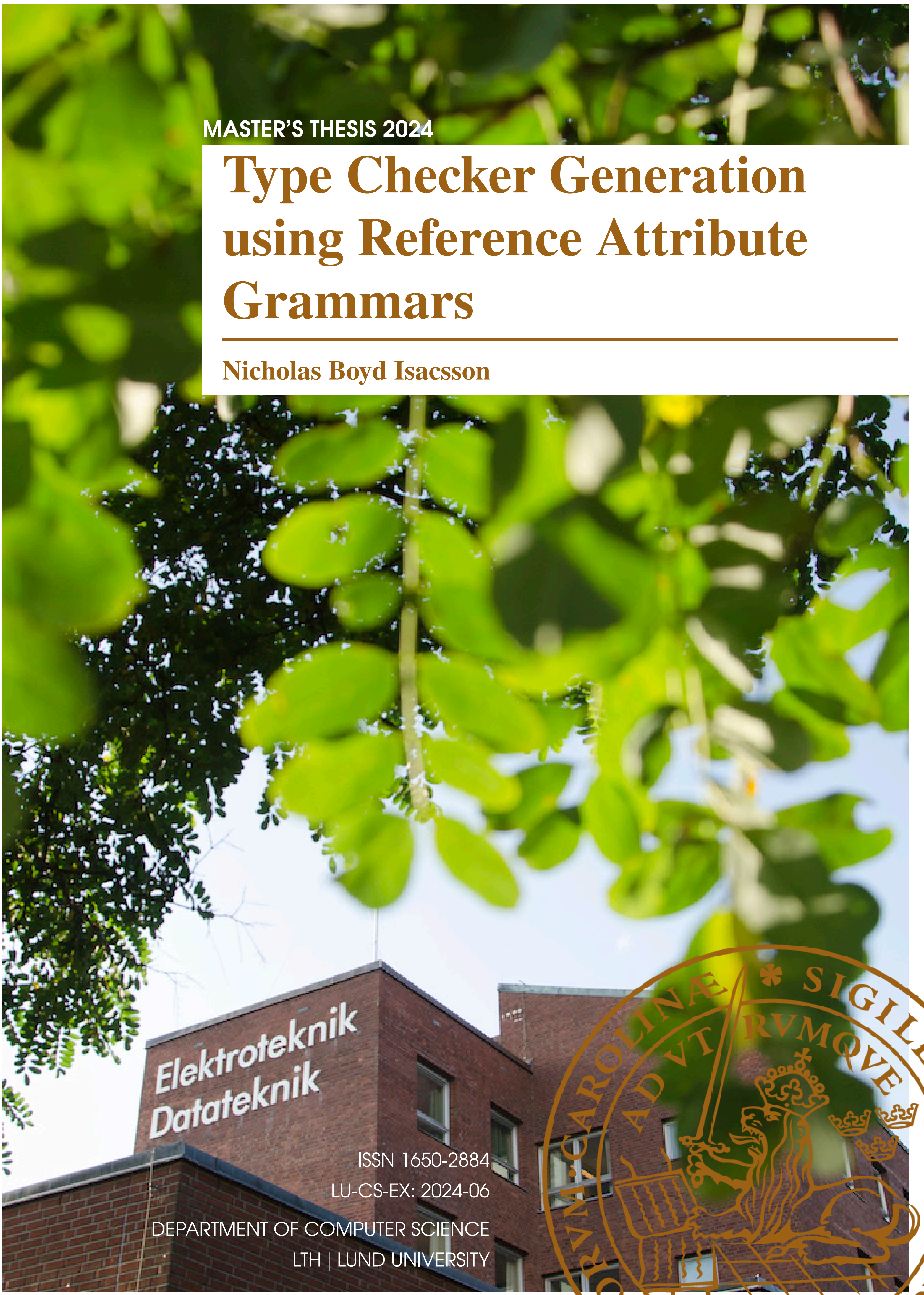


MASTER'S THESIS 2024

Type Checker Generation using Reference Attribute Grammars

Nicholas Boyd Isacsson

ISSN 1650-2884
LU-CS-EX: 2024-06

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-06

Type Checker Generation using Reference Attribute Grammars

Genererering av Typkontrollerare med Referens-Attribut-Grammatik

Nicholas Boyd Isacsson

Type Checker Generation using Reference Attribute Grammars

Nicholas Boyd Isacsson
nicholas@isacsson.se

26th January 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Christoph Reichenbach, christoph.reichenbach@cs.lth.se

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

Static type checkers are a popular method for early detection of bugs in computer programs, by ensuring the correct data types are used throughout the program. One approach for implementing type checkers are Reference Attribute Grammars (RAGs), a formalism for specifying the static semantics of programming languages by declaring attributes on an abstract tree-representation of the program, and is used by the JastAdd meta-compilation system.

This report contains a study on the usability of JastAdd to programmatically generate executable type checkers from a subset of possible typing rules, known as 'syntax-directed' typing rules. This aims to provide integration with existing compilers implemented with JastAdd, as well as a playground for type systems usable as an educational tool.

We find RAGs to be a flexible model for syntax-directed typing rules, and our software was able to support small languages from research literature. It proved difficult to provide compile-time verification of the typing rules, and further work is required to extend the compiler to support type environments, a necessity for most real-world type systems.

Keywords: type checking, type theory, code generation, reference attribute grammars

Acknowledgements

Sincere thanks to Christoph Reichenbach, whose enthusiasm for the field inspired me to pursue this project and whose guidance in matters of both implementation and writing was key to this thesis. Additional thanks to Niklas Fors for providing several valuable insights, greatly simplifying certain components and Alexandru Dura for providing some guidance to JastAdd internals.

I would also like to thank my partner Lo-Isabel Krantz Andrée as well as my family, friends, and Moritz for their support throughout the duration of this project.

```
 /\/\
:0.0 :
`,  \
 uut_).,
```

Contents

Chapter 1
Introduction

While writing software, developers often make mistakes. Catching these mistakes, as early as possible, can prevent these bugs from ever reaching production code. Type checkers are one of the most common static checking methods, ensuring that the correct data types are used. While some languages include type checkers in their compilers, work has also been done to add static type checkers to existing languages. This requires writing large amounts of specialised code on a per language basis, but has shown to be effective at detecting bugs[1][2][3]. Furthermore, making changes to the typing rules of a language can require extensive rewrites in many different parts of the type checker's codebase.

In this thesis we examine the feasibility of automating the writing of type checkers, based on the formal definition of the language's typing rules, using JastAdd's implementation of Reference Attribute Grammars (RAGs).

Pacak et al. have previously implemented a compiler for generating incremental type checkers using declarative logic programming language Datalog[4]. It focused on defining a way of expressing typing rules as a series of relations solvable by the Datalog compiler. Our use of RAGs allows us instead to build our analysis directly onto a representation of the program's structure. JastAdd is also an interesting target because it has already been used to implement compilers of several languages, to which our typing rule implementations could potentially be integrated. It is also used extensively in courses within the Department of Computer Science at LTH, where our project could serve as an educational tool, providing students an interactive way to define and test simple type systems.

Typing rules form a complex, often ambiguous, and infinite relation. To be able to translate them into JastAdd code, we will need to define a limited subset of syntax-directed typing relations. We identify a number of different kinds of typing rules, used in previous work in the field[5], which share a common set of features for us to implement. We then devise a strategy to translate these rules into snippets of iterative Java code, which can be evaluated by JastAdd.

We define a custom parsable language for typing rules adapted from natural semantics, and introduce a strategy to compile it into RAG attributes. Our case studies show that simple type systems can be systematically compiled into attributes on the Abstract Syntax Tree (AST), and the tree syntax provides a natural approach for recursive evaluation. Overall, this initial exploration shows that RAGs are a flexible approach for implementing type checkers for syntax-directed typing rules, though there are difficulties when implementing rule verification at compiler runtime.

1.1 Research questions

To formulate our goals more concretely, we have set out to answer the following research questions:

- RQ1: What kinds of typing rules can we translate using RAGs?
- RQ2: How can we implement the translations and what algorithms should we use for the output code?
- RQ3: What are the challenges in translating typing rules into JastAdd-based RAGs?

1.2 Chapter outline

The rest of this thesis is outlined as follows. Chapter 2 presents the necessary background information for the report. Chapter 3 describes the design and implementation of our program, followed by demonstrations of its results in chapter 4. Chapter 5 discusses the results in a more general sense, before the conclusion in chapter 6.

Chapter 2

Background

This chapter aims to provide all the background information required to understand the work presented in the rest of the report. Section 2.1 explains abstract syntax trees, a simplified tree-based representation of a program, suitable for performing static analysis on. These are extensively utilised in almost every stage of our project, both in terms of our implementation and the input to the program. Section 2.2 details type systems, by explaining each of the component pieces, followed by a description of their use within static type checking.

2.1 Abstract syntax trees

The syntax of a language is the definition of the symbols and structures which make up a valid program. An abstract syntax tree (AST) is a simplified tree-based representation of the syntax. Each node in the tree represents a syntactic construct of the language, such as expressions, statements or declarations and the branches represent the structure. It is abstract in that it neglects to include all the concrete details of the syntax, but rather focuses

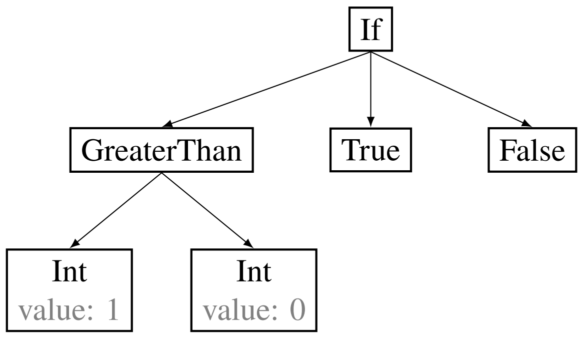


Figure 2.1: Example of an abstract syntax tree

on the structure of the program. Structural elements of the source code, such as parenthesis, commas and brackets, are no longer necessary, as the structure of the tree conveys this semantic information.

This makes it easier to write analysis based on the behaviour of a program, without concerning oneself with its exact formulation. Several different formulations of a program with identical behaviour may be abstracted into identical ASTs. The hierarchical structure of the tree lends itself especially well for recursive analyses, and eases the passing of information between connected structures by connecting them as parent and children nodes. Figure 2.1 shows an example of an abstract syntax tree representation of the expression `if 1 > 0 then true else false`.

2.2 Type theory

In this section we will discuss the elements of type theory required to understand this thesis. Firstly, we will discuss what types are and give some examples of their use. Next, we will describe type systems, which form a relation between types and expressions.

2.2.1 Values and expressions

The main purpose of most computer software is to handle and manipulate data. Each of these individual pieces of data is a value, which could be anything from the number 4 or the text ‘Hello!’.

Expressions are a broader term, indicating any syntactic structure which evaluates to a value. This includes but is not limited to values, which evaluate to themselves, but also constructs of values, operators or function calls, etc.

2.2.2 Data types

Data types can be considered a categorisation of values which share common properties, such as supporting numerical operations or accessing a certain element in a list of characters. While expressions may be arbitrarily complex, nested structures, by assigning it a type, we can perform useful analysis on it, without having to evaluate its value.

Most programming languages come with a number of built-in data types, from different kinds of numbers (integers, floating point) to strings or lists and arrays.

Different parts of programs may have different constraints on the types of expressions, such as a print function requiring a string of text, or a plus function requiring some kind of number.

2.2.3 Type checking

Verifying that these constraints are upheld can assist us in catching programming mistakes, preventing the wrong data types from being used in the wrong places, which may have led to erroneous or unpredictable behaviour. There are two main approaches to ensuring the constraints are upheld, dynamic and static type checking. As our project generates a static

```
abstract Type;

Bool : Type;
Int  : Type;
```

Figure 2.2: Type syntax consisting of two types, for boolean and integer values, written as a JastAdd AST specification (see Section 2.3.2.1).

type checker, we will focus our explanations on that topic, but provide a quick overview of their differences here.

Dynamic checking is performed at runtime, with each type checked only when needed. While this means no checks are performed unnecessarily, it makes it harder to verify the constraints over the whole program, as it will not be tested until each specific part is executed.

Static checking is performed on the code before it is run, usually as part of compilation, preventing many faulty programs from being run at all. However, since it lacks knowledge of what happens at runtime, it is required to take a more conservative approach, flagging potential errors even in code which will never executed. Static checkers analyse the structure of the source code, often in the form of an abstract syntax tree.

2.2.4 Type systems

A type system typically consists of two parts, a type syntax, describing the available types of a language; and typing rules, denoting how types should be assigned to expressions within the language.

2.2.4.1 Type syntax

We're using the term 'type syntax' to mean a definition of the valid types within a language, often referred to simply as the set of types. It can consist of fundamental types, such as integers or booleans. There may also be parameterised or composite types, for example lists may have a type parameter defining what values it can contain. This enables differentiating lists that store integers from those that store strings and ensuring that any item added to a list must be of the same type as the type parameter.

Figure 2.2 shows an example specification of a type syntax, consisting of only two types, integers and booleans. This specification is used for all the languages used in our project.

2.2.4.2 Typing rules

The typing relation, notated `t : T`, is the link between the expressions of a language and the types of the type syntax. The left-hand side is an expression in the object language, typically written in its native syntax, the right-hand side a type from the aforementioned type syntax.

Typing rules are a manner of inference rules, which in their simplest form, consist of no more than a typing relation. The typing rule `T-True` seen in Figure 2.3, for example, simply assigns the type of `Bool` to the expression `true`. Additionally, the typing rules in these examples have a name, given in parentheses next to the rule, though this is merely for reference.

$$\text{true} : \text{Bool} \tag{T-True}$$

Figure 2.3: Typing rule for a true value

Typing rules may also have premises, a number of conditions that must hold for the typing rule to apply. The notational convention here is to put the premises above a bar, with the conclusion underneath it.

$$\frac{t_1 : \text{Bool} \quad t_2 : \mathsf{T} \quad t_3 : \mathsf{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \mathsf{T}} \tag{T-If}$$

Figure 2.4: Typing rule for an if-expression

Figure 2.4 shows an example rule, typing an if-expression. The bottom line, the conclusion, contains the keywords of the if-expression and three free variables t_i which represent any term within the language. Above the bar are three premises, placing constraints on the three variables, which must hold for the typing rule to apply. The first constrains t_1 to the type bool, the second two constrain t_2 and t_3 to a type variable T, which represents any single arbitrary type within the type syntax. While T may be any type, it must consistently be the same type within any single evaluation of the rule, i.e. the two terms must have the same type. This T also appears in the conclusion, as the resulting type of the if-expression, meaning the result of the entire expression will be that of the t_2 and t_3 terms.

2.2.4.3 Evaluating typing rules

While performing static type checking, the typing rules form the basis of our analysis. By analysing the program, we attempt to match the syntactic form with the typing rules, to find the type of the expression. If no matching typing rules can be found, a type error is produced.

The simplest program possible with the typing rules T-True and T-If, consists of the single value `true`. It can immediately be matched to the rule T-True, giving the type `Bool`.

A somewhat more complex case can be expressed as `if true then true else true`. The only rule that matches the structure, is T-If, with `true` values taking the place of each of the t_i variables. To evaluate this rule, we are thus required to recursively evaluate the types each of our t_i variables, to ensure they match the premises. In this case, the type of t_1 evaluates to `Bool`, matching the first premise of T-If. The types of both t_2 and t_3, represented as the type variable T, also evaluate to `Bool`. Since both the uses of this type variable within the premises are consistent, all three of the rule's premises have been fulfilled, and the entire expression can be typed as T, which has been found to correspond to `Bool`.

$$\text{zero} : \text{Int} \qquad \text{(T-Zero)}$$

Figure 2.5: Typing rule for a zero value

Adding a second value of a different type allows us to express incorrectly typed programs, for example the program `if zero then zero else true`. Recursively evaluating this according to T-If, we find that the premise concerning t_1 no longer upholds, as the type of it now evaluates to `Int`. Additionally, the type variable T typed by both t_2 and t_3 now has contradictory use, being assigned both as the type `Int` and `Bool`. As this program cannot be found to satisfy any set of typing rules in our type system, it is found to be invalid.

2.3 Reference Attribute Grammars

This section introduces the subject of Reference Attribute Grammars (RAGs), first explaining the theory behind them, then introducing JastAdd, an implementation of RAGs within a Java meta-compilation system[6].

2.3.1 Reference Attribute Grammars

Reference Attribute Grammars are an extension to Attribute Grammars (AGs), a formalism used for specifying the static semantics of programming languages. AGs provide a systematic way to define computations on the nodes of an abstract syntax tree, associating attributes with language constructs and are commonly used in the context of compiler construction, where they help define the translation of source code into executable code[7].

RAGs extend these properties by introducing a new kind of attribute, allowing references to other nodes in the AST, and through the references accessing the attributes of other nodes. This provides a simplified and more flexible method for interconnectedness between nodes[8].

2.3.2 JastAdd

JastAdd is an implementation of RAGs for Java, offering a flexible system for writing language analysis in an object-oriented fashion[6]. Our project uses it both for the implementation of our compiler and as a compilation target, as it outputs JastAdd code.

2.3.2.1 AST specification

One of the key components of JastAdd projects are the `.ast` files, which declare the structure of the abstract syntax tree. This specification is then used by JastAdd to generate Java classes representing the nodes of the AST. These AST classes can have a variety of forms, supporting both abstract classes and subclasses. They can also contain components such as Java types, tokens and child nodes, and these components may also be optional or list components.

```
Expression ::= Term;

abstract Term;
True  : Term;
False : Term;
Or    : Term ::= Left:Term Right:Term;
Zero  : Term;
```

Figure 2.6: AST specification for the Bools language.

```
aspect TypingRules {
  syn Type Zero.type() {
    return new Int();
  }
}
```

Figure 2.7: Snippet of a .jrag file, declaring a type attribute for the Zero AST node

Figure 2.6 shows a small example of the specificaton syntax. Each of the node definitions begins with a class name. `True : Term;` indicates that the `True` node is a subclass of the abstract class `Term`. If a node has child nodes, these are provided after the `::=`, and may either have a name and a type separated by a colon, or only a type. For a detailed overview of the syntax, see the reference manual[9].

2.3.2.2 RAG Modules

The RAG modules are written in `.jrag` files and declare the attributes to be included in the classes generated by the AST specification. Figure 2.7 shows a minimal example. The files use a syntax very similar to that of Java to declare attributes of a variety of different kinds, and the body of methods corresponds directly to Java code.

Chapter 3
Approach

We begin this chapter with a section presenting our language for writing typing rules, an adaptation of the natural semantics used to notate typing rules in the academic context. In Section 3.2 we proceed with presenting the overall architecture of our software, including an overview graph in Figure 3.4, which may be useful to bear in mind while reading the rest of the report. Section 3.3 goes into the details of our implementation, how we generate code for different parts of the typing rules.

3.1 Language design

To express typing rules, we have defined an ASCII representation of the natural semantics syntax used by Pierce[5], with its concrete syntax given in Figure 3.1. Our language can express a subset of the notation used by Pierce, with some differences.

The largest difference lies in the notation used for the expression to be typed. Pierce uses the object language's native syntactic form, whereas our representation is based on the corresponding AST nodes. Node names are consistent with those used by JastAdd, followed by parentheses containing variable names to denote their child nodes in the order of their definition. This change is necessary to allow us to type arbitrary languages without needing to understand their syntax and grammar, though it does make translating rules from one syntax to the other less straightforward.

Another difference is the use of the type variables, which have been made lowercase. This disambiguates them from the concrete types, which as AST nodes themselves are written with uppercase letters, and also aligns the syntax for type terms with that of regular terms, where concrete nodes are written in uppercase and variables in lowercase. Commas are also used to separate the premises, and the horizontal line separating premises from the conclusion is represented as a series of dashes.

Figures 3.2 and 3.3 show two rules from Pierce in both their original syntax and our ASCII representation side by side. The former figure describes a simple rule named T-

```
ruleset = rule_list;

rule_list =
    rule
  | rule_list rule;

rule =
    '[' uppercase_id ']' premises_list '---+' formula
  | '[' uppercase_id ']' formula;

premises_list =
    formula
  | premises_list ',' formula;

formula = term ':' tyterm;

term =
    uppercase_id '(' term_list ')'
  | uppercase_id '(' ')'
  | uppercase_id
  | lowercase_id;

tyterm =
    uppercase_id
  | lowercase_id;

term_list =
    term
  | term_list ',' term;

uppercase_id = "[A-Z][A-Za-z0-9-]";
uppercase_id = "[a-z][A-Za-z0-9-]";
```

Figure 3.1: A definition of the concrete syntax for the typing rule language in a notation similar to Backus-Naur form (BNF).

$$\text{true} : \text{Bool} \qquad \text{(T-True)}$$

```
[T-True]
True : Bool
```

Figure 3.2: A typing rule for the value true in natural semantic syntax on the left and its equivalent ASCII representation on the right

$$\frac{t_1 : \text{Bool} \quad t_2 : \text{T} \quad t_3 : \text{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{T}} \qquad \text{(T-If)}$$

```
[T-If]
t1 : Bool,
t2 : t,
t3 : t
------------------
If(t1, t2, t3): t
```

Figure 3.3: A typing rule describing an if-expression in natural semantic syntax and its equivalent ASCII representation

True, which assigns the `true` expression the type `Bool`.

Looking at the notation of the typed expressions in Figure 3.2, the difference is minor, the syntactic term `true` has been replaced by the name of the AST node `True`. The notational change is more apparent in Figure 3.3, where instead of the full native syntactic form, the expression is written as an AST node with its children represented as parameters.

3.2 Architecture

In this section we will be presenting our architecture. Figure 3.4 provides an overview of all the components and the connections between them. The rest of the section goes into further detail about individual components.

3.2.1 Typing rule definition

To express typing rules, we created a simple ASCII representation of the natural deduction notation used by Pierce[5].

3.2.1.1 Typing rules parser

The parser for typing rules is written using JFlex[10], a scanner generator, and Beaver[11], a parser generator. JFlex takes a series of regular expressions to generate a tokenizer for the parsed language. The tokens are passed on to Beaver along with a parser definition in extended Backus-Naur form (EBNF), to generate a parser.

3.2.1.2 Typing rules AST

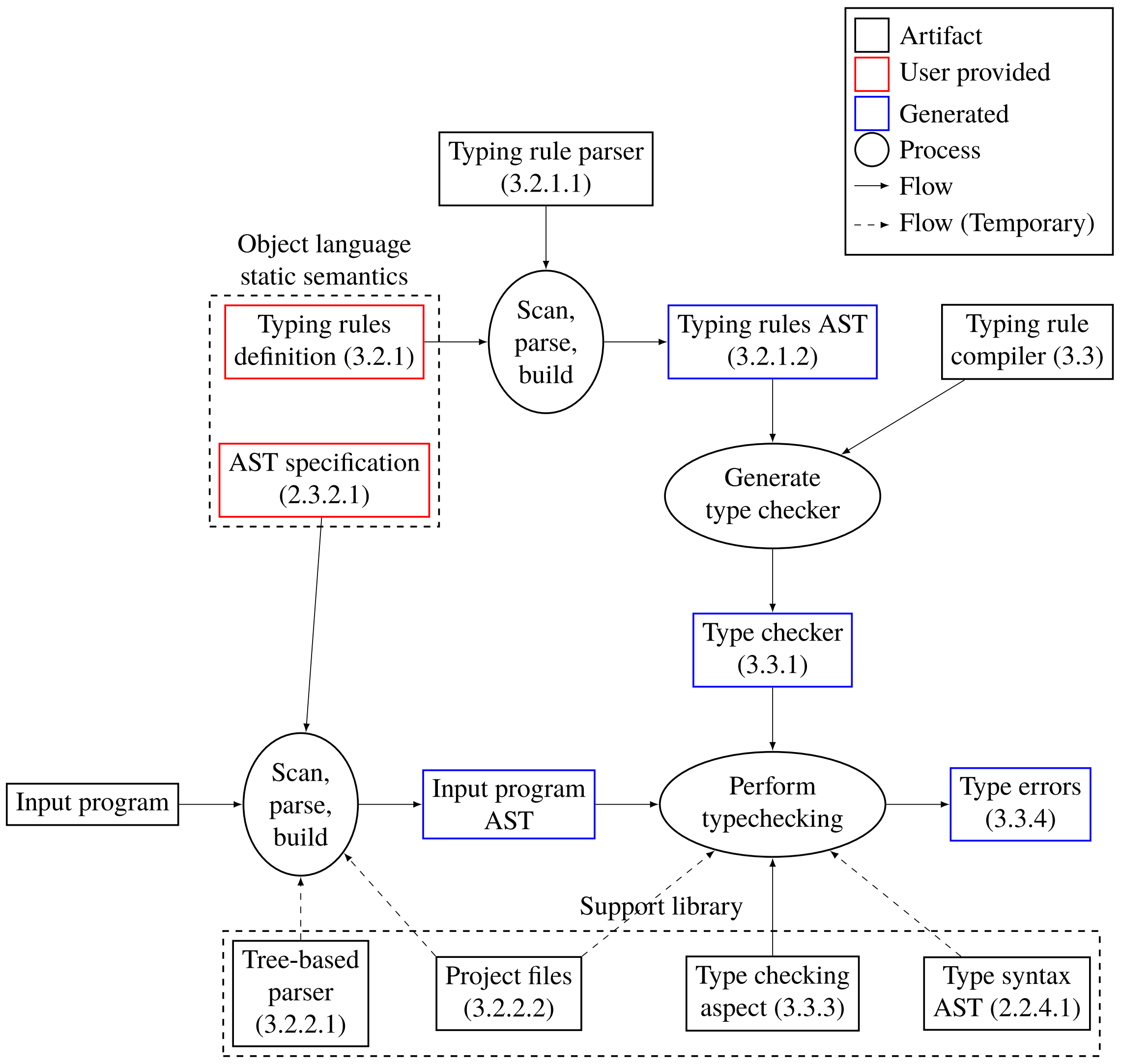
The abstract syntax tree for the typing rules language consists of a `RuleSet` root node. The root node contains only a list of rules, where each rule has a name, a conclusion and a list of premises. The rule's name is currently unused but is intended to be utilised in error reporting, to clearly specify what rules are available for each node, or which rule is causing an error.

Both conclusion and premises are represented by the abstract HasType node, consisting of a term, which may be a function or value; and a typeterm, representing either a type value or a type variable. For the conclusion, this node represents what type should be assigned to the term on the left, whereas for the premises, the node represents a requirement for the node on the left to have the type on the right.

Currently, our compiler does not support type parameters, so they are not included in the AST. To support this in future, an additional `TyTerm` could be added which includes a list of `TyTerm`s along with its ID.

3.2.2 Support library

In addition to the generated type checking files, there are a number of files that are added to the output file that are identical for all generated type checkers, which we refer to as



Artifact
User provided
Generated
Process
Flow
Flow (Temporary)
Typing rule parser (3.2.1.1)
Object language static semantics
Typing rules definition (3.2.1)
AST specification (2.3.2.1)
Scan, parse, build
Typing rules AST (3.2.1.2)
Typing rule compiler (3.3)
Generate type checker
Type checker (3.3.1)
Input program
Scan, parse, build
Input program AST
Perform typechecking
Type errors (3.3.4)
Support library
Tree-based parser (3.2.2.1)
Project files (3.2.2.2)
Type checking aspect (3.3.3)
Type syntax AST (2.2.4.1)

Figure 3.4: An overview of the project components and flow. It represents the flow of the program, with components and artefacts (represented as rectangular boxes) with arrows linking them to actions (elliptical boxes) which in turn have arrows to the new components and artefacts they produce. The dashed arrows represent links from the support library components that exist in the current experimental state of the project, but in a finished product would instead link from components out of the object language's own compilation environment.

```
RuleSet ::= Rule*;

Rule ::= <Name> Conclusion:Formula Premises:Formula*;

abstract Formula;
HasType : Formula ::= Expr:Term Ty:TyTerm;

abstract Term;
Function : Term ::= <ID> Term*;
Value    : Term ::= <ID>;

abstract TyTerm;
TyVal : TyTerm ::= <ID>;
TyVar : TyTerm ::= <ID>;
```

Figure 3.5: Typing rules abstract syntax tree specification

the support library. Some of these are key components of the type checker, but others are dummy components included solely for convenience of testing. We expect that in a real world use case, our software would integrate into an object language's compiler pipeline, and utilise its own parser and project files to provide the abstract syntax tree for our type checking to tie into. For testing purposes, we have written generic versions of these components, which work for any supported object language.

3.2.2.1 Input program parser

For testing purposes, our support library comes with a generic tree-based parser. Programs are written as a tree of AST node names followed by potential parenthesis within which a comma separated list of child nodes are written. The parser parses the tree recursively, using Java's reflection API to find a class with a matching name, and a constructor with parameters of the same amount and types. Figure 3.6 shows an example input file for the Bools language, which will be discussed further in Section 4.1.

```
Expression(
  Or(
    True,
    Or(
      True,
      False
    )
  )
)
```

Figure 3.6: Example tree-representation input for the Bools language

This approach does have the issue of occasionally parsing invalid programs, due to the reflection API finding an additional parameterless default constructor. This leads the parser to always accept nodes written with no children, such as `Expression(Or())`, even if the AST specification requires them. These children will be null, causing issues further down the line, though as this parser is only intended for testing it is of little concern. Our tests have been carefully written to avoid this error, and any faulty tests would result in obvious null pointer exceptions during further evaluation.

3.2.2.2 Project files

For ease of use the compiler will output a complete runnable Gradle project. It can parse simplified tree-representations of programs, and utilise these to perform the type checking. In a future real world use case, it is intended for our generated type checking aspects to integrate into a pre-existing project, which already has all the necessary project files to parse an AST tree, to which our type aspects could integrate.

3.3 Implementation

In this section we present our approach to translating typing rules. We start by presenting pseudocode for our typing rule evaluation. This is followed by Section 3.3.2, which explains how we generate code to implement this algorithm.

3.3.1 Type checking algorithm

```
foreach typevar in rule.typevars:
  typevar.type = null;
  foreach premise in rule.premises where premise.rhs == typevar:
    if typevar.type == null:
      typevar.type = premise.lhs.type()
    else if typevar.type != premise.lhs.type():
      throw typeerror("Type variable mismatch")

// Check if all evaluated left-hand side types match their right-hand
    side
if allTrue(rule.premises.map(p => p.lhs.type() == p.rhs)):
  return rule.conclusion.rhs
else:
  throw typeerror("Typechecking failed")
```

Figure 3.7: Pseudocode for the implementation of the type() attribute

Here we aim to give an overview of the algorithm used to evaluate the typing rules. Figure 3.7 contains a pseudocode algorithm for the evaluation of a typing rule. It is not representative of the actual generated code, but captures its behaviour in the manner of an interpreted language.

3.3.2 Code generation

In this section we will present the code we generate to implement the previously discussed algorithm. Once typing rules have been parsed, we end up with an abstract syntax tree, in the form of Figure 3.5, where the `RuleSet` is the root node. From the root node, calls propagate down the tree to fill in increasingly specific sections of the code.

3.3.2.1 Aspect generation

The `RuleSet` node is responsible for generating the overall structure of the output file, a JastAdd aspect called `TypingRules`. It sorts its `Rule` children by which nodes they apply to, so that if multiple typing rules type the same node, they can be inserted into the same method implementation (though this feature is not yet fully working).

3.3.2.2 Method implementation overview

The implementation of a typing rule can range from a single return statement to a system of declarations of variables, type variables and control flow. These are explained in further detail in subsequent sections, but here follows a quick overview.

Firstly, the algorithm checks if the number of premises is zero, in which case it immediately generates the conclusion, consisting of a return statement. If there are premises, it first generates code for the declarations of variables and both declarations and unification for type variables. This is followed by an if-statement to test whether the premises hold true, containing a return statement. Should the premises not hold, a type error is produced.

3.3.2.3 Child nodes

When the typing rules concern a node with children, these are bound to free variables which can be named arbitrarily. A variable with the same name will be instantiated at the beginning of the method, fetching the corresponding child node by index. This naming format means any errors reported will use the same variable names as the user has defined in the typing rules, in order to minimise the disconnect when analysing errors from the generated code. However, it also means it is not possible to name a variable any of the reserved keywords in Java. This could be alleviated by adding a prefix to the variable names, at the cost of a slight indirection for the user. These variables are then utilised in later premises or type variable declarations.

3.3.2.4 Type variables

In the case of type variables, much of the logic is handled at runtime, to simplify the code generation. Each type variable that appears in the conclusion is instantiated to a null value, and the code generated by each premise first checks if the value is still null and either assigns the variable its own type or tests to make sure the existing type matches its own.

The generated code could be simplified in future work by instantiating the type variable to the type of the left-hand side of the first premise it appears in, and then only checking if each of the subsequent uses match it.

3.3.2.5 Premises/control flow

For rules containing premises, the return statement is protected by an if-statement. At this point we've declared variable for each of the child nodes and type variables used, so the premises can very directly be translated into boolean expressions. A premise `x : Bool` translates into the expression `x.type().matches(new Bool());`. This includes a call to the `type()` of each left-hand side of the premise, recursively evaluating the necessary types. Since our typing rules can only reference their direct children, this recursion will always progress down the program's AST, ensuring termination. Each of the boolean expressions then joined by the `&&` operator into a single if-statement, protecting the return of the right-hand side of the conclusion. For the case where any of the boolean expressions is false, the if-statement is followed by a thrown exception representing a type error.

3.3.3 Type checking aspect

The type checking aspect contains the declaration of the `type()` attribute for all AST nodes, as well as a default implementation. This default implementation immediately produces a type error, to cause an error when the type of an expression not defined in the typing rules is demanded.

It also contains the implementation of the type matching attribute, as defined in Figure 3.8. The current implementation of `Type.matches(Type t)` checks that the two types are the exact same, though we also considered an implementation that checks if the Type it is called on is either of the same class or a subtype of the type `t`. This would allow the language limited sub-typing at the type checker generation step, but at the risk of causing hard to predict behaviour and hidden order dependencies with type variables.

It is also possible to override the `matches` implementation for a specific Type, by adding a definition to a JastAdd aspect included in the output type checker. This could potentially be used to compare type parameters in future versions, or support for ad-hoc type polymorphism.

```
syn boolean Type.matches(Type t) {
  return getClass().equals(t.getClass());
}
```

Figure 3.8: Definition of the type matching method

3.3.4 Error handling

The current error handling consists of throwing a runtime exception at the first discovered type error, ceasing the type checking immediately. This method was chosen solely for its simplicity of implementation, rather than its functionality. Possible improvements to it are discussed in Section 6.1.2.

It provides a stack trace with information on see where in the type checking code the error occurs (such as what typing rule fails), but doesn't currently provide any information about what section of the input program causes the error.

Chapter 4
Evaluation

In the first section of this chapter, we present typing rule definitions for a few simple type systems, alongside the resulting code produced as an output when they are input to our compiler.

In Section 4.2, we evaluate the overall result of our study by answering our research questions.

4.1 Results

We have chosen to evaluate our program using two example languages, inspired by the languages Booleans and Numbers introduced in Types and Programming Languages[5]. We have chosen to simplify the languages somewhat, to reduce the amount of equivalent cases for our program to parse, making it easier to evaluate each feature individually. Thus, we have defined two languages Bools and BoolsIf, where the former is a simple language consisting of boolean values, an `OR` operator, and a zero-value. This is a minimal case to produce a meaningful type checker, as it contains values of two different types and a rule with premises limiting the types.

The second language BoolsIf is an extension to the Bools language. It adds an if-expression, which typing rules utilises type variables to ensure both if and else terms have the same type, and then uses that type as its own type.

4.1.1 Bools

A simple example of a language is provided in Figure 4.1, consisting of three values, `True`, `False` and `Zero`; and the boolean operator `Or` with two parameters, Left and Right. In the AST specification, there are no limitations on what terms can be provided as parameters to the `Or` operator, allowing illogical constructs such as `Or(True, Zero)`.

```
Expression ::= Term;

abstract Term;
True  : Term;
False : Term;
Or    : Term ::= Left:Term Right:Term;
Zero  : Term;
```

Figure 4.1: AST specification for the Bools language

```
[T-True]
True : Bool

[T-False]
False : Bool

[T-Or]
left : Bool,
right : Bool
----------------------
Or(left, right) : Bool

[T-Zero]
Zero : Int
```

Figure 4.2: Typing rules for the Bools language

To prevent cases like these, we define typing rules via the specification in Figure 4.2. The typing rules for `True`, `False` and `Zero` contain no premises and define the type as `Bool` and `Int` as appropriate. The typing rule for the `Or` operator, which takes two parameters `left` and `right`, contains two premises, specifying that the `Or` operator has the type `Bool` only if both the `left` and `right` terms also have the type `Bool`.

The typing rules from Figure 4.2 generate the type definitions in Figure 4.3. The simple rule definitions for the values compile into equally simple implementations, consisting of a single return statement of a newly constructed `Type` object. For the `Or` node, the code becomes slightly more complex. It declares variables corresponding to the child nodes given in the parameters, followed by an if-statement to check if the premises are upheld by recursively evaluating the types of the child nodes. Depending on the result of this evaluation, either a `Bool` type is returned, or a type error is thrown.

This type checker accurately captures the intended behaviour of the typing rules.

4.1.2 BoolsIf

As an example of rules utilising type variables, we have extended the Bools language with an if term, as used in Pierce's Booleans language[5]. This if term captures each of the utilities of type variables in typing rules, using it both to ensure that multiple terms are of the same type and then using that same type as the result of the rule evaluation. As shown in the AST extension in Figure 4.4, it consists of three terms: a conditional, a term to be

```
aspect TypingRules {

  syn Type Zero.type() {
    return new Int();
  }

  syn Type Or.type() {
    ASTNode left = getChild(0);
    ASTNode right = getChild(1);

    if(left.type().matches(new Bool()) &&
       right.type().matches(new Bool())) {
      return new Bool();
    }
    throw new RuntimeException("Typechecking failed");
  }

  syn Type True.type() {
    return new Bool();
  }

  syn Type False.type() {
    return new Bool();
  }
}
```

Figure 4.3: Generated typing rules for the Bools language (reformatted for readability)

```
If : Term ::= Cond:Term Then:Term Else:Term;
```

Figure 4.4: The additional term introduced in the BoolsIf language, extending the Bools AST in Figure 4.1

```
[T-If]
x: Bool,
y: a,
z: a
--------------
If(x, y, z): a
```

Figure 4.5: Additional typing rules for the BoolsIf language, an extension for the Bools typing rules in Figure 4.2 with an if term

```
syn Type If.type() {
  ASTNode x = getChild(0);
  ASTNode y = getChild(1);
  ASTNode z = getChild(2);

  Type tyvar_a = null;
  if(tyvar_a == null)
    tyvar_a = y.type();
  else if (!tyvar_a.matches(y.type()))
    throw new RuntimeException("Typechecking failed: Type variable
   mismatch");
  if(tyvar_a == null)
    tyvar_a = z.type();
  else if (!tyvar_a.matches(z.type()))
    throw new RuntimeException("Typechecking failed: Type variable
   mismatch");
  if(x.type().matches(new Bool()) &&
     y.type().matches(tyvar_a) &&
     z.type().matches(tyvar_a)) {
    return tyvar_a;
  }
  throw new RuntimeException("Typechecking failed");
}
```

Figure 4.6: Generated typing rule for the if term added in the BoolsIf extension. The rest of the typing rules are identical to those in Figure 4.3. (Reformatted for readability)

evaluated if the condition is true and a condition to evaluate if false. A typing rule of this if term can be seen in Figure 4.5. The rule has three premises, the first of which declares the conditional must have the type `Bool`. The next two declare that the two other children must be of the type `a`, with the lowercase identifier indicating a type variable. This type variable `a` also subsequently appears in the resulting type of the if term.

The code for this rule is shown in Figure 4.6, and begins with three local variable declarations as in previous rules, followed by a new type variable declaration on line 6, which is instantiated to null. For each invocation of the type variable, a code section is generated, which checks if the type variable is still null, and if so assigns it to the type of the left-hand side term, or if it has been assigned checks whether the types match each other. Should any non-matches be found, a type error is thrown.

The rest of the method is similar, with premises being checked – including a redundant check of the type variables, as mentioned in Section 3.3.2.4 – before returning the type variable, or a type error being thrown. This type checker accurately captures the intended behaviour of the typing rule.

4.2 Research questions

4.2.1 RQ1: What kinds of typing rules can we translate using RAGs?

Our project has shown that generating type checkers using RAGs is a viable solution for a number of different language constructs. To evaluate the completeness of our implementation, we refer to Pierce[5] as a reference work. Typing rules are first introduced in chapter 8, with two simple languages in Figure 8-1 and 8-2. The first, Booleans (generally referred to as B) consists of only the syntactic structures `true`, `false` and `if t then t else t`. Extracts of the typing rules for the B language were used Section 3.1. The second language called Numbers (NB) is an extended version of B, adding arithmetic expressions. This introduces the terms `0`, `succ t`, `pred t` and `iszero t`.

Both the B and NB languages are fully supported by our project, and our Bools and BoolsIf languages were devised to utilise all the same language features but in a more condensed form. In chapter 9, Pierce introduces the Simply Typed Lambda Calculus, which typing rules our compiler does not yet support, as they require environments, a feature we'll discuss in Section 6.1.1.

We have implemented support for simple syntax-directed type declarations, where a value or function always has a certain type. These simple rules are not of much interest in themselves but are an important backbone to the subsequent analysis. We also support type declarations with premises that must hold for the rule to apply, allowing us to express relationships between types and bringing with them the possibility of type errors.

Type variables extend these relationships further by providing a layer of abstraction, allowing us to write rules where the types are not precisely defined. This enables rules where the return type of an expression depends on its child expressions, or where child expressions must have the same types.

4.2.2 RQ2: How can we implement the translations and what algorithms should we use for the output code?

The basis of our approach has been to translate typing rules into `type()` attributes added to the program's AST. We opted for an approach where a successfully evaluated typing rule returns a `Type` value, defined in a separate type syntax file, while a failed rule throws a runtime exception representing a type error.

In this section we will be analysing different aspects of this approach, and compare them to potential alternative approaches.

4.2.2.1 Type syntax

We have chosen to represent the right-hand side of type equations, the types themselves, as an abstract AST class. While our current implementation only supports simple types, this representation gives us a lot of flexibility for future extensions. Parametric types, such as lists where the type of the list contains information about the types of its elements, could be expressed as simply as `MyList : Type ::= Type;`.

The current implementation of `matches()` would however consider all lists to match, without comparing the child nodes. Perhaps the default implementation could be extended to compare all child elements, at the risk of potentially tricky evaluation, or these more advanced types would require manually overriding the method. Yet another approach may be to design a way to define type equality within the typing rules, allowing users to customise the behaviour without having to resort to manually writing JastAdd code.

4.2.2.2 Error handling

We have implemented premises as conditions of a single if-statement, where we check that each premise holds true before returning the type, or falling through to a type error if the premises do not hold. It might instead be worthwhile to check each condition individually, to be able to specify in the error message precisely which premises have failed and why. If paired with the current error handling of throwing an exception to convey type errors, it would make it harder to implement having multiple valid rules providing the type of an expression, as the first failure would throw an exception, stopping the evaluation of any further typing rules which may have been successful. An alternative error handling would suit it better, one able to collect multiple errors without seizing evaluation, perhaps merely adding all the errors to a list and checking whether it is empty before returning a type.

A more informative solution to this might be to view rules as a kind of pattern matching, where each rule is evaluated one by one until a matching one is found. Instead of reporting the reason any individual rule failed, the error might instead display either all or only the best matching of the rules it tried to apply, perhaps with a list of non-holding premises for each one.

4.2.2.3 Variables

The variables in the typing rules are compiled into regular Java variables containing the AST node. Notably, the declarations of these variables type them merely as `ASTNode`. As we currently only utilise the `type()` attribute, which is declared for all nodes, this lost information has little impact. However, in the future it may become relevant to type these more specifically, to support more complex rules that require deeper analysis within the tree.

4.2.2.4 Type variables

Type variables are implemented as Java variables and as we currently only support type variables scoped within a single typing rule and lack support for type parameters, all uses of it will be known directly at the evaluation of the typing rule. Thus, our unification algorithm can be very simple. All our type variables are initiated as null, to represent the type variable's unassigned state. This is followed by a check for each point of use in the typing rule, to see whether the type variable is still unassigned and if so assign it to the type of the left-hand side.

This approach could be improved by finding the first use of each type variable during the rule compilation, and generating a single declaration statement initialising it to the type of the corresponding expression.

4.2.3 RQ3: What are the challenges in translating typing rules into JastAdd-based RAGs?

4.2.3.1 Technical challenges

Writing idiomatic JastAdd components often requires splitting a single algorithm into many different pieces of code. Our approach to generating type attributes was based around recursively traversing the typing rule AST to fill in each bit of information, rather than having one parent node trying to form an understanding of the entire tree. The plus side of this recursive approach is that each component does not need an intricate understanding of the nodes underneath it. For example, a `Rule` node does not need to know whether the right-hand side of the conclusion is a concrete type or a type variable. The `HasType` node representing the conclusion generates the structure of the return statement, calling on the right-hand side node to fill in the exact type.

However, this code could at times become complex, as certain lines required bits of information from several different parts of the AST, several steps apart. Propagating this information required adding intermediate attributes to nodes throughout the AST, leading to a complex set of interdependencies. If we want to change a specific token of the generated code, tracing down the exact attribute it is generated in can be tricky.

Using JastAdd simultaneously as a library and a build component proved difficult on several occasions. The largest issue was that some of the features we intended to support, such as verifying that typing rules correspond to valid nodes within the object language AST, were complicated by having to parse the object language AST at runtime, using JastAdd as a library rather than a build tool. JastAdd does not expose any methods for

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \tag{T-Sub}$$

Figure 4.7: A typing rule for subtyping, an example of a non syntax-directed typing rule.

parsing an AST tree at runtime, and we had to access undocumented, private methods through Java's reflection API to get the relevant information. Unfortunately, the resulting data was hard to work with, and we could not allocate enough time to implement the planned features. It may be that this use case is beyond the intended scope of JastAdd, or it is perhaps only poorly documented. In either case, while our experimentation has shown it to be technically possible, the complexity is worth bearing in mind for future work.

Not being able to perform these error checks during the generation of our type checker means our translation will emit nonfunctional typing definitions if a mistake is written in the typing rules. These errors will then go unreported until the output project is compiled or run, at which point the user will have to attempt to trace these errors back to the typing rule definition, to find the actual cause. As JastAdd itself also adds a layer of indirection, by taking the attributes defined in RAG modules and compiling them into generated Java class files, this can be particularly tricky.

We also ran into a few tooling issues. The first was Gradle not willing to recognise JastAdd as both a library and a build component, which we eventually solved by manually providing a jar file separately from the resolved build dependency. The second one was our Language Server Protocol (LSP) server, jdtls, not recognising JastAdd imports and flagging every use thereof as an error, which we never managed to fix.

4.2.3.2 Conceptual challenges

On the more conceptual side, a fundamental challenge of our approach is how to implement non syntax-directed rules. The structure of our program is based around turning typing rules into evaluable attributes on the AST. This means we inherently tie typing rules to a specific AST node, creating a direct link between typing rules and the syntax.

However, more complex type systems may require rules that are not syntax-directed, where a rule does not apply to any specific AST node. A common example use case for this is subtyping, where any expression of a certain type can also be considered to be any of that type's supertypes. Figure 4.7 shows a subtyping rule taken from Pierce[5].

With our approach, this rule would have to change the behaviour of every generated `type()` attribute. Additionally, we would require an implementation supporting type polymorphism, where a single term can be evaluated to have multiple different types.

Chapter 5
Related work

Pacak et al. implemented a compiler for generating incremental type checkers using declarative logic programming language Datalog[4]. While the goal is similar to ours, their project defines a formalism for expressing typing rules within the Datalog language, whereas ours defines a language which aims to keep as close as possible to the traditional inference rules used for typing rules, to minimise the difficulty of translation.

Their work ends up largely focused on expressing typing rules within the limits of the Datalog language, which can only compute finite relations. We have a significantly different compilation target, JastAdd, which allows us to use reference attribute grammars to evaluate the typing relations, utilising a built-in AST structure. Whereas Datalog is a fully declarative language, with JastAdd we express typing relations as declared attributes on the AST, with their implementations written in imperative Java code.

Other related work has been done in the domain of compiling natural semantics, without the specific focus on generating type checkers.

An early compiler for natural semantics was made by Mikael Pettersson in 1996[12]. They wrote a compiler in Standard ML which generated C code, linked to a custom runtime system.

Saioc and Hüttel developed a compiler for natural semantics[13]. They did so by defining a parsable meta language adaptation of natural semantics, similar to the one we defined for writing typing rules. Their project was however focused on verifying the correctness of these definitions and finding errors in their definitions. This was something we had hoped to explore more in our own work, but ended up deprioritising after running into issues with JastAdd.

Chapter 6
Conclusions

Our experiments have shown that reference attribute grammars are a flexible method for implementing type checker generators. Implementing the first end-to-end prototype was challenging and took quite a lot of work, but extending it with additional features such as type variables turned out to be a surprisingly painless experience.

However, our compiler still has a long way to go before it can be used for real world use cases and general purpose languages. Chief among the missing features is the concept of environments, without which we are unable to support even common language features such as variables or functions.

6.1 Future work

6.1.1 Environments

Implementing environments, or type contexts, would be the clear next step for this project. However, its implementation is not entirely clear-cut, with several aspects requiring consideration.

The essence of environments is the mapping of variables or functions to types, and an appropriate representation of this would need to be found. A method often used by traditional type checkers is a symbol table, a separate data structure mapping language constructs to their types. In our case, we may instead be able to leverage our reference attribute grammars to store the information within the AST. By connecting each use of a variable or function to the AST node representing its declaration site, we could utilise the node to store the type information. This approach would however also have to consider the typing of external language constructs, such as built-in functions or those imported from external libraries, for which we would not have access to the declaration site.

A type checker working on these principles is certainly a possibility, though it remains to be seen how it could be generalised to the generation of typing rules for arbitrary languages.

6.1.2 Improved error handling

The current exception-based error handling is rather primitive and leaves much to be desired. Some improvements, such as providing more useful error messages, could be possible by making minor changes to how these exceptions are generated. Information such as what part of the input program the type error occurred at, what the failing typing rule was and what prerequisites it has, could be added to the exception message, for a quick usability improvement.

An improvement demanding more extensive changes would be collecting multiple errors, instead of seizing the type checker at the first discovered error. This could be implemented by adding another type class to be returned when an error occurs, representing the 'Any' type. Any `matches()` call on this type would return true, to avoid one error cascading into many, but still allowing other unrelated errors to be discovered. This implementation would also need new functionality to collect the error messages, perhaps by splitting the type resolution and type errors into separate attributes for each node, where the type errors propagate to a program-wide collection.

It may be possible to create this by wrapping around the current implementation with two additional attributes. One would be a total function, that returns the type evaluated by the current `type()` method if it is successful or returns the Any type in the case of an exception. The second attribute would contribute an error to the error collection only in the case of an exception. The only other changes required would be to use the total wrapped method within the generated attributes.

6.1.3 Verify rules against AST

Major improvements to the user experience could be made by tying the typing rules to the object language AST before generating the type checker. Currently, there are no checks in place to verify the completeness of the typing rules. If a user accidentally leaves out the typing rule for a certain construct, they will receive no warning until the outputted type checker attempts to check the expression's type.

Similarly, no checks are made to ensure the nodes described in the typing rules actually exist within the language's AST. Our project will compile invalid or misspelled typing rules without complaint, leading to errors which will not be caught until the outputted type checker is attempted to be compiled.

6.1.4 Handle multiple rules for the same node

In type systems it is generally considered valid to have multiple typing rules for the same construct, with different, non-intersecting prerequisites. It could, for example, be useful to type an addition operator for a variety of different numerical types, which may not have a common ancestor.

Most of the work to support this is in place, though the first rule to fail would throw a type error exception, preventing the later rules from evaluating. These errors would either have to be caught, and only emitted if all rules had failed, or a new error handling system not based around exceptions could be introduced. Additionally, an extra layer of scoping would have to be introduced, to prevent variable naming collisions between the different rules. This could be solved simply by wrapping each rule within brackets, creating a separate scope for the evaluation of each rule, or relabelling the variables to prevent collisions.

References

[1] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 681–690, New York, NY, USA, 2011. Association for Computing Machinery.

[2] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 9, USA, 2004. USENIX Association.

[3] Momina Rizwan, Ricardo Caldas, Christoph Reichenbach, and Matthias Mayr. Ezskiros: A case study on embedded robotics dsls to catch bugs early. In *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*, pages 61–68, 2023.

[4] André Pacak, Sebastian Erdweg, and Tamás Szabó. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[5] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[6] Görel Hedin and Eva Magnusson. Jastadd - a java-based system for implementing front ends. *Electronic Notes in Theoretical Computer Science*, 44:59–78, 06 2001.

[7] Donald Ervin Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2:127–145, 1968.

[8] Görel Hedin. Reference attributed grammars. *Informatica*, 24(3):301–317, 2000.

[9] Jastadd reference manual. `https://jastadd.cs.lth.se/web/documentation/reference-manual.php`. [Accessed 06-12-2023].

[10] JFlex - the fast scanner generator for Java. `https://jflex.de/`. [Accessed 28-11-2023].

[11] Beaver - a LALR Parser Generator. `https://beaver.sourceforge.net/`. [Accessed 28-11-2023].

[12] Mikael Pettersson. A compiler for natural semantics. In Tibor Gyimóthy, editor, *Compiler Construction*, pages 177–191, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[13] Georgian-Vlad Saioc and Hans Hüttel. A tool for describing and checking natural semantics definitions of programming languages. *Electronic Proceedings in Theoretical Computer Science*, 369:51–66, September 2022.

Appendices

Appendix A
Source code

The source code for our project is available at
`https://github.com/nichobi/thesis-project`,
with the README.md file providing instructions for use.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-01-25

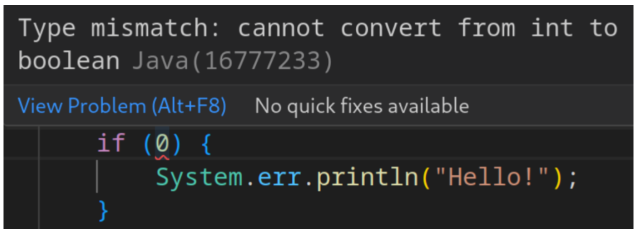
EXAMENSARBETE Type Checker Generation using Reference Attribute Grammars
STUDENT Nicholas Boyd Isacsson
HANDLEDARE Christoph Reichenback (LTH)
EXAMINATOR Niklas Fors (LTH)

Autogenererad typsäkerhet för programmeringsspråk

POPULÄRVETENSKAPLIG SAMMANFATTNING **Nicholas Boyd Isacsson**

Typkontrollerare är ett vanligt verktyg för att hitta buggar i datorprogram, men är relativt komplicerade att skapa. Jag har byggt ett program som automatiskt skapar typkontrollerare för program utifrån en kort definition.

Datorprogram hanterar ofta många olika datatyper, och fel användning av dem är en av de vanligaste buggarna. För att hitta dessa buggar används typkontrollerare, som systematisk går igenom hela programmet och verifierar varenda användning av variabler eller värden för att se till att rätt datatyp använts på rätt plats.



```
Type mismatch: cannot convert from int to
boolean Java(16777233)
View Problem (Alt+F8)    No quick fixes available
    if (0) {
        System.err.println("Hello!");
    }
```

Ett typfel i ett vanligt programmeringsspråk, där en siffra används istället för ett booleskt (sant/falskt) värde.

Dessa typkontrollerare är dock komplicerade att skapa, och är ofta starkt kopplade till ett specifikt programmeringsspråk. Det innebär att varenda programmeringsspråk behöver skapa sin egen typkontrollerare, och leder till en otrolig mängd upprepat arbete. I mitt arbete har jag skapat ett verktyg för att generera typkontrollerare per automatik, utifrån en kort och koncis definition av programmets regler.

Detta görs genom att baka in kod i ett av mellanleden när programmet omvandlas från människoskriven text till maskinexekverbar kod. Mellanledet har en struktur lik ett släktträd där allt härstammar ifrån roten, med grenar som representerar olika kodstrukturer, och som i sig kan delas av i flera grenar.

Därmed kan vi enkelt skapa typkontrollerare för nya språk på ett kortfattat vis, genom att bara definiera vilka regler som ska finnas och inte behöva fundera över hur man implementerar dem. Det underlättar också om man vill göra förändringar i systemet, eftersom det är mycket lättare att ändra de koncisa reglerna än att ändra de många raderna kod som kan krävas för att implementera den.

I nuläget är programmet ganska begränsat i hur komplicerade typkontrollerare det kan generera, och skulle inte kunna appliceras på något av de stora, kända programmeringsspråken. Däremot klarar det flera exempel från forskningslitteraturen, och tillvägagångssättet har dock visats vara flexibelt och utbyggbart. Man borde alltså kunna vidareutveckla programmet för ett bredare stöd i framtiden.