errors that occurred during unification.

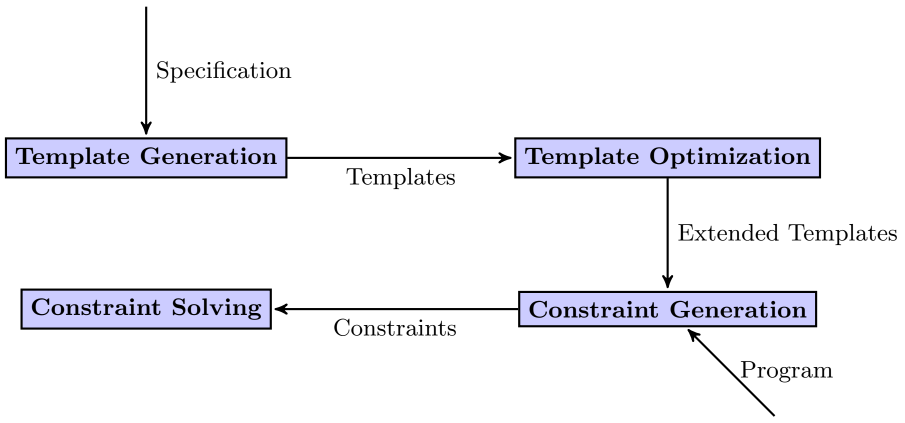

Figure 5.1: Phases of the type checker generator

Figure 5.1 shows the phases and their relationships. Nodes represent phases and arrows express that data flows from one node to another. Labels on arrows describe the data format.

The four phases correspond to modules or tools. Each phase has a well-defined interface, therefore the implementation can be exchanged freely. This facilitates the use of different constraint solvers, constraint generators or template optimizers.

The following sections describe the implementation of the type checker generator. Each section focuses on one of the phases.

5.3 Template Generation

The first phase translates type system specifications into templates. This phase is not just a simple translation from a human readable representation into a representation suitable for programs, but also normalizes the resulting templates. Normalization comprises the elimination of implicit equalities and resolves dependencies between premises. All following phases assume normalized templates. The normalization allows simplifications in the implementations of the phases.

Definition 5.1. The Stratego implementation appends to all constructors two underscores to avoid name collisions with target languages. This is necessary as the module system of Stratego is not strong enough to avoid those collisions. We omit these here for the sake of readability. The non-terminals Inputs, Outputs, Term, and Error may contain arbitrary terms.

$\langle \textit{Template} \rangle$ ::= ‘`Template`’ $\langle \textit{Premises} \rangle$ $\langle \textit{Conjecture} \rangle$

$\langle \textit{Conclusion} \rangle$ ::= ‘`Conclusion`’ $\langle \textit{Judg} \rangle$ $\langle \textit{Name} \rangle$ $\langle \textit{Pattern} \rangle$ $\langle \textit{Outputs} \rangle$

$\langle \textit{Premises} \rangle$ ::= ϵ | $\langle \textit{Premise} \rangle$ $\langle \textit{Dependencies} \rangle$ $\langle \textit{Premises} \rangle$

$\langle \textit{Premise} \rangle$ ::= ‘`Lookup`’ $\langle \textit{Ctx} \rangle$ $\langle \textit{Inputs} \rangle$ $\langle \textit{Outputs} \rangle$ $\langle \textit{Error} \rangle$
 | ‘`Judgment`’ $\langle \textit{Judg} \rangle$ $\langle \textit{Inputs} \rangle$ $\langle \textit{Binding} \rangle$ $\langle \textit{Outputs} \rangle$ $\langle \textit{Error} \rangle$
 | ‘`Eq`’ $\langle \textit{Term} \rangle$ $\langle \textit{Term} \rangle$ $\langle \textit{Error} \rangle$
 | ‘`Neq`’ $\langle \textit{Term} \rangle$ $\langle \textit{Term} \rangle$ $\langle \textit{Error} \rangle$

$\langle \textit{Dependencies} \rangle$ ::= ϵ | $\langle \textit{Judg} \rangle$ $\langle \textit{Outputs} \rangle$ $\langle \textit{Dependencies} \rangle$

$\langle \textit{Name} \rangle$ ::= ‘`Some`’ $\langle \textit{String} \rangle$ | ‘`None`’

$\langle \textit{Judg} \rangle$ ::= $\langle \textit{Int} \rangle$

5.3.1 Templates

A template is an intermediate representation of a typing rule that is more suitable for the constraint generation process and defined in Definition 5.1.

Templates serve as the input format for the constraint generation phase of the type checker. Besides being better suited for constraint generation templates also have the advantage that we can adapt the system to other specification languages by translation into templates. The structure of the templates has no hard requirements on the specification language, although it has to be declarative and the structure of the templates is similar to the structure of the typing rules of the specification language.

Before we are going to describe the structure of templates in depth, we highlight the conceptual differences between templates and typing rules. Templates take advantage of the input/output tags of typing judgments and context definitions by splitting everything up into an input and output part. This will become handy in the constraint generation phase, where we can see immediately which are the patterns to match against the expressions and which are the output positions that we have to compute. As described in the introduction to this chapter, we also normalize templates. For each implicit equality in the conclusion and premises of typing rules we add an explicit equality. Thus every variable occurs in each judgment only once. Further we sort premises by dependencies between their inputs/outputs. We describe and motivate the normalization process in the remainder of the section. The template optimization and constraint generation phase take advantage of these normalizations and therefore expect their inputs to be normalized.

Note: We transform all meta-variables of the specification into the variables (`Var(name)`) that we use for constraint generation as we do not need the additional information of meta-variables anymore.

Premises in Templates

Premises in templates have multiple shapes: Judgments, context lookups, equalities, and inequalities. A premise always has a (possibly empty) list of dependencies. This list contains the number and the outputs of the judgment the premise depends on. Later on in this section we define what it means for a premise to have dependencies, first we take a look at the different kinds of premises.

Judgments correspond to the user defined judgments of the specification. They consist of the judgment number $\boxed{1}$ which refers to the position in the declaration section of the specification, the positions of the judgment marked as inputs $\boxed{2}$, context modifications $\boxed{3}$, $\boxed{4}$, $\boxed{5}$, the output positions of the judgments $\boxed{6}$, and potentially error messages $\boxed{7}$.

The main difference between context modifications in templates and typing rules is that all modifications are at one place and can always be evaluated in the same manner. A context modification can either be an addition to a context $\boxed{3}$ which consists of a context identifier (this identifier refers to the position in the declaration section of the specification), a list of inputs and a list of outputs. It can also be a context identity $\boxed{5}$ which corresponds to a context meta-variable in the specification and context identities do not modify the context and a reset operation $\boxed{4}$ which resets the given context and corresponds to the empty context. A valid context can be obtained by applying the context modifications from right to left to a context instance. A detail explanation of the semantics in our type checker will follow in Section 5.5.

Example 14.

```
Judgment(
          1 $\boxed{1}$
        , [Var("X0")] $\boxed{2}$
        , [ Binding(1, [Var("X1")], [Var("X2")]) $\boxed{3}$
          , Reset(1) $\boxed{4}$
          , Ctx(2) $\boxed{5}$
          ]
        , [Var("X3")] $\boxed{6}$
        , Some([Error([Var("X0"), "has type", "{}"])]) $\boxed{7}$
        )
```

In example 14 we have judgment one that has only one input position `Var("X0")` and one output position `Var("X3")`. It leaves context two as it is, resets context one and then adds the input/output pair `Var("X1")` and `Var("X2")` to context one. In addition it is annotate with one error message.

Context lookups consist of the context number $\boxed{8}$, of the input $\boxed{9}$ and of the output $\boxed{10}$ positions of the context lookup from the specification and potentially of error messages $\boxed{11}$. We do not treat lookups as normal judgments as their semantics is not defined within the specification. Therefore we have to deal with them separately in the type checker.

Example 15.

```
Lookup(
        1 [8]
      , [Var("X0")] [9]
      , [Var("X1")] [10]
      , None() [11]
      )
```

In example 15 we look up `Var("X0")` from context one and bind the result to `Var("X1)"`. This context lookup has no error messages attached.

(In)equalities are predefined judgments and therefore treated separately. They have a judgment number, exactly two input positions, no output positions and potentially an error message. The judgment number is used to keep track of the non-terminals of the (in)equality.

Example 16. `Neq(4, Var("X0"), Var("X1"), None())`

In example 16 we test if `Var("X0")` and `Var("X1")` are not equal and provide no error message.

In some cases it is relevant in which order we evaluate premises. As we have not talked about evaluation yet, we assume premises are evaluated like functions. We provide terms for input positions an retrieve terms for the output positions. If we take a look at Example 17 we see that typing rule `T-Tapp` from Appendix A.1 has two premises. When we compare the premises with the judgment definitions, we see that `%x` occurs as an input of the first premise and as within an output position of the second premise, but not at all in the conclusion. If we want to evaluate the first premise we have to find a term for `%x`, but that term is only provided by the output of the second premise. Therefore we have to evaluate the second premise first.

Example 17.

```
judgments
TermBinding{I} "|" TypeBinding{I} "|-" Exp{I} ":" Type{O}.
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.

rules
~U = [ %x -> ~S ] ~T
$C1 | $C2 |- ~e : all %x . ~T
============================== T-Tapp
$C1 | $C2 |- ~e [ ~S ] : ~U
```

We make those dependencies visible in the template language by annotating each premise with a list of its dependencies. Those dependencies contain the number of the premise on which the premise depends and the output positions of that premise. The outputs are redundant information and only added to make it easier to check which information are proivded by that dependency. The dependency for the first premise in Example 17 would look like `(1,[TAll(Var("X0"), Var("X1"))])`. Here

`Var("X0")` and `Var("X1")` are the generated variable names for `%x` and `~T`. In this example `TAll(Var("X0"), Var("X1"))` is the only output provided by the premise and `1` refers to the second premise, because we sort premises of templates topological according to the dependencies, as we will describe in the next section.

Conclusions in Templates

Conclusions contain the judgment identifier $\boxed{\textbf{12}}$, the input positions of the judgment $\boxed{\textbf{13}}$, the context modifications $\boxed{\textbf{14}}$ as well as the output positions of the conclusion $\boxed{\textbf{15}}$. In contrast to the conclusion in a typing rule from the specification, a typing rule in a template has no error message. As it was only possible to annotate the conclusion with error messages for implicit equalities those error messages propagate into the premisses that are introduced to make the implicit equalities explicit.

Example 18.

```
Conclusion(
      3 $\boxed{12}$
    , ( [Var("X177")] $\boxed{13}$
      , [ Binding(2, [Var("X178")], [])
        , Ctx(2)
        ] $\boxed{14}$
      )
    , [] $\boxed{15}$
    )
```

Example 18 shows a conclusion that has judgment number three and only one input position. The context pattern specifies that there has to be at least one element in the second context. In this example judgment three has no outputs.

A template consists of a name, a list of premisses with dependencies and a conclusion. Figure 5.2 shows how a complete template looks like for the variable typing rule in the simply typed lambda calculus. On the left side of Figure 5.2 the typing rule from the specification is shown and on the right side its representation as a template. Note that in the example we hide the underscores in the constructor names of the specification language. Due to that we see in Figure 5.2 the term `Var (Var("X52"))` where actually the outermost `Var` constructor is part of the target language and the inner `Var` is part of the specification language.

5.3.2 Generation

The Stratego rule `to-template` does the main part of the template generation. It takes a rule from a specification and transforms that rule into a template. The strategy `to-templates` does that iteratively for every rule in a specification. The first step in the conversion is the elimination of implicit equalities in the premises and the conclusion.

For each implicit equality we create an explicit equality by collecting all variables that occur more than once in an input position of a premise or conclusion. After that we create fresh names for the collect variables and create premises that state the equality of the fresh meta-variables. Implicit equalities in premises are not transformed into explicit equalities if the meta-variables also occur in the conclusion. These equalities are either ensured by explicit equalities from the conclusion or if there are no implicit equalities in the conclusion for this meta-variable, by the fact that this variable occurs only once as a source and thus, cannot introduce different values. If a meta-variable occurs more than twice we define the equalities for the new variables transitively.

Example 19 shows this for a typing rule of the judgment

```
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}
```

which models type substitution in the SystemF, see Appendix A.1. On the left side of Example 19 you see the typing rule with an implicit equality and on the right hand side the transformed version without the implicit equality.

Example 19.

```
                                  %y = %z
======================            ======================
~S = [ %x -> ~S ] %x              ~S = [ %y -> ~S ] %z
```

If there are error messages for implicit equalities, we associate those error messages with the corresponding explicit equalities. Then we replace the variables in the error message by the fresh meta-variables of the corresponding explicit equality and add the error message as a normal error to the explicit equality.

In order to treat the introduced explicit equalities like the equalities introduced in the type system specification, we have to introduce corresponding judgments. This is important as we rely in the template optimization phase on the information about the non-terminals in the judgments.

To introduce judgments for equalities that are generated from implicit equalities, we have to determine the non-terminals of the equality. We infer the non-terminals by analyzing the surrounding terms and by exploiting the scope of the meta-variable.

In case the meta-variable is directly in an input or output position of a judgment, context binding or context lookup, we infer the non-terminal from the judgment or context definition. Otherwise, we fetch all productions of the target language that could possibly have produced the surrounding term and extract the non-terminal from the corresponding position. If there is more than one production that could have created that term, we try to disambiguate by the scope of the meta-variable. In case there is still more than one production with different non-terminals for this meta-variable, we raise an exception and this equality has to be made explicit in the specification.

After this step, there are no implicit equalities left, i.e. all meta-variables within a judgment are different. As a next step we analyze the premises of the typing rule for dependencies. One premise depends on the other, if it uses meta-variables in input positions that occur in an output position of another premise and not as an input position in the conclusion. As we need to supply the input positions with concrete terms to check if a premise holds, we can only evaluate a dependency if we first evaluate all its dependencies. In the generation process we do two things: We sort the premises topologically according to their dependencies and annotate them with the output positions of the dependencies, as described in the previous paragraph. The implementation of this ordering is almost standard. First we collect all premises and their dependencies in a graph. Due to the nature of the dependency it is more natural to create edges from a premise to the premises it

```
                                   Template(
                                       Some("T-var")
                                     , [ ( Lookup(
                                             1
                                           , [Var("X52")]
                                           , [Var("X53")]
                                           , None()
%x : ~T in $C                              )
============== T-var                     , []
$C |- %x : ~T                            )
                                       ]
                                     , Conclusion(
                                         1
                                       , ([Var(Var("X52"))], [Ctx(1)])
                                       , [Var("X53")]
                                       )
                                     )
```

Figure 5.2: Typing rule and template of T-Var

depends on. As it is easy to check if a premise has an input position that does not occur in the conclusion but as the output of another premise. Therefore, the resulting dependency graph is a transposed dependency graph. We sort this graph topologically with the algorithm by Kahn [Kah62]. The result is the reversed order, as we used the transposed dependency graph. Therefore we reverse the result of the topological sort. If we encounter cycles in the dependency graph we report them and abort the template generation.

After we have resolved implicit equalities and dependencies we begin to generate templates. This process is, except for technical subtleties, a straight forward rewriting of the rules from the specification.

5.4 Constraint Template Optimization

In this section we describe and motivate the optimization strategies we have used to reduce the number of non-syntax directed templates. However, the template optimization phase can potentially do arbitrary modifications. We focus here on optimization strategies that reduce ambiguities between the templates.

Informally, two templates are ambiguous if it is not known a priori which (if any) of the templates can be used to extend a derivation to a successful derivation. Therefore, we have to use the rules according to a heuristic and to backtrack if the decision did not work out. The goal of the optimization strategies described in this section is to eliminate some of those ambiguities before we even start to create derivations.

Before introducing the optimization strategies, we define more formally what an ambiguity is.

Definition 5.2. A template is *when-ambiguous* if there is another template such that there is at least one term that matches the conclusion of both templates.

Definition 5.2 describes the general case of an ambiguity. There is an ambiguity if there is a template that has an syntactic overlap with another template. We call those ambiguities when-ambiguity, because it is not clear when (i.e. at which point in the derivation) we have to apply an ambiguous template.

Definition 5.3. A template is *which-ambiguous* if there is another template such that the set of terms matching the conclusion of both templates is equal.

Definition 5.3 aims at more limited ambiguities. Templates are which-ambiguous if they match exactly the same terms. In other words their conclusions are equal modulo variable renaming. As there are two (or more) templates that apply to exactly the same terms, it is not the question *when* we have to apply a template, but only *which* one we have to apply. The distinction of those two kinds of ambiguities will help to implement optimization strategies. Of course all templates that are which-ambiguous are also when-ambiguous.

Before describing the optimization strategies we implemented, we extend the template language to make ambiguities explicit.

Definition 5.4. The following grammar defines the extended template language.

$\langle \mathit{Template} \rangle$ ::= ... | '`Fork`' $\langle \mathit{Templates} \rangle$

$\langle \mathit{Templates} \rangle$ ::= $\langle \mathit{Template} \rangle$ $\langle \mathit{Template} \rangle$ | $\langle \mathit{Template} \rangle$ $\langle \mathit{Templates} \rangle$

The new constructor `Fork` has a list that contains at least two templates. The idea is that templates contained in `Fork`s are all when-ambiguous to each other. We later use those groups of ambiguous templates to decide at which point in the derivation we have to decide which template we apply. A fork with less than two templates contains no ambiguities and is therefore always decomposed.

We do four types of optimizations in the optimization phase. First we eliminate which-ambiguities that are due to redundancies. Second we unfold when-ambiguous templates by inserting all possible structures in the variables of the conclusion. Directly after this we eliminate which-ambiguities that are due to redundancies and were created by the unfolding. Third we remove all templates that have unsatisfiable premises and fourth we remove all valid premisses from templates. After those optimizations we order all templates in forks such that the most general templates are evaluated at last.

5.4.1 Which-Ambiguities

Now we look now into the optimization of which-ambiguities. First we identify all which-ambiguities and create `Fork`s of them. We implemented this using a strategy that groups lists of templates such that each group contains only which-ambiguous templates. Two templates are which-ambiguous if they have the same judgment number (i.e. the position in the judgment declaration section) and if the term pattern and context pattern of the conclusion are equal modulo variables. Of the resulting template groups we wrap all groups with more than one element into a `Fork`.

We optimize forks containing which-ambiguities by checking whether a template of a fork is subsumed by a template that is which-ambiguous to that template.

Definition 5.5. A template t_1 subsumes another template t_2 if t_1 and t_2 are which-ambiguous to each other and the following formula holds

$$\forall FV(p_1, q_1, \dots, p_n, q_m) \,.\, ((p_1 \wedge \dots \wedge p_n) \implies (q_1 \wedge \dots \wedge q_m)) \tag{5.1}$$

where $p_1 \dots p_n$ and $q_1 \dots q_m$ are the premisses of t_2 and t_1 respectively.

A template t_2 that is subsumed by a template t_1 can be removed without changing the semantics of the type system. To show this, we have to argue that at any position where t_2 can be applied, we can also apply t_1. As the conclusion of both templates matches the exact same terms, we can attempt to apply both templates at the same positions. Now we have to ensure that whenever all premisses of t_2 are satisfied, all premises of t_1 are satisfied as well. Which is exactly our definition of subsumption.

In Example 20 we have two subtyping rules for records. The rule `Depth-1` is the standard depth subtyping rule for records from Appendix A.2. The other rule `Depth-2` was introduced to make depth subtyping of records reflexive.

Example 20.

```
judgments Type{I} "<:" Type{I}.

rules

~T <: ~S
{ $R } <: { $U }
=============================== Depth-1
{ %l : ~T $R } <: { %l : ~S $U }

~T = ~S
{ $R } <: { $U }
=============================== Depth-2
{ %l : ~T $R } <: { %l : ~S $U }
```

Now we consider the presence of a general reflexivity rule for the subtyping relation, like:

```
T = S
====== Refl
T <: S
```

In the presence of rule `Refl` the rule `Depth-2` is subsumed by the rule `Depth -1` as the conclusions of both rules are equal modulo variables and we can prove with the help of a first-order formula representation of rule `Refl` that the following formula holds.

$$\begin{aligned}\forall r, u, t, s \,.\, ((t = s \wedge \mathit{record}(r) <: \mathit{record}(s)) \implies \\ (t <: s \wedge \mathit{record}(r) <: \mathit{record}(s)))\end{aligned} \tag{5.2}$$

The first conjunct of the conclusion can be proven by applying the first conjunct of the premise to the first-order formula representation of rule `Refl` and the second conjunction is verbatim the second conjunct of the premise.

We try to prove that a template subsumes another for all which-ambiguous templates using vampire by transforming the templates into a first-order formula representation and generating the formula of Definition 5.5. If the proof succeeds we remove the subsumed template from the fork. If no proof can be found in the given time limit we assume the template is not subsumed and do not delete it. The translation of templates into a first-order formula representation is similar to the translation of specifications into a first-order formula representation and actually reuses most of the code.

5.4.2 When-Ambiguities

Introducing the rule `Refl` rendered one of the depth subtyping rules for records redundant and we were able to detect and remove that redundancy. However the rule `Refl` introduces a when-ambiguity into the type system, as its conclusion matches all types and not just records and there is not just only the rule `Refl`.

For the examples we now assume we have the type system specification of Appendix A.2, where types are defined as following:

$\langle \mathit{Type} \rangle$ ::= ‘`Int`’ | $\langle \mathit{Type} \rangle$ ‘`->`’ $\langle \mathit{Type} \rangle$ | ‘`{`’ $\langle \mathit{Record} \rangle$ ‘`}`’

$\langle \mathit{Record} \rangle$ ::= ϵ | $\langle \mathit{ID} \rangle$ ‘`:`’ $\langle \mathit{Type} \rangle$ $\langle \mathit{Record} \rangle$

A template within a fork that has only variables in its conclusion judgment is always ambiguous as a fork contains at least two templates. We unfold such templates to make them syntax directed. Unfolding means that we create templates with all possible structural variants of the variables. To do this we first look up the non-terminals of the variables in the judgment definition. For all non-terminals we then fetch the corresponding SDF productions from the target language. Then we generate templates where we instantiate the variables with all possible combinations of the productions. Non-terminals in the productions are replaced by fresh variables.

Example 21.

```
Int = Int           ~A -> ~B = Int           Int = ~C -> ~D
========== R1       =============== R2       =============== R3
Int <: Int          ~A -> ~B <: Int          Int <: ~C -> ~D

~A -> ~B = ~C -> ~D         { R } = Int       Int = { S }
==================== R4     ============ R5   ============ R6
~A -> ~B <: ~C -> ~D        { R } <: Int      Int <: { S }

{ R } = C -> D         A -> B = { S }         { R } = { S }
=============== R7     =============== R8     =============== R9
{ R } <: C -> D        A -> B <: { S }        { R } <: { S }
```

Example 21 shows the unfolding for the template `Refl`. That unfolding creates from non-syntax directed templates a set of syntax directed templates. Of course the unfolding can create new ambiguities and not all templates are applicable at all. In the current optimization step we will try to eliminate the newly created which ambiguities. We do this in the same way as described before, except that we now also try to proof the formula from Definition 5.5 by structural induction. Before we describe how we do the induction, we explain the problem at Example 21.

When comparing the typing rules from Example 21 and the typing rules from the type system specification of the simply typed lambda calculus from Appendix A.2 we see that the following two rules have the same conclusion modulo variable renaming.

```
                                    ~C <: ~A
~A -> ~B = ~C -> ~D                 ~B <: ~D
==================== R4             ==================== S-arrow
~A -> ~B <: ~C -> ~D                ~A -> ~B <: ~C -> ~D
```

We can try to show that one of these templates subsumes the other. We now try to prove that the left template is subsumed by the right template.

$$\forall a, b, c, d \,.\, (a\texttt{->}b = c\texttt{->}d \implies (c <: a \wedge b <: d)) \tag{5.3}$$

$$\forall a, b, c, d \,.\, (a = c \wedge b = d \implies (c <: a \wedge b <: d)) \tag{5.4}$$

$$\forall a, b, c, d \,.\, (c = a \wedge b = d \implies (c <: a \wedge b <: d)) \tag{5.5}$$

5.4 follows from the injectivity of the type constructor `->` and 5.5 from the symmetry of equality. But now we are stuck as we cannot show $c = a \implies c <: a$ nor $b = d \implies b <: d$, because we have removed the general reflexivity rule. However we can prove 5.3 by structural induction on a, b, c, and d. We will now prove some cases of the induction, the other cases are analogous. We first show the base case with $a = b = c = d = \texttt{Int}$.

$$\texttt{Int -> Int} = \texttt{Int -> Int} \implies (\texttt{Int} <: \texttt{Int} \wedge \texttt{Int} <: \texttt{Int}) \tag{5.6}$$

The premise of the implication in the base case 5.6 is valid as the terms are syntactically equal. The conclusion of the implication holds because of rule `R1`, which was created by the unfolding.

Now we show a step case with $a = a_1\texttt{->}a_2$, $b = \texttt{Int}$, $c = c_1\texttt{->}c_2$ and $d = \texttt{Int}$.

$$(a_1\texttt{->}a_2)\texttt{->Int} = (c_1\texttt{->}c_2)\texttt{->Int} \implies (c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \wedge \texttt{Int} <: \texttt{Int}) \tag{5.7}$$

$$(a_1\texttt{->}a_2) = (c_1\texttt{->}c_2) \wedge \texttt{Int} = \texttt{Int} \implies (c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \wedge \texttt{Int} <: \texttt{Int}) \tag{5.8}$$

$$(a_1\texttt{->}a_2) = (c_1\texttt{->}c_2) \implies c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \tag{5.9}$$

$$(c_1\texttt{->}c_2) = (a_1\texttt{->}a_2) \implies c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \tag{5.10}$$

$$(a_1 <: c_1) \wedge (c_2 <: a_2) \implies c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \tag{5.11}$$

5.8 follows from the injectivity of the type constructor `->`. We can drop $\texttt{Int} = \texttt{Int}$ from the premise of the implication as it is valid and make the second conjunct of the conclusion true by applying rule `R1`, which leads to 5.9. 5.10 follows from symmetry of equality and 5.11 by applying the induction hypothesis to $(c_1\texttt{->}c_2) = (a_1\texttt{->}a_2)$. 5.11 holds as it is a variant of rule `S-arrow`.

Note that we can apply rule `S-arrow` as we want to retain this rule, but not `R4` as we are trying to show that this rule is redundant. If we would need it, it would not be redundant.

We now show the case where $a = \{r\}$, $b = \texttt{Int}$, $c = \{s\}$, and $d = \texttt{Int}$.

$$\{r\}\texttt{->Int} = \{s\}\texttt{->Int} \implies (\{s\} <: \{r\} \land \texttt{Int} <: \texttt{Int}) \tag{5.12}$$
$$\{r\} = \{s\} \land \texttt{Int} = \texttt{Int} \implies (\{s\} <: \{r\} \land \texttt{Int} <: \texttt{Int}) \tag{5.13}$$
$$\{r\} = \{s\} \implies \{s\} <: \{r\} \tag{5.14}$$
$$\{s\} = \{r\} \implies \{s\} <: \{r\} \tag{5.15}$$

Here 5.13 again follows from the injectivity of the type constructor `->`. We can cancel out `Int` = `Int` as it is valid and the second conjunct of the conclusion follows from rule `R1`, which leads to 5.14. Now 5.15 follows from symmetry of equality and it holds as it is a variant of rule `R9`. Those were the three interesting cases of the induction the other cases follow analogous.

Now we have shown that template `R4` is subsumed by template `S-arrow`, thus we can remove template `R4`. As it would be tedious to proof those propositions by hand, we have implemented a tactic that attempts fully automated structural induction proofs of this kind.

After unfolding the non-syntax directed templates we collect which-ambiguous templates and attempt for each pair a direct proof that one is subsumed by the other. If this proof fails we attempt a structural induction on all free variables.

The induction cases are generated by collecting the non-terminals for each variable we do induction on. If there are multiple non-terminals for a variable we collect all of them to be sure that we do not miss a case. After we have collected the non-terminals we fetch the corresponding productions and create every possible combination of productions under consideration of the variable position. All non-terminal productions are replaced by fresh and arbitrary constants. We introduce constants instead of variables to be able to control where induction hypothesis are applied.

Example 22. Suppose we have two variables x_1 and x_2 for which we have collected the non-terminal `Type`, with the following productions.

$\langle \textit{Type} \rangle$ ::= '`Int`' | $\langle \textit{Type} \rangle$ '`->`' $\langle \textit{Type} \rangle$

From this we create the following cases:

- $x_1 = \texttt{Int}$ and $x_2 = \texttt{Int}$
- $x_1 = \texttt{Int}$ and $x_2 = c_1\texttt{->}c_2$
- $x_1 = c_1\texttt{->}c_2$ and $x_2 = \texttt{Int}$
- $x_2 = c_1\texttt{->}c_2$ and $x_2 = c_3\texttt{->}c_4$

What is left to do is the generation of induction hypothesis. While creating the induction cases we keep track of the non-terminals of the constants. In Example 22 all variables have the non-terminal `Type`. Now we create induction hypothesis for each case, by substituting constants into the proposition we then to prove such that the type of the variables in the proposition matches the type of the substituted constants. We can do this for all constants, as each constant is structurally smaller than the term in the induction case. This is the case because we substituted only non-terminals within a substructure with constants.

5.4.3 Unsatisfiable Templates

The optimization in the previous section left us with a bunch of templates that do not seem to be applicable at all. For example look at template `R2` in Example 21. This template has the premise `~A -> ~B = Int` which is unsatisfiable, as those terms are syntactically different. However, to apply a template, we have to satisfy all its premises. Therefore, template `R2` is not applicable at all.

We remove templates with an unsatisifable premise by attempting a proof that the negation of the premise is valid. If this proof succeeds we remove the whole template, otherwise we cannot be sure and keep the template.

Applying this strategy to all templates of Example 21 after the optimization of the previous section the following templates of the unfolding remain:

```
Int = Int                            { R } = { S }
========== R1                        =============== R9
Int <: Int                           { R } <: { S }
```

5.4.4 Valid Premises

Now that we have remove templates that contain unsatisfiable premises, we can look at premises that are always satisfied, i.e. at valid premises. As they are always satisfied, we do not have to check whether they are satisfied during type checking and can remove them from the corresponding template.

Following the same scheme as in the previous section, we attempt a proof that a premise is valid. If that attempt succeeds we remove the premise from the corresponding template, otherwise we cannot be sure and keep the premise.

If we look at the remaining rules in the previous section, we observe that `Int = Int` is a valid premise. Therefore the resulting set of unfolded templates looks like:

```
                                     { R } = { S }
========== R1                        =============== R9
Int <: Int                           { R } <: { S }
```

If we compare the remaining rules with a set of hand-crafted syntax directed subtyping rules for the type system in Appendix A.2, we see that the only possible further optimization is to show that template `R9` is redundant. However this cannot be optimized with the current strategies as we can only check if which-ambiguous templates subsume each other and `R9` is strictly when-ambiguous.

5.4.5 Ordering

At the end of the optimization phase we sort the templates in the remaining forks. Templates with a more special conclusion are in front of forks with a more general conclusion. This ordering has the advantage that we can evaluate the templates in a fork from left to right, so that no general template will catch all terms. More formally, we sort the templates within a fork by $<$, which is defined as

$$t_1 < t_2 \iff \exists\varphi\,.\,t_2 \cdot \varphi = t_1 \tag{5.16}$$

where φ is some substitution and $\cdot$ is substitution application. We implement the comparison as a Stratego strategy and use the quick sort implementation of the Stratego standard library. The comparison strategy does the following: It checks whether the judgment number is equal, which is always satisfied as we compare templates within a fork. Then it checks whether the constructors of the term and context pattern of the left operand match the constructors of the term and context pattern of the right operand modulo variable names.

5.5 Constraint Generation

Constraint generation is the first phase of the generated type checker. The inputs of this phase are templates generated from a type system specification and the program that should be type checked. The only input method for expression we currently support is via the conjecture section of the specification. However it should be straightforward to extend this phase with support for e.g. `stdin`.

The constraint generation phase is structured as follows. First we initialize contexts and then generate constraints according to the program's structure and the templates. After generating the constraints we pass to the constraint solver whether the expression should be well-typed or not.

Lets first look at the implementation of contexts. We store all contexts in a hash table (called *context store*), with the context number as key. The contexts itself are hash tables extended with a list that keeps track of the insertion order. Hash tables are a convenient data structure in the Stratego standard library to store (multiple) data points at a key. They reflect the intuition of the context declarations as cross products of the terms corresponding to non-terminals. It is important to track the insertion order, because some type systems (e.g. ordered type systems) might rely on the ordering within contexts. For every program expression a fresh context store is created and initialized according to the program expression.

For the context representation we decided to use hash tables instead of normal key-value lists to reduce the lookup time in realistic programs which declare identifiers much less then they refer to them. However we do not have experimental evidence if this actually speeds up type checking, because we do not have a key-value list implementation of contexts for reference.

Implementing contexts using extended hash tables forced us to separate context operations from normal inputs. On the one hand this creates more exceptional cases (e.g. more complex matching algorithms that are described in Section 5.4) and on the other hand it leads to a modular context implementation with the potential to exchange the implementation easily.

Before we can explain how we generate constraints, we have to define the constraint language.

Definition 5.6. Constraint Language

⟨*Constraint*⟩ ::= '`CFail`' ⟨*Error*⟩
  | '`CEq`' ⟨*Term*⟩ ⟨*Term*⟩ ⟨*Error*⟩
  | '`CNeq`' ⟨*Term*⟩ ⟨*Term*⟩ ⟨*Error*⟩

We annotate all constraints with error messages. These error messages are either generated during the constraint generation or passed through from the specification. The constraint solver shows the error messages if a constraint cannot be solved. In Definition 5.6 are three types of constraints. The constraint `CFail` corresponds to false and cannot be solved. Further `CEq` and `CNeq` are equality respectively inequality constraints.

To start the constraint generation, we need besides the templates and the program, information about the initial contexts, the judgment number of the typing judgment and the output positions of that judgment. This information are available as we read the program expression from the conjecture section of the type system specification. If we would only read the program expression (e.g. from `stdin`) we would need to pass information about the initial contexts, the judgment number and the output positions separately. In this case it makes sense to hard-code that information, as it is unlikely to change frequently.

We have divided the main logic of constraint generation into three Stratego strategies: `generate` selects the template that matches the current term, `generate '` applies the current term to the conclusion of the selected template and `execute` evaluates the premises of the template. All three strategies return a tuple of the computed outputs and the generated constraints. We wrap the computed outputs into `Option` to model the absence of outputs and constraints in cases of errors.

After initializing the contexts we call the strategy `generate` with the judgment number and the inputs of the conjecture. We also pass the expected outputs and set the parameter `init` to `true`. This parameter is `false` for all other cases except the initial call to `generate` and used to generate an equality constraint for the expected and actual outputs.

Now `generate` calls `find-match` to find a template whose conclusion matches the inputs, judgment number and the current state of the contexts. Before `find-match` tests the inputs and the current state of the contexts it filters all templates with the correct judgment number and tests only those templates for matches. In the case of a `Template` `find-match` checks whether the constructors of the term pattern in the conclusion and of the input term match. They can have variables for whole levels and match if they have the same constructors on all levels. In addition the context pattern of the conclusion has to reflect the current state of the contexts. The context pattern reflects the state of the contexts if it can be applied from left to right to the context without failures. In case of `Fork` only those templates are retained int the Fork that match.

`find-match` safely takes the first template that matches, because in Section 5.4 we have ensured that there is only one template that matches. Remember `Fork` is a template as well. If no template matches, we return `None` for the outputs and the singelton list containing the constraint that always fails with an appropriate error message. If `find-match` succeeds to find a template it calls `generate'` with that template.

The rule `generate'` now evaluates the selected template according to the given input. In order to evaluate the premisses of the selected template we need to extract parts of the current input term. We call the substitution of terms for variables *instantiation*. For example in a rule for function application we have to obtain the body of the function to check its type.

We initialize variables in inputs of a judgment using a term and a pattern which describes the structure of the term. We search a variable in the pattern to instantiate it, by walking through the abstract syntax tree of the pattern and record the path to the variable. Every variable is distinct as we resolved every implicit equality in the templates and have introduced no new implicit equality. Therefore we can take the first and only occurrence of the variable that matches. If we find a path to that variable, we use it to retrieve terms from the corresponding term, by walking along the path and fetching the node at its end. We ensure that the pattern and term always have the same structure, otherwise we would retrieve false positives. Because there are also other sources (e.g. premise dependencies) we extend the term and pattern if needed. We call adding to this tuple *bringing into scope*.

If the selected rule is a fork, the evaluation works as follows. In correspondence to the bindings in the conclusions context pattern, we pop all terms from the contexts and bring them into scope. This is needed for example in the typing rules for the freshness condition in SystemF, see Appendix A.1.

If a variable is bound by the outputs of the conclusion and used as the input of a premise but neither an output of an other premise or an input of the conclusion, we bring the variable into scope with the expected outputs of the conclusion. This allows to deal with typing rules that have free variables in their premises as for instance in the subsumption rule for subtyping.

Example 23.

```
$C |- ~e : ~S
~S <: ~T
============= T-Sub
$C |- ~e : ~T
```

Example 23 shows a subsumption rule for the type system in Appendix A.2. In the rule `~T` occurs free in the second premise. Binding it to the expected output of `~T` allows to still evaluate this premise.

Now we have to replace all variables introduced by the selected template with fresh variables, otherwise applying a template twice would lead to collisions. All variables introduced in the previous phases have a prefix different from `Y`. Therefore we replace all variables that do not have the prefix `Y` with a fresh variable with the prefix `Y`. Additionally this allows to verify visually whether all variables in a constraint set are fresh.

Now we evaluate the premises of the template. A key element in the evaluation is the instantiation of the variables in the premises as explained above.

In Section 5.3 we have described that the premises are topologically sorted. Therefore we can evaluate them in the given order. However, to ensure that the terms of all dependencies are available during evaluation, we have to accumulate the outputs of the evaluated premises and add those to the patterns and terms used for the instantiation.

The strategy `execute` does the evaluation of the premises. We call `execute` for each premise with a copy of the contexts to ensure that one premise cannot pollute the contexts of another premise. We have implemented a safe version of the strategy `hashtable-copy` from the standard library to ensure that after the evaluation of the premise all copies are destroyed.

We now describe what `execute` does for the different types of premises.

Lookup: The inputs of the lookup are instantiated, then the resulting term is looked up in the corresponding context. If the lookup succeeds, the result is returned as an output, together with a constraint set that contains equality constraint between the output pattern and the looked up output. Those constraints are annotated with the corresponding error message, which is instantiated and in which the hole (`{}`) is replaced by the output variables. If the lookup fails, `None` is return as output together with the `CFail` constraint and an appropriate error message.

(In)equality: Equalities and inequalities are initialized and returned as constraints, together with the output `None` as they have no output positions.

Judgment: If the premise is a judgment it is first instantiated. Then we call `generate` on the resulting input term with a version of the store that has been updated according to the context pattern of the judgment. In addition, we instantiate the expected outputs of the judgment to make the binding of free variables in premises more precise. After the call to `generate` returns we generate equality constraints for the bindings in the context pattern of the premise. Further, we generate equality constraints between the output pattern and the computed output and instantiate the corresponding error messages. The hole (`{}`) in the error messages is here replaced by the output variables.

In case `generate'` encounters a fork it evaluates all templates contained in the fork by calling `generate'`. It tries to solve the resulting constraint set and returns the outputs of the first evaluation that succeeds. If no templates can be evaluated to a solvable constraint set we return the constraint that always fails with an appropriate error message. Because of the ordering of forks in Section 5.4 this procedure ensures that rules that potentially produce solvable constraint sets but do not make real progress are deferred to the end.

5.6 Constraint Solving

Constraint solving is the last phase of the type checker. The constraint language is simple because it only consists of three constructs, the algorithm to solve a constraint set is simple as well.

We use unification to solve the constraint sets. To be even more precise we use a variant of Robinson unificiation [Rob65]. During the unification we compute a Most General Unifier (MGU) to instantiate the variables in the outputs and (in case of ill-typed programs) in the error messages that we collect every time a constraint cannot be solved.

During unification we ensure that a certain kind of malformed constraint set does not lead to infinite loops in the unification. If there are only inequalities left that contain at least one variable, we abort unification. As inequality is not transitive, we can never solve those constraint sets.

Chapter 6

Evaluation

In this chapter we present three type system specifications, discuss how the specification language affected the formulation of typing rules, and which optimization strategies are successful.

6.1 SytemF

We have implemented a version of the polymorphic lambda calculus SystemF that is close to the version described in Figure 23-1 in [Pie02]. The complete implementation can be found in Appendix A.1. There are three notable differences between the version in [Pie02] and our implementation, which we will describe in the following.

In [Pie02] term and type variable bindings are collected in a single context. We need two separate contexts, one for the term variable binding and one for the type variable binding, because we can only define homogeneous contexts.

```
contexts
TermBinding := ID{I} x Type{O}
TypeBinding := ID{I}
```

Context `TermBinding` is the term variable binding and associates identifier with types. `TypeBinding` binds type variables. Type variables are associated to nothing because we have no notion of kinds in SystemF . In our implementation variable and type identifiers are build from the same set of identifiers.

The typing judgment has to reflect that we have two contexts. Therefore our typing judgment is defined as follows:

```
judgments
TermBinding{I} "|" TypeBinding{I} "|-" Exp{I} ":" Type{O}.
```

The second difference is that in [Pie02] it is assumed “that the names of (term and type) variables should be chosen so as to be different from all names already bound” by the context. We enforce this by defining an explicit freshness check in the type system specification for term and type variables. The freshness check is implemented in a separate module, which is imported in the specification of SystemF. In the following we show the judgments and rules of the freshness check.

```
judgments
ID{I} "fresh in" TermBinding{I}.
ID{I} "fresh in" TypeBinding{I}.
ID{I} "!=" ID{I} is Neq.

rules

========================= Fresh-Term-Empty
%x fresh in (TermBinding)

========================= Fresh-Type-Empty
%x fresh in (TypeBinding)

%x != %y
%x fresh in $C
========================== Fresh-Term-Step
%x fresh in (%y : ~T ; $C)

%x != %y
%x fresh in ?C
===================== Fresh-Type-Step
%x fresh in (%y ; ?C)
```

The last difference between the text book version and our implementation is of the same kind as the last difference. In the text book version type substitution is assumed to be defined outside of the type system. We had to implement type substitution within our specification language as we have no built-in substitution mechanism. However type substitution is implemented in the same fashion like in a functional programming language with pattern matching. Substitution is also implemented as a separate module and then imported into the SystemF specification.

```
judgments
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.
ID{I} "!=" ID{I} is Neq.

rules

===================== Subst-Eq
~S = [ %x -> ~S ] %x@1    @implicit %x " does not equal " %x@1.

%y != %x
==================== Subst-Neq
%y = [ %x -> ~S ] %y

~U = [ %x -> ~S ] ~T
========================================== Subst-All
(all %y . ~U) = [ %x -> ~S ] (all %y . ~T)

====================== Subst-Int
int = [ %x -> ~S ] int
```

```
~U1 = [ %x -> ~S ] ~T1
~U2 = [ %x -> ~S ] ~T2
==================================== Subst-Arrow
~U1 -> ~U2 = [ %x -> ~S ] ~T1 -> ~T2
```

This demonstrates that our specification language is well suited to express standard type systems in a way close to text books as only one difference in the implementation is due to a restriction of our specification language. The other differences occur natural as we had to define concepts explicitly that were left implicit before.

The generated templates for our SystemF specification show that there is only one ambiguity, namely between `Subst-Eq` and `Subst-Neq`. This ambiguity cannot be solved as we would have to decide the equality of terms, before actually knowing these terms. Nevertheless the creation of forks in the template optimization phase is helpful. As we have only one fork, we know that all templates besides `Subst-Eq` and `Subst-Neq` are syntax directed.

6.2 Lambda-Calculus with Subtyping

We have implemented a variant of the simply typed lambda calculus with records and subtyping as described in [Pie02]. The specification for this type system is divided into two modules. One module specifies the type system without subtyping and the other module extends the first module with subtyping. The implementation of the type system without subtyping differs in two aspects from the text book formalization. First, we have to implement the freshness condition explicitly, as in the previous section. Second, we have to model rules that talk about all elements of a record inductively.

Formula 6.1 shows the formalization of the record typing rule from [Pie02]. This rule says that a record is well-typed if all its elements are well-typed. In 6.1 this is expressed by quantification over the elements of the record.

$$\frac{\text{for each } i\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\ ^{i \in 1\ldots n}\} : \{l_i : T_i\ ^{i \in 1\ldots n}\}} \tag{6.1}$$

This quantification is not possible in our specification language. Therefore we model this condition inductively as shown in the following.

```
                        $C |- ~e : ~T
                        $C |- { $R } : { $S }
============= base      ==================================== step
$C |- {} : {}           $C |- { %l = ~e $R } : { %l : ~T $S }
```

Here we ensure with rule `step` that the first element is well-typed and the record without the first element is well typed. Rule `base` is the base case for this definition and assigns the empty record the empty record type. Note that we support in contrast to [Pie02] the empty record. We included the empty record in our definition to demonstrate a top rule for records in the subtyping module. We have implemented the membership test for the projection typing rule `T-proj` in a similar way.

The module implementing subtyping for this lambda calculus contains a typing rule for function application that can deal with subtyping and the definition of a subtyping relation. We have added the typing rule for function application in favor of a general subsumption rule, because we do not have optimization strategies to inline the subsumption rule into the function application rule. We are able to detect that the function application rule with subtyping is more general (in the case of an reflexive subtyping relation) than the function application rule without subtyping and therefore are able to remove the latter.

The subtyping relation is defined by a generic reflexivity rule `S-refl` and a rule for function types `S-arrow` such that the argument type is contravariant and the return type is covariant.

```
                        ~T1 <: ~S1
======== S-refl         ~S2 <: ~T2
~S <: ~S                ======================== S-arrow
                        ~S1 -> ~S2 <: ~T1 -> ~T2
```

As we have seen in Section 5.4.2 rule `S-refl` is non-syntax directed and can be safely replaced by the following two rules.

```
                        { R } = { S }
==========              ==============
int <: int              { R } <: { S }
```

The subtyping relations have a further rule that defines the empty record `{}` to be the top element of records, as well as rules for with and depth subtyping and permutation of record elements. In conclusion we can specify a variant of the simply typed lambda calculus with an intuitive subtyping relation and reduce the non-determinism in the type system.

Chapter 7

Related Work

JavaCOP: Declarative Pluggable Types for Java JavaCOP [MME$^+$10] is a framework for pluggable type systems in Java. It hooks directly into `javac` and therefore integrates nicely into the normal development cycle. JavaCOP provides three tools: A declarative language to describe structural constraints on the AST of Java programs in a flow-insensitive manner, an API to use the declarative language for flow-sensitive data flow analysis and a test harness which helps to test that a program that is well-typed actually satisfies the invariants of the type system. In contrast to [Ber07] the syntax of Java cannot be extended.

A Generator for Type Checkers Gast introduces in [Gas05] a type checker generator that can produce type checkers from declarative type system specifications for functional as well as imperative and object-oriented programming languages. Type checking is done by a specialized proof search, which is based on unification and backtracking. A distinguishing feature of Gast's work is the possibility to annotate the typing rules with optimizations. For example it is possible to reuse (potentially incomplete) proofs using subproof extraction. Gast also provides a formal foundation for his proof search. However, the resulting type checkers do not report specialized error messages on ill-typed programs.

Automatic Generation of Object-Oriented Type Checkers Ortin et al. present in [OZQG14] the framework TyS for the implementation of type checkers for object-oriented programming languages. TyS provides a type checker constructor TyCC which produces a type checker when given a file that specifies the types together with the subtyping relation and operations on these types. Operations on types correspond to typing rules and are implemented using an API that is provided by TyS. Currently only Java is supported for the implementation of the operations. The addition of new languages requires the replication of the API in that language.

TyS can be used in conjunction with different parser generator tools and has been tested with flex, bison, yacc, and ANTLR. It supports the generation of static and dynamic type checkers and was applied (among others) to the object-oriented language Drill and the imperative language Frog. TyS does not support polymorphic types.

In contrast to the present work the TyS framework does not provide a high-level specification language in favor of readable generated code and is tied to object oriented languages. It however delivers a tool that has been successfully integrated in existing tool chains.

Typmix: A Framework for Implementing Modular Extensible Type Systems Typmix [Ber07] is a framework for implementing type systems for the extensible compiler xoc. Although type systems can be implemented in xoc directly, the author claims that they are often verbose and contain redundancies. Type system specifications in Typmix are written in two languages. One describes the context modifications, which are called scoperules. The other describes typing rules using premises and conclusions. The contexts in the typing rules are referred to as `Env` and concrete contexts as `Env.Ctx`. This separation of concerns increases modularity, e.g. adding a context to a judgment requires only changes were it is actually used. Typmix is used to implement a type system for an ML-like language and for FeatherweightJava.

As typmix is integrated in an extensible compiler, its focus is on extending type systems. Although the scope and type language are declarative it has no interface to proof assistants.

Automatic Type Inference via Partial Evaluation Tomb and Flanagan [TF05] use Prolog to implement type inference. If typing rules are syntax directed Prolog can type check a program efficiently. A Prolog type checker can be easily changed to infer types by leaving type variables unbound. However this can easily diverge due to Prologs depth-first search. This inefficiency is solved in [TF05] by a two phase approach. In the first phase some Prolog clauses are evaluated, which is determined by a partitioning parameter. This parameter is usually set such that relations that depend on types associated with type variables are delayed into the second phase. These two phases resemble a constraint generation and a constraint solving phase. The result of the first phase may contain arbitrary Prolog terms, but the authors claim that for common cases, the result is simple and can be efficiently solved by e.g. Datalog. This approach also applies to non-syntax directed typing rules but might be less efficient.

Just as the present work [TF05] generates a two-phase constraint-based type inference system from a high-level specification. The advantage of choosing a logic language like Prolog is that the correctness of the type inference algorithms is entailed by the correctness of the partial evaluator. However, the current approach does only work for language definitions in Prolog and does not attempt to reduce the non-determinism that is introduced from non-syntax directed rules.

Chapter 8

Summary

8.1 Conclusion

In this thesis we have presented an optimizing type checker generator that optimizes declarative type system specifications according to automatically conducted proofs.

We have introduced a high-level declarative type system specification language that can be used with most SDF syntax specifications of programming languages. This specification language allows to define type systems modular and close to the notation of text books. Further we have developed a translation of type system specifications into equivalent first-order formulas and have shown that those are suitable for type checking using automated theorem provers.

Based on the type system specification language and the first-order model for specifications, we have developed an optimizing type checker generator. Our type checker generator is modular, constraint based and uses a normalized intermediate representation of the specifications. With the optimization phase of the type checker generator we make a step towards to generation of efficient and syntax directed type checkers from non-syntax directed high-level specifications.

8.2 Future Work

In this thesis we build the foundation to create more sophisticated optimizing type checkers. The main future goal is twofold. On the one hand we want to develop more strategies to optimize type system specifications, for example to detect redundancies between strictly when-ambiguous templates. On the other hand we want to investigate into a more expressive and efficient way of proving the properties needed for the optimization strategies.

Besides that, it is desirable to extend the specification language with constructs to quantify over syntactic constructs. This would allow for example to express that all elements of a record are well typed without explicit recursion. Another improvement of the specification language would be a more modular way of specifying contexts. For example inspired by Typmix [Ber07].

Furthermore, there is room for improvements with regard to usability. Currently a project hast to be recompiled on changes of meta-variables, contexts, and judgments. SugarJ[ERKO11] provides syntax definition on the fly that could solve this problem. Another usability concern is the lack of editor integration, for example the highlighting of type errors in the program.

To improve the performance of the type checker it would be useful to explore different context implementations and to implement or use a state of the art constraint solver.

Bibliography

[Ber07] Tom Bergan. Typmix: a framework for implementing modular, extensible type systems. *Master's thesis, University of California Los Angeles*, 2007.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *ACM SIGPLAN Notices*, volume 46, pages 391–406. ACM, 2011.

[ERKO12] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In Krzysztof Czarnecki and Görel Hedin, editors, *SLE*, volume 7745 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2012. URL: `http://dblp.uni-trier.de/db/conf/sle/sle2012.html#ErdwegRKO12`.

[EvdSV$^+$13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In Martin Erwig, RichardF. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013. URL: `http://dx.doi.org/10.1007/978-3-319-02654-1_11`, `doi:10.1007/978-3-319-02654-1_11`.

[Gas05] Holger Gast. A generator for type checkers. 2005.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf. *SIGPLAN Not.*, 24(11):43–75, November 1989. URL: `http://doi.acm.org/10.1145/71605.71607`, `doi:10.1145/71605.71607`.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1975.

[Kah62] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962. URL: `http://doi.acm.org/10.1145/368996.369025`, `doi:10.1145/368996.369025`.

[KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. `doi:http://doi.acm.org/10.1145/1869459.1869497`.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[MHR+12] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2384616.2384666`, `doi:10.1145/2384616.2384666`.

[MME+10] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. Javacop: Declarative pluggable types for java. *ACM Trans. Program. Lang. Syst.*, 32(2):4:1–4:37, February 2010. URL: `http://doi.acm.org/10.1145/1667048.1667049`, `doi:10.1145/1667048.1667049`.

[OZQG14] Francisco Ortin, Daniel Zapico, Jose Quiroga, and Miguel Garcia. Automatic generation of object-oriented type checkers. *Lecture Notes on Software Engineering*, 2(4), 2014.

[PHR14] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 212–222, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2597008.2597152`, `doi:10.1145/2597008.2597152`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[Rek92] Jan Rekers. Parser generation for interactive environments, 1992.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[SM06] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006. URL: `http://dx.doi.org/10.1109/JSAC.2002.806121`, `doi:10.1109/JSAC.2002.806121`.

[TF05] Aaron Tomb and Cormac Flanagan. Automatic type inference via partial evaluation. In *Proceedings of the 7th ACM SIGPLAN international conference on principles and practice of declarative programming*, pages 106–116. ACM, 2005.

[Thi02] Peter Thiemann. Programmable type systems for domain specific languages, 2002.

[VdBdJKO00] M. G. T. Van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, March 2000. URL: `http://dx.doi.org/10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y`, `doi:10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y`.

[vdBSVV02] M.G.J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In *Compiler Construction (CC'02*, pages 143–158. Springer-Verlag, 2002.

[vdBvdMSH10] M. G. J. van den Brand, A. P. van der Meer, A. Serebrenik, and A. T. Hofkamp. Formally specified type checkers for domain specific languages: Experience report. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 12:1–12:7, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1868281.1868293`, `doi:10.1145/1868281.1868293`.

[Vis01a] Eelco Visser. Scoped dynamic rewrite rules. In *Rule Based Programming (RULE'01), volume 59/4 of Electronic Notes in Theoretical Computer Science*, pages 1–1. Elsevier Science Publishers, 2001.

[Vis01b] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001. `doi:http://link.springer.de/link/service/series/0558/bibs/2051/20510357.htm`.

[Vor95] Andrei Voronkov. The anatomy of vampire. *Journal of Automated Reasoning*, 15(2):237–265, 1995. URL: `http://dx.doi.org/10.1007/BF00881918`, `doi:10.1007/BF00881918`.

Appendix A

Type System Specifications

A.1 SystemF

```
module SystemF/Typesystem

imports SystemF/Freshness hiding (language meta-variables)
        SystemF/Substitution hiding (language)

// Available at https://www.github.com/pSub/master-thesis
language specifications/SystemF/SystemF

contexts
TermBinding := ID{I} x Type{O}
TypeBinding := ID{I} // Used to check freshness of type variables

meta-variables  Term "~" { Type Exp }
                TermCtx "$" { TermBinding }
                TypeCtx "?" { TypeBinding }
                Id "%" { ID }
                Num "&" { Int }

judgments
TermBinding{I} "|" TypeBinding{I} "|-" Exp{I} ":" Type{O}.
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.
ID{I} "!=" ID{I} is Neq.

rules

%x : ~T in $C1
@error %x "should have type" ~T "but has type" {}.
=============== T-Var
$C1 | ?C2 |- %x : ~T
```

```
============== T-int
$C1 | ?C2 |- &i : int


(%x : ~T1 ; $C1) | ?C2 |- ~t2 : ~T2
@error ~t2 "should have type" ~T2 "but has type" {}.
%x fresh in $C1
@error %x "is not fresh".
==================================== T-Abs
$C1 | ?C2 |- \ %x : ~T1 . ~t2 : ~T1 -> ~T2


$C1 | ?C2 |- ~t1 : ~T11 -> ~T12
@error ~t1 "should have type" ~T11 "->" ~T12 "but has type" {}.
$C1 | ?C2 |- ~t2 : ~T11
@error ~t2 "should have type" ~T11 "but has type" {}.
================================ T-App
$C1 | ?C2 |- ~t1 ~t2 : ~T12

$C1 | (%x ; ?C2) |- ~t2 : ~T2
@error ~t2 "should have type" ~T2 "but has type" {}.
%x fresh in ?C2
@error %x "is not fresh".
======================================= T-Tabs
$C1 | ?C2 |- \ %x . ~t2 : all %x . ~T2

~U = [ %x -> ~S ] ~T
@error ~U "is not" ~T "where" %x "is replaced by" ~S.
$C1 | ?C2 |- ~e : all %x . ~T
@error ~e "should have type all" %x "." ~T "but has type" {}.
============================== T-Tapp
$C1 | ?C2 |- ~e [ ~S ] : ~U
```

```
module SystemF/Freshness

// Available at https://www.github.com/pSub/master-thesis
language specifications/SystemF/SystemF

contexts
TermBinding := ID{I} x Type{O}
TypeBinding := ID{I}

meta-variables  TermCtx "$" { TermBinding }
                TypeCtx "?" { TypeBinding }
                Type "~" { Type }
                Id "%" { ID }

judgments
ID{I} "fresh in" TermBinding{I}.
ID{I} "fresh in" TypeBinding{I}.
ID{I} "!=" ID{I} is Neq.

rules

========================= Fresh-Term-Empty
%x fresh in (TermBinding)


========================= Fresh-Type-Empty
%x fresh in (TypeBinding)


%x != %y
%x fresh in $C
========================== Fresh-Term-Step
%x fresh in (%y : ~T ; $C)


%x != %y
%x fresh in ?C
===================== Fresh-Type-Step
%x fresh in (%y ; ?C)
```

```
module SystemF/Substitution

// Available at https://www.github.com/pSub/master-thesis
language specifications/SystemF/SystemF

contexts

meta-variables  Term "~" { Type Exp }
                Id "%" { ID }

judgments
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.
ID{I} "!=" ID{I} is Neq.

rules

===================== Subst-Eq
~S = [ %x -> ~S ] %x@1    @implicit %x " does not equal " %x@1.


%y != %x
==================== Subst-Neq
%y = [ %x -> ~S ] %y


~U = [ %x -> ~S ] ~T
========================================= Subst-All
(all %y . ~U) = [ %x -> ~S ] (all %y . ~T)


====================== Subst-Int
int = [ %x -> ~S ] int


~U1 = [ %x -> ~S ] ~T1
~U2 = [ %x -> ~S ] ~T2
===================================== Subst-Arrow
~U1 -> ~U2 = [ %x -> ~S ] ~T1 -> ~T2
```

A.2 Simply Typed Lambda Calculus with Subtyping and Records

```
module Typesystem

// Available at https://www.github.com/pSub/master-thesis
language specifications/Subtyping-Algo/SimplyTypedLambdaCalculus

contexts Context := ID{I} x Type{O}

meta-variables  Term "~" { Type Exp }
                Ctx "$" { Context }
                Id "%" { ID }
                R "$" {TRecordEntries RecordEntries}
                Num "&" { Int }

judgments
Context{I} "|-" Exp{I} ":" Type{O}.
TRecordEntries{I} "has" Exp{I} ":" Type{I}.
ID{I} "fresh in" Context{I}.
ID{I} "!=" ID{I} is Neq.

rules

============== T-int
$C |- &i : int


%x : ~T in $C
============== T-var
$C |- %x : ~T


(%x : ~T1 ; $C) |- ~e : ~T2
================================= T-abs
$C |- \ %x : ~T1 . ~e : ~T1 -> ~T2


$C |- ~e1 : ~T -> ~S
$C |- ~e2 : ~T
========================= T-app
$C |- ~e1 ~e2 : ~S


$C |- ~e : { $R }
$R has %l : ~T
========================= T-proj
$C |- ~e . %l : ~T
```

```
============= T-empty
$C |- {} : {}


$C |- ~e : ~T
$C |- { $R } : { $S }
=====================================  T-record
$C |- { %l = ~e $R } : { %l : ~T $S }


%m != %l
$R has %l : ~T
====================== Record-step
%m : ~T $R has %l : ~T


====================== Record-contained
%l : ~T $R has %l : ~T

/* Freshness Condition */

==============
%x fresh in ()


%x != %y
%x fresh in $C
==========================
%x fresh in (%y : ~T ; $C)
```

```
module Subtyping

imports Typesystem hiding (language)

// Available at https://www.github.com/pSub/master-thesis
language specifications/Subtyping-Algo/SimplyTypedLambdaCalculus

contexts

meta-variables

judgments Type{I} "<:" Type{I}.

rules

$C |- ~e1 : ~T11 -> ~T12
$C |- ~e2 : ~T2
~T2 <: ~T11
========================= T-app
$C |- ~e1 ~e2 : ~T12


======== S-refl
~S <: ~S


~T1 <: ~S1
~S2 <: ~T2
======================== S-arrow
~S1 -> ~S2 <: ~T1 -> ~T2


============ S-top
{ $R } <: {}


~T <: ~S
{ $R } <: { $U }
============================= S-depth
{ %l : ~T $R } <: { %l : ~S $U }


======================================== S-width
{ %m : ~S %l : ~T $R } <: { %l : ~T $R }


======================================================== S-perm
{ %l1 : ~T1 %l2 : ~T2 $R } <: { %l2 : ~T2 %l1 : ~T1 $R }
```

A.3 Information Flow Security Type System

```
module Typesystem

// Available at https://www.github.com/pSub/master-thesis
language specifications/STWL/STWL

contexts Domain := ID{I} x Type{O}

meta-variables  Exp "~" { Exp Type }
                AExp "~1" { AExp }
                BExp "~2" { BExp }
                Dom "$" { Domain }
                Id "%" { ID }
                Num "&" { Int }

judgments
Domain{I} "|-" AExp{I} ":" Type{O}.
Domain{I} "|-" BExp{I} ":" Type{O}.
Domain{I} "|" Type{I} "|-" Exp{I}.

rules

================ num
$dom |- &n : low

%x : low in $dom
================ var
$dom |- %x : low

$dom |- ~1e1 : low
$dom |- ~1e2 : low
========================= opa
$dom |- ~1e1 + ~1e2 : low

================== true
$dom |- true : low

=================== false
$dom |- false : low

$dom |- ~2e : low
===================== not
$dom |- not ~2e : low

$dom |- ~1e1 : low
$dom |- ~1e2 : low
========================= opr
$dom |- ~1e1 < ~1e2 : low
```

```
$dom |- ~2e1 : low
$dom |- ~2e2 : low
=========================== opb
$dom |- ~2e1 and ~2e2 : low

================== higha
$dom |- ~1e : high

================== highb
$dom |- ~2e : high

================== skip
$dom | ~pc |- skip

$dom | high |- ~e
================= sub
$dom | low |- ~e

%x : high in $dom @error %x "should have type high".
======================== assgnh
$dom | ~pc |- %x := ~1e

$dom |- ~1e : low
======================= assgnl
$dom | low |- %x := ~1e

$dom | ~pc |- ~e
$dom | ~pc |- ~f
===================== seq
$dom | ~pc |- ~e ; ~f

$dom |- ~2b : ~pc
$dom | ~pc |- ~e
================================= while
$dom | ~pc |- while ~2b do ~e od

$dom |- ~2b : ~pc
$dom | ~pc |- ~e
$dom | ~pc |- ~f
===================================== ite
$dom | ~pc |- if ~2b then ~e else ~f
```

```
conjectures

===
(z : high ;()) | high |- if z < 0 then z := 0 else z := 1

// Order of domain makes a difference in proof search!
// For the next two conjectures the applied rules are given

// VerificationSuccess(["lookup base","var","assgnl",
//                        "goal","opa"])
===
(x : low ; y : low ; z : high ; ()) | low |- z := x + y

// VerificationSuccess(["lookup base","assgnh","goal"])
===
(z : high ; y : low ; x : low ; ()) | low |- z := x + y

===
(x : low ; y : low ; ()) | low |- x := x + y

===
(x : low ; y : low ; ()) |- (x + 5) + y : high

/===
(x : low ; y : high ; ()) | low |- x := y

/===
(h : high; l : low ; ()) | high |- if h < 1 then l := 1 else skip
```