Extending the Domain Specific Language for the Pipelines for Interactive Environments build system

Master's Thesis

Ivo Wilms

Extending the Domain Specific Language for the Pipelines for Interactive Environments build system

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ivo Wilms
born in Pijnacker, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Extending the Domain Specific Language for the Pipelines for Interactive Environments build system

Author: Ivo Wilms
Student id: 4488466

Abstract

Build systems speed up builds by reusing build step outputs from previous builds when possible. This requires precise definitions of the dependencies for build steps. Pipelines for Interactive Environments (PIE) is a build system with precise dependencies, but its task definitions in Java are verbose. The PIE domain-specific language (DSL) allows pipeline developers to write concise definitions of PIE tasks, but the PIE framework has evolved and the PIE DSL cannot express many tasks and projects.

This thesis presents PIE DSL 2, which improves on PIE DSL 1 in three areas. It extends the language itself with a module system, generics and support for suppliers and injected values, which allows it to express more tasks within the DSL. There are four improvements for the code base. The first two are a specification of the static semantics in Statix and a new compiler backend that compiles to an abstract syntax tree (AST) instead of using string interpolation, both of which extend the features for the DSL that can be expressed. The second pair is constructors for semantic types and tests, which improve development speed of the DSL. The final area we improve is the user experience, which we improve by adding documentation for expressions and types in the PIE DSL.

We compare PIE DSL 2 to Java in a case study. Only a single task can be expressed in the DSL, which means that the boilerplate is not reduced. Furthermore, the Java ecosystem has better error detection except for nullability. Finally, the PIE DSL is simpler than Java, but only when the full pipeline is supported by the DSL. We conclude that the DSL is not better than Java for full projects yet, but for tasks that it can express it is a slight improvement over Java. This leads to the hypothesis that it has potential to become better if it can express enough tasks.

Due to time constraints, the case study did not use the latest version of the DSL. In theory the latest version of the DSL can express 11 of the 19 tasks, but this has not been verified experimentally.

Overall, this thesis makes improvements to the PIE DSL and its environment, but that has not translated to the DSL being better than Java.

Thesis Committee:

Chair:	Dr. N. Yorke-Smith, Faculty EEMCS, TU Delft
Committee Member:	Dr. Ir. G.D.P. Konat, Faculty EEMCS, TU Delft
University Supervisor:	Dr. C.B. Poulsen, Faculty EEMCS, TU Delft

Thesis advisor:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft

Preface

This thesis was a massive effort, and has taught me a lot about how to conduct research, planning, and how to apply proper software engineering practices in practice. I could have never completed this thesis project with the same quality it has now and learned as much as I did without the help of a lot of people, and I am extremely grateful for all their time, expertise and support.

Thank you Gabriël for the guidance, feedback and help. Your feedback on my writing and pull requests helped immensely, not just for my thesis but for any future writing I do as well.

A huge thanks to Eelco as well. Your expertise was invaluable, and your untimely passing has left a hole in the community that will never be filled entirely. You will be sorely missed.

Thank you Casper for taking over as my supervisor and for the feedback, which was excellent despite Spoofax being outside your area of expertise.

Thank you Aron and Hendrik for your help with Statix and the fast bug fixes and support for the few bugs I ran into.

And thanks to everyone else in the Programming Languages group for their support during my work on this thesis.

Thanks to Guido and the rest of Oracle Labs for the opportunity to use PIE in a setting outside our development group and the resulting valuable feedback.

Thank you Neil for being the chair of my thesis committee.

Finally, thanks to my friends and family for being there for me. In particular, thanks to my parents for helping figure out my priorities and keeping me on track during my thesis.

Ivo Wilms
Delft, the Netherlands
June 21, 2022

Contents

Preface iii

Contents v

List of Figures vii

List of Tables xi

1 Introduction 1
- 1.1 Build systems . . . 1
- 1.2 Pipelines for Interactive Environments . . . 1
- 1.3 PIE Domain Specific Language . . . 2
- 1.4 Problems with the PIE DSL . . . 5
- 1.5 Contributions . . . 6

2 Problem analysis and background information 7
- 2.1 Problems . . . 7
- 2.2 Background . . . 14

3 PIE Improvements 21
- 3.1 Expressive power . . . 21
- 3.2 Code base . . . 37
- 3.3 User experience: reference documentation for types and expressions . . . 44

4 Evaluation 45
- 4.1 Evaluation questions . . . 45
- 4.2 Case study . . . 47
- 4.3 Answers to evaluation questions . . . 48
- 4.4 Comparison between PIE DSL 1 and 2 . . . 62

5 Related work 75

6 Future work 77
- 6.1 Increase the expressive power of the DSL . . . 77
- 6.2 Improve the code base of the PIE DSL . . . 86
- 6.3 Improve the user experience of the DSL . . . 87
- 6.4 Roadmap . . . 90

7 Conclusion 93

Bibliography **97**

Acronyms **99**

A Case study data **101**

B End-to-end test full files **103**

C A PIE DSL implementation of Unsound.Java **109**

D Ideal DSL code for case study **111**

List of Figures

1.1 An example of a pipeline in PIE DSL 1. Note: this is a simplified example, the main goal is to illustrate what the PIE DSL looks like. Some examples of simplifications: it discards messages from parsing and it only returns messages from the static analysis. It also assumes that parsing does not throw exceptions, but always returns something that the static analysis can accept as input, even when the program text does not parse. 3

2.1 The workarounds required when the first sub-expression of a comparison (a), if-else expression (b) or a list literal (c) is not a supertype of the other sub-expression(s). In each case, the code shows a failing example and workarounds, both in cases where one of the types is a supertype of the other(s), and where none of the types is a supertype of the other(s). 8

2.2 An example of a generic class and how it can be specialized to use it in PIE DSL 1. The qualifier `mb.common.option` for the foreign java definitions has been omitted for conciseness. 10

2.3 A Stratego rule to transform a PIE integer literal AST to a Java Integer. Both versions omit the empty list of statements with side-effects for conciseness. Compiling to AST is more verbose in this case because there is almost no syntax and the constructor includes all elements that are omitted in the template version. 12

2.4 Specifications of the PIE DSL if-else expression in different Spoofax meta-DSLs. *Continued on the next page.* 16

2.4 *Continued from previous page.* Specifications of the PIE DSL if-else expression in different Spoofax meta-DSLs. 17

2.5 An example program and its scope graph. Identifiers have indices to differentiate uses of the same name. The following elements were omitted for clarity or brevity: the module statement and module tree, the full types of the functions, and the instantiated scope and reference from referencing $\texttt{sumList}_1$. 18

3.1 A proposal for standard library functions to access fields 23

3.2 An example of ways to access enum values. *Continued on the next page.* 24

3.2 *Continued from previous page.* An example of ways to access enum values. 25

3.3 A comparison of generics in Java and the PIE DSL. 26

3.4 An example of implicit wildcard bounds. `Result<String, ?>` does not specify a bound for the wildcard, even though an unbounded wildcard is not within the bounds of `E`. This unbounded wildcard is nevertheless valid because it implicitly copies the upper bound from `E`. 27

3.5 An example of a generic data type `Box` with two type parameters `A` and `B`. The indices do not affect the name of an identifier, they are there to make the origin of each identifier in the scope graph clear. The type parameters for the constructor are named `C` and `D` to make it easier to differentiate them from `A` and `B`. 28
3.6 An example of instantiating type parameters when there is subtyping. 29
3.7 The current Java code and the replacement in the PIE DSL for `TigerStyle`. 31
3.8 An example of referencing a task in another module. `x` uses a fully qualified reference of the task. `y` uses a direct import. `z` uses a qualified reference from an imported module. 33
3.9 Three simple imports and the equivalent multi-import 34
3.10 An import with the multi-import in the middle, and its decompositions according to two possible semantics for decompositions of multi-imports. The first decompositions, in Figure 3.10b, does not work, because it creates two imports that both declare a name `c`. To avoid that, the actual semantics defines that the name in the multi-import is used as well. This is shown in Figure 3.10c. 34
3.11 The tree building algorithm. A MODULE is a three-tuple of a fully qualified module name, associated scope, and whether it is a concrete module or a pseudo module. The $addToTree$ function takes a tree constructor and a MODULE, and returns a new tree constructor with the module added. It is called for each module, the updated tree is passed along with each call. $declareInScopeGraph$ takes the final tree constructor and declares the tree in the scope graph. 35
3.12 The import tree algorithm takes all imports in a file, resolves them, declares them, and declares the import tree. 36
3.13 The qualified reference resolution algorithm. The namespace to resolve $name$ in is derived from the syntactic position of the reference. 37
3.14 SPoofax Testing language (SPT) tests for the grammar and static semantics of if-else expressions. 39
3.15 Fragments of the end-to-end test for a call to a generic foreign Java static function. The full files can be found in Appendix B. 40

4.1 PIE DSL code that will generate a note. 47
4.3 The library file that re-declares `org.spoofax.interpreter.terms.IStrategoTerm` from Java in the PIE DSL. 48
4.5 The PIE DSL only gives a warning on appending an empty list if the type is an empty list. 53
4.6 IntelliJ gives a warning for certain cases of appending an empty list. 54
4.7 Examples of duplicate imported names in the PIE DSL and Java. 55
4.8 An example of a correct Java warning suppression annotation that becomes incorrect due to a code change somewhere else. 57
4.9 The imports of `TigerCompleteTaskDef` in Java and in the PIE DSL. 59
4.10 The declaration of `TigerCompleteTaskDef` in Java and the PIE DSL. 60
4.11 The implementation of `TigerCompleteTaskDef` in Java and the PIE DSL. 61
4.13 Three methods for calling a Stratego Strategy. 68
4.14 The function used by `ConstraintAnalyzeTaskDef` to either get an existing `ConstraintAnalyzerContext` or to construct a new one. 70

6.1 Rough drafts of what support for fields could look like. Note that the compiled Java code is the same for both cases. 81
6.2 A rough draft of what support for enums in the DSL could look like. 82
6.3 An example of how to use `comparable-with` to specify that `ArrayList[T]` is comparable with lists that contain subtypes or supertypes of `T`, not just `List[T]`. . . . 84

6.4 An example of a command declaration for Spoofax 3, the current way in `spoofaxc.cfg` and a proposed way using annotations in the PIE DSL. 89

List of Tables

4.2 An overview of the reduction in boilerplate due to switching from pure Java to also using the PIE DSL when appropriate. 'Excluding layout' means that consecutive layout is counted as one character, and lines with only layout are excluded for the line count. Libraries are always excluded for Java, 'including libraries' only applies to the PIE DSL. Note that for the percentages, the baseline value is the total amount in Java. Since this changes when including or excluding layout, the percentage values including and excluding layout have different baselines and cannot be meaningfully compared. 49

4.4 Overview of possible mistakes in Java and the PIE DSL where the detection stage or severity differ. The Java language refers to the Java language as specified by the Java Language Specification. IntelliJ and Gradle refers to the Java ecosystem, which includes IntelliJ, Gradle, the Checkerframework Gradle plugin, and the Java compiler. 51

4.12 An overview of the language features each task in the case study uses. We consider a workaround better than a Java helper method, so if a feature works with either it will be listed as 'workaround'. Lightgray symbols already worked in PIE DSL 1, gray symbols work since PIE DSL 2, and black symbols still do not work. 'Subclassing' means that the task does not directly implement `TaskDef`, but instead extends a class which does. 'Injected values' are values which are added to the `TaskDef` when it is constructed. 'Resource dependencies' are dependencies on resources, such as files, open editors in an Integrated Development Environment (IDE), Uniform Resource Locators (URLs), and resources loaded in the Java Virtual Machine (JVM). Tasks 1 and 2 use `ExecContext`, which cannot be expressed or worked around. 63

A.1 Aggregated data for the Tiger case study. This description uses 'the Java version' to refer to the version of the case study that is fully implemented in Java, and 'the (PIE) DSL version' to refer to the version that also uses the PIE DSL when practical. The fourth column ('diff') and the fifth column ('diff (%)') show the absolute and relative difference between the Java version and the DSL version, with Java as the baseline. The row names are divided into four parts. Lines and characters refers to whether we count the lines or the characters. The language can be Java, PIE DSL or total. Java counts only lines/characters in the Java parts of the case study version, PIE DSL counts the lines/characters in the PIE DSL parts of the version, and total counts both Java and PIE DSL for that version. Libraries are code which is defined in a separate project. This code is never included for the Java version, since it is the exact same for both versions. For the PIE DSL version, libraries refers to foreign Java declarations for types and functions from libraries. The final part of the row names is layout. For both Java and the PIE DSL, layout is whitespace and comments. It includes JavaDoc, since it is unfair to count that in Java when the benefits such documentation bring are not included in the counts for the PIE DSL. Including layout simply counts all layout/characters in a file. Excluding layout means that consecutive layout is counted as one character, and lines with only layout are excluded for the line count. For a detailed explanation of how layout is counted, see section 4.1. Finally, the last section of the table shows task counts. 'Tasks implemented in Java' refers to tasks which are implemented in Java for each version. The tasks implemented in the PIE DSL are subdivided into two catagories: fully implemented in the DSL and implemented with helper functions. Tasks are considered to be implemented with helper functions if they need a function that is implemented in Java, which is specific for that task. . . . 102

Chapter 1

Introduction

1.1 Build systems

A build systems is a program or framework (like Make or Gradle) that builds new software or artifacts by invoking other programs or functions (like gcc or javac, or a function to upload a file to a server).

A build target is a build artifact that should be produced (e.g. a compiled executable) or an action that should be performed (uploading the executable to a server). Common build targets are producing binary executables, running all tests, extracting documentation or publishing the produced artifacts somewhere.

The goal of a build system is to concisely specify the build targets, and to make the build repeatable and reproducible. Repeatable means that running the build multiple times on the same machine will produce the same results. Reproducible means that running the build on a different machine also produces the same result. Note that 'same results' does not mean results which are exactly equal, but results which are equivalent in all areas that we care about. For example, when building artifacts, it does not matter in which order the artifacts are built, as long as all of them are built. Build targets can be split up into multiple build steps (called 'tasks' in Gradle and Pipelines for Interactive Environments (PIE), and 'targets' in Make and Ant). To be concise, steps common to multiple builds can be implemented at a single location and referenced in each of the builds. This avoids code duplication.

A common problem for larger projects is that building the full project from scratch every time takes too long. Build systems solve this with incremental builds. Incremental builds only rebuild intermediate results when necessary to speed up the current build. When rebuilding is not necessary, the build system reuses the cached intermediate result from a previous build. To determine whether rebuilding is necessary, users declare the dependencies for each step. While dependencies for build steps are anything that influences the output of the build step, most build systems restrict the dependencies to specific files or directories, classes of files (e.g. `build/generated/java/**/*.class` to match all `.class` files in `build/generated/java`) and other build steps (packaging Java class files into a jar depends on compiling the Java source files to Java class files).

The build system checks the declared dependencies for each step. If there is a known output for the current state of the dependencies of the build step, it can skip executing the build step and reuse the output. If one of the dependencies is another build step, that build step should be executed or skipped first.

1.2 Pipelines for Interactive Environments

PIE is a build system for interactive environments. This build system consists of an abstract algorithm of a sound and precise build system, the Java framework that implements that

algorithm, and a domain-specific language (DSL) for concisely implementing pipelines in the framework. In this thesis, PIE refers to the Java framework by default, but it may also refer to the DSL if that is clear from the context. This section gives a quick, high-level overview of PIE. For more information, PIE is introduced in Konat, Steindorfer, et al. (2018), and summarized in Konat, Sol, et al. (2019). The algorithm is explained in more detail in Konat, Erdweg, and Visser (2018).

PIE is designed to work in interactive environments such as Integrated Development Environments (IDEs). In particular, it has the following design requirements:

- *Quick builds*
  The user wants feedback while they type. To get this feedback, the file needs to be parsed and analyzed in real time.

- *Incrementality*
  To achieve quick builds, previously computed task outputs must be reused where possible. Ideally, PIE only runs tasks that are outdated, and reuses cached results otherwise.

- *Transparent caching*
  Pipeline developers should not have to think about caching the inputs and outputs of their tasks. Instead, the PIE framework should handle caching automatically, with as little involvement from the pipeline developer as possible.

- *Precise dependencies*
  Underapproximated dependencies are dependencies that exist, but which are not declared. This is unsound, because a change in the dependency will not lead to a rebuild.

  Overapproximated dependencies are dependencies which are declared, but which do not actually exist. This is very common in many build systems. For example, in Make[1], instead of meticulously specifying which files each file depends on, it is common practice to just say 'all `*.c` files depend on all `*.h` files'. This hurts incrementality and therefore the build time, because a change in any header file now requires recompilation of all C files, while there are probably only a few C files that *actually* import that header file.

  Precise dependencies are dependencies which are neither underapproximated nor overapproximated. To achieve precise dependencies without declaring them all up front, PIE uses dynamic dependencies. This means it can incorporate newly discovered dependencies while building. This is an uncommon feature for build systems: most build systems require that all dependencies are declared up front, which necessitates the overapproximation.

1.3 PIE Domain Specific Language

The DSL for PIE is called the PIE DSL. Throughout this thesis, it is also referred to as 'PIE' and 'the DSL'. This thesis developed a new version of the PIE DSL. The DSL at the start of this thesis is referred to as PIE DSL 1, while the version presented in this thesis is PIE DSL 2.

Figure 1.1 shows an example of a simple analysis pipeline in PIE DSL 1. It takes a list of files, parses and analyzes them, and then returns a `CommandFeedback` with messages from analysis.

The first 22 lines are foreign (Java) declarations. These include type declarations with the `data` keyword and function declarations with the `func` keyword. The DSL uses `*` to turn a

[1] `https://www.gnu.org/software/make/`

```
1  data Term = foreign java org.spoofax.interpreter.terms.IStrategoTerm {}
2  func parse(program: string) -> Term = foreign org.example.lang.Parse
3
4  data Throwable = foreign java java.lang.Throwable {}
5  data FileException : Throwable = foreign java org.example.pie.FileException {}
6  func fileException(file: path) -> FileException
7    = foreign java constructor org.example.pie.FileException
8
9  data Message = foreign java mb.common.message.Message {}
10 func message(test: string, exception: Throwable) -> Message
11   = foreign java constructor mb.common.message.Message
12 data MultiFileResult = foreign java mb.constraint.common.ConstraintAnalyzer.MultiFileResult
13 {
14   func getErrors() -> Message*
15   func getWarnings() -> Message*
16   func getNotes() -> Message*
17 }
18 func analyzeMulti(programs: Term*) -> MultiFileResult
19   = foreign org.example.lang.AnalyzeMulti
20 data CommandFeedback = foreign java mb.spoofax.core.language.command.CommandFeedback
21 func commandFeedback(messages: Message*) -> CommandFeedback
22   = foreign java mb.spoofax.core.language.command.CommandFeedback#of
23
24 func check(files: path*) -> CommandFeedback = {
25   val asts: Term* = [{
26       val program: string? = read file;
27       if (program == null)
28         return commandFeedback([message("Could not read $file", fileException(file))]);
29       parse(program!)
30     } | file <- files];
31   val res: MultiFileResult = analyzeMulti(asts);
32   commandFeedback(result.getErrors() + result.getWarnings() + result.getNotes());
33 }
```

Figure 1.1: An example of a pipeline in PIE DSL 1. Note: this is a simplified example, the main goal is to illustrate what the PIE DSL looks like. Some examples of simplifications: it discards messages from parsing and it only returns messages from the static analysis. It also assumes that parsing does not throw exceptions, but always returns something that the static analysis can accept as input, even when the program text does not parse.

type into a list, so `Message*` is a list of messages. Every definition consists of a declaration (before the `=`) and a definition (after the `=`). The DSL makes no distinction between tasks, static methods and constructors in regards to function calls: all of them are declared as functions, and function declarations are always the same by design. Because they have to called differently in the generated Java code, they do have different implementations: PIE tasks implemented in Java use `foreign` (lines 2 and 18-19), Java static methods use `foreign java` (line 21-22), and Java constructors use `foreign java constructor` (lines 6-7 and 10-11). Data type declarations can declare methods, as is shown on lines 12-17.

The first 22 lines do not implement anything new, they only declare types and functions with implementations in Java. The last function `check` on lines 24-33 implements the pipeline. It takes a list of paths, `files: path*`. A `path` is a built-in type, which is more convenient than the regular types that Java and many other languages use, and safer than just using strings. It returns a `CommandFeedback`, which represents feedback in an interactive environment, e.g. an IDE. The definition of the function comes after the `=`. A definition can be any expression,

but in most cases it will be a block, as is the case here. The first expression of the block defines a value `asts`, with type `Term*` (line 25, expression extends to line 30). Type annotations for values are optional, but included in this example for clarity. They are called values, and not variables, because they are immutable. This means that they cannot be re-assigned, but their internal state can still be mutated with method calls or direct field access in Java.

The expression that is assigned to `asts` is a list comprehension over the files (lines 25-30). Each file is read into a string using the built-in `read` expression (line 26). This returns a string on success or null if reading fails. This is reflected in the type of `program`, which uses the question mark in `string?` to signify that it can be null. It checks for null, and returns a `CommandFeedback` with a single error message (lines 27-28). If reading succeeded, it casts `program` to non-nullable using the exclamation mark in `program!` and parses the program (line 29). This parsing happens by calling the `parse` task as a regular function, the fact that it is a task that is incremental is completely transparent to the caller. The list of abstract syntax trees (ASTs) is then analyzed with the `analyzeMulti` task (line 31). Different methods are called on the result of the analysis to get the different kinds of messages, which are then concatenated and wrapped in a `CommandFeedback` (line 32). Finally, the last expression in the block is implicitly returned.

The main goal of the DSL is to make pipeline development with PIE easier. This can be broken up into the following objectives:

- *Catch mistakes earlier in the development cycle*
  Java and the Java ecosystem catch many mistakes statically. However, not every mistake is caught statically. In particular, certain invariants specific to PIE are only checked at runtime by PIE itself. For example, tasks have a unique identifier (ID). This is checked at runtime, but could in many cases be checked statically. Even better, mistakes could be prevented outright by deriving an ID that is guaranteed to be unique. Another example of runtime errors are hidden dependencies. These exist when a task reads a file, and another task later writes to the file. PIE assumes that files are immutable after being written once, so this is not allowed. It is detected at runtime, but it should be possible to detect this statically in some cases.

  There are also some general mistakes that could be caught earlier or more reliably. In particular, nullability remains a pain point in Java, even with plugins that make nullability explicit.

- *Improve readability and simplicity of the code*
  PIE pipelines in Java often have large amounts of boilerplate in the form of a package statement, imports, class declaration, optionally nested classes for input and output, constructor, and method that defines the task ID. All of this can be derived from the function definition, so that is exactly what the PIE DSL does. The DSL also includes syntax for domain related concepts, such as filesystem paths and dependency declarations, and common operations, such as mapping over a list with a list comprehension.

- *Improve automation for pipeline development*
  For example, with editor features like function suggestions.[2] While the Java ecosystem already offers suggestions for normal methods, it does not for PIE tasks, because those are implemented as classes that implement the `TaskDef` or `Task` interface, not regular Java methods.

It is also important to note a non-goal of the DSL: *it is not a requirement that the DSL can express every task.* Java has many language features, and requiring the DSL to express them

[2]Note that function suggestions are not implemented yet, it is just an example of a feature. The PIE DSL lacks editor features because they have not had priority yet.

all means that it basically becomes Java with funky syntax. Instead, the DSL should be able to express most tasks in order to achieve its actual goals, but it is fine if certain tasks that use obscure Java features or patterns cannot be expressed in the DSL. This allows the DSL to stay a lean and simple language and for the language development to focus on the actual goals.

1.4 Problems with the PIE DSL

There are three areas of problems with PIE DSL 1: the expressive power, the code base and the user experience. This section gives a high level overview of the problems in these areas and explains why they are important. The problems are explained in detail in section 2.1.

1.4.1 Expressive power

There are many tasks that PIE DSL 1 cannot express, and it does not support multi-project setups. While the original projects could be implemented, the current use cases use more complicated patterns that require more complicated features. These features are not supported by PIE DSL 1. The DSL cannot meet its stated goals when the tasks are still implemented in Java, so this is a major problem. The tasks cannot be expressed because they use data types, functions or other tasks with a public interface that cannot be expressed in the PIE DSL 1. For example, generic functions and data types, functions that compile to `void`, and data types that do not provide getters but which use public fields. PIE DSL 1 dies a death by a thousand cuts: no feature is used by all tasks, and no single task needs to be implemented in the DSL, but almost every task depends on some elements that use an unsupported feature, so most tasks cannot be expressed in the DSL. While we try to avoid universal statements by using 'most tasks', that is an understatement: none of the 19 tasks in the case study (see chapter 4) can be expressed in PIE DSL 1.

1.4.2 Code base

The code base of the DSL has four problems. First of all, there are no tests. Tests prevent regressions, which means that less time has to be spent on fixing bugs. This speeds up development of the DSL.

The second problem is the meta-language used for the specification and implementation of the static semantics of PIE DSL 1: Name Binding Language 2 (NaBL2) (Antwerpen, Néron, et al. 2016). It does not have the expressive power required to express some of the features mentioned in 1.4.1. It also slows down development due to limited static checks and runtime debugging options, which makes it very tedious to work with. Finally, it is now deprecated in favor of its successor, Statix (Antwerpen, Poulsen, et al. 2018; Rouvoet et al. 2020).

Additionally, the specification of the static semantics of the PIE DSL does not differentiate between syntactic and semantic types. Both of these use the same constructors, which leads to bugs when the syntactic and semantic type are not interchangeable but there is no explicit transformation from syntactic to semantic type.

The final problem with the code base is that the compiler uses string interpolation. This technique for compilation is not best practice, because it is inefficient if we want to optimize the generated Java code.

1.4.3 User experience

The final problem area is the user experience. While there are many possible improvements within this area, this thesis focuses on only one: user documentation. We can say a lot about documentation and why it is important, but instead we will boil it down to a single quote:

"if a feature isn't documented it doesn't exist" – Simon Willison, 12th January 2022.[3] The PIE DSL was not documented at all, so it did not exist. While this is obvious hyperbole, it captures the essence of the problem: if a user does not know how to use a language feature or that the feature exists, that feature is worthless: it does not help the users in achieving their goal.

1.5 Contributions

The work for this thesis includes the following contributions. Except for the evaluation, these contributions can be found in chapter 3.

Language
We introduce PIE DSL 2, which includes the following improvements over PIE DSL 1:

1. A module system
2. Parametric polymorphism AKA generics. This also solves the issue of referencing fields and enum values by way of introspection.
3. Support for suppliers, which represent tasks as values in PIE.[4]
4. Support for injected values in tasks.
5. Least upper bound replaces overly restrictive subtyping constraints
6. Class inheritance, including overriding[5] and method collisions

Code base

7. New compiler backend: now compiles to AST instead of using string interpolation
8. Tests for the grammar, static semantics and the compiler
9. Static semantics implementation (i.e., name binding and type checking) in the Statix meta-DSL.
10. Constructors for semantic types

User experience

11. User documentation for types and expressions

Evaluation

12. We evaluate PIE DSL 2 by comparing it to Java in a case study on a Spoofax 3 Tiger pipeline in chapter 4.
13. We compare PIE DSL 2 to PIE DSL 1 in section 4.4.

[3] From the blog post 'How I build a feature',
`https://simonwillison.net/2022/Jan/12/how-i-build-a-feature/`.

[4] While suppliers can represent arbitrary computations, not just tasks, the PIE DSL does not support general higher order functions. The only way to obtain suppliers in the DSL is by creating one from a task, from a method implemented in Java, or by wrapping an already computed value.

[5] The overriding does not take overloaded methods into account yet, so overloading a method in a subclass shadows the original method in the super class.

Chapter 2

Problem analysis and background information

This chapter describes each of the problems mentioned in the introduction in more detail. It also provides background information on Spoofax and scope graphs. Spoofax is the language workbench used to implement the PIE DSL and the environment where we conduct our case study. Scope graphs are used to model name resolution by both NaBL2 and Statix, the Spoofax DSLs in which the static semantics of the PIE DSL are implemented.

2.1 Problems

The problems with PIE DSL 1 fall into three categories. There are problems with the language itself, problems with the language implementation, and problems with the user experience. For the user experience, we only discuss one problem: a lack of user documentation.

2.1.1 Problems with the PIE DSL

One of the main problems of PIE DSL 1 is a lack of expressive power. PIE DSL 1 is used in Konat, Steindorfer, et al. (2018) for the implementation of a minimal language pipeline that generates a parse table and uses it to parse an example file. However, it turns out that this minimal language pipeline does not scale to larger pipelines while staying simple. The PIE framework has since become more complicated, and PIE DSL 1 is no longer expressive enough to interoperate with the more complicated tasks in the PIE framework. The PIE DSL becomes completely useless if it cannot express tasks, so this is a major problem.

PIE DSL 1 has a few areas with insufficient or unsatisfactory expressive power. First of all, if-else expressions, list literals and comparisons all have overly restrictive subtyping constraints. Secondly, PIE DSL does not support references to fields and enum values. Furthermore, many tasks now use `Result` as return type, but that is a generic class and cannot be expressed in PIE DSL 1. Suppliers are used as parameter type, and these can also not be expressed. Injected values are used as well but are also inexpressible. Finally, PIE DSL 1 does not support class inheritance.

Overly restrictive subtyping constraint in list literals, if-else expressions and comparisons

If-else expressions, list literals and comparisons all have a TODO to use least upper bound (lub) to determine their type.[1] This is because NaBL2 cannot express least upper bound. It also cannot express the constraint 'T1 is a subtype of T2 *or* T2 is a subtype of T1', because it

[1] This turned out to be false for comparisons, they should use the greatest lower bound (glb) instead.

```
someApple == someFruit;   // (1) fails
someFruit == someApple;   // (2) works

someApple == someOrange;  // (3) fails
val theApple: Fruit = someApple;
theApple == someOrange;   // (4) works
```

(a) Workarounds for the overly restrictive subtyping constraint on comparisons. Comparing an apple to fruit in (1) fails, because `Fruit` is not a subtype of `Apple`. To make it work, (2) swaps the two sub-expressions. This solution fails in (3), where neither sub-expression is a subtype of the other. A standard workaround for this issue is shown in (4). It upcasts the element by assigning it to a value with an explicit type.

```
if(cond) fruit else apple;    // (1) fails
if(!cond) apple else fruit;   // (2) works

if(cond) pear else apple;     // (3) fails
val theApple: Fruit = apple;
if(cond) pear else theApple;  // (4) works
```

(b) Workarounds for no least upper bound on if-else expressions. (1) fails because `Fruit` is not a subtype of `Apple`. The workaround in (2) negates the condition and swaps the branches. This fails in (3) because neither is a subtype of the other. The workaround in (4) upcasts `apple` to `Fruit` by assigning it to a value with an explicit type.

```
[apple, fruit, pear];         // (1) fails
[fruit, apple, pear];         // (2) type checks, but changes list order
[apple] + [fruit, pear];      // (3) fails (cannot concatenate Apple* and Fruit*)
[apple] + fruit + pear;       // (4) fails (cannot append Fruit to Apple*)
[] + apple + fruit + pear;    // (5) error: cannot add Apple to top*
val list: Fruit* = [];        // (6a) allowed due to special (unsound) case
list + apple + fruit + pear;  // (6b) error: cannot add Apple to Fruit*
val theApple: Fruit = apple;
[theApple, fruit, pear];      // (7) only workaround that works

[apple, orange, pear];        // (8) fails
[theApple, orange, pear];     // (9) works
```

(c) Workarounds for no least upper bound in list literals. (1) fails because `Fruit` is not a subtype of `Apple`. The workaround in (2) resolves this by making `fruit` the first element of the list, and every other element of the list is a subtype of `Fruit`. However, this does change the order of the list. Attempted workaround (3) gets around the issue by splitting the list into two lists where the first element is a supertype of all other elements, but still fails because a list of `Apple` cannot be concatenated with a list of `Fruit`. Attempted workaround (4) also splits the list, but now runs into a similar issue with appending a `Fruit` to a list of `Apple`. (5) appends to an empty list. This fails because appending a subtype to a list is not allowed due to the implementation of the add operator. (6) tries to resolve this issue by explicitly setting the type of the empty list to a list of `Fruit`, and then adding the elements one by one. Surprisingly, assigning the empty list with as type a list of the top type to a value with as type list of `Fruit` is allowed. However, it still fails in (6b) with the same error as (5): we cannot append a subtype. (7) uses the common workaround of upcasting the first element to the required supertype, and then using that variable. (8) shows that it also does not work when none of the elements is a supertype of the others. Finally, (9) shows that the common workaround works in this case too.

Figure 2.1: The workarounds required when the first sub-expression of a comparison (a), if-else expression (b) or a list literal (c) is not a supertype of the other sub-expression(s). In each case, the code shows a failing example and workarounds, both in cases where one of the types is a supertype of the other(s), and where none of the types is a supertype of the other(s).

cannot handle disjunctions. As a way to still get some type checks, these expressions use the type of an arbitrary sub-expression, and check that the other sub-expression(s) are subtypes. This is illustrated in (1) in the sub-figures of Figure 2.1. That same figure also shows some of the problems with and workarounds for these type checks.

For comparisons, both sub-expressions can almost always be swapped to satisfy the subtyping constraint. There are two exceptions to this common fix. First, when both sub-expressions have side effects and the order of these side-effects matters, they cannot be swapped. Secondly, in the case where both sub-expressions are not a subtype of the other, swapping them does not satisfy the subtyping constraint. The example in 2.1a shows a comparison of apples and oranges, which are of course always unequal. However, types without overlapping values can still be equal to each other. An example is Java lists, which are specified to be equal when the elements of the list are equal, regardless of the specific implementing classes of the lists. This means that types that look like they should never be equal can still be equal, such as a `Stack<String>` and a `Vector<Integer>`, which are equal if both are empty. In case swapping the sub-expressions does not work, there is a standard workaround which works for comparisons, if-else expressions and list literals. The first element is upcasted by assigning it to a value which is explicitly typed as the common supertype. This value is then used instead of the original expression.

Workarounds for if-else expressions are shown in 2.1b. It is possible to swap the branches, but that requires negating the condition, which likely makes it slightly harder to reason about. The standard workaround of upcasting using an explicitly typed variable works here too.

Finally, the workarounds for list literals are shown in 2.1c. List literals are ordered, so lists cannot be arbitrarily reordered so that the first element is a supertype of all the others. This means that (2) type checks, but it changes the meaning of the code. (3) to (6) show examples of things one might try to get around the subtyping, but which all fail due to limitations in the type system. (5) fails in an unexpected way. The type of an empty list is the list of top type, so one might expect that everything can be appended to this list. However, functions in NaBL2 cannot perform arbitrary computation, they can only pattern match on the input values. Due to this limitation, appending is only allowed when the element type is exactly equal to the list element type. This means that subtypes are not allowed, so (5) fails. (6) also fails in an unexpected way. One might expect that (6a) fails, because assigning a list of the top type to a list of `Fruit` is in principle unsound. However, there is no way to create a list of the top type except for the empty list, so it is a little hacky but it works out. Therefore, this is allowed by a rule that explicitly sets the list of the top type as subtype of any list type. Unfortunately, (6) still fails in (6b) because appending a subtype to a list is not allowed. Only the standard workaround works here, as is shown by (7) and (9).

Overall, a lack of least upper bound in these elements makes it harder to write code that the type system will accept. It also makes the resulting code longer and harder to understand.

References to field and enums

Instance fields are common on data classes, static fields are the standard to hold constants, and enums are often used to represent different configuration options, so the need to reference any of these can be expected in pipelines. However, there is no way in PIE DSL 1 to refer to any of them. The current workaround is to create a custom getter method, but that is inefficient, inelegant and annoying. There is no general workaround outside the DSL because PIE DSL 1 also does not support generics, which would be required for such generic getter methods. While this problem has a workaround and thus does not completely prevent using the DSL, it does make it far more annoying and verbose than Java.

```
data Option[T] = foreign java Option {
  func get() -> T?
  func getOr(default: T?) -> T?
}
```

(a) An example of a class with a type parameter, which is not allowed in PIE DSL 1.

```
data IntOption = foreign java Option {
  func get() -> int?
  func getOr(default: int?) -> int?
}
```

(b) An example of a generic class that has been specialized to a specific type argument. This definition is legal in PIE DSL 1.

Figure 2.2: An example of a generic class and how it can be specialized to use it in PIE DSL 1. The qualifier `mb.common.option` for the foreign java definitions has been omitted for conciseness.

Generics

The PIE framework is under active development. This sometimes adds new requirements on things to express using the PIE DSL.

When PIE DSL 1 was designed, the PIE framework used to throw exceptions to signal pipeline failure. Checked exceptions either need to be declared or caught, but they cannot be declared generically. They therefore do not interoperate well with higher order functions. The PIE framework fundamentally uses higher order functions by running tasks[2], so checked exceptions in Java hurt the composability of tasks. This can be avoided by using unchecked exceptions, but then the type system will not check that all exceptions are handled. Throwing exceptions to signal pipeline failure also has some semantic issues: exceptions are meant to signify an exceptional condition. Malformed input is not unexpected in an editor environment where users are actively working on the code, so this should not lead to an exception. The PIE framework solved this by encoding the status of a task in its return value. While a task can encode this in any way it likes, the standard way in PIE is to use the provided library class `Result<T, E>` that encodes either success with a normal value `T` or failure with an exception `E`. These results are composable: a task can take a result and either do some computation if it has a value, or just pass along the exception if it had an exception. This allows easy composition of tasks, where one only needs to check the final result for success or failure.

While this undoubtedly improves the PIE framework, it is a problem for PIE DSL 1: it does not support generic classes like `Result<T, E>`, `Supplier<T>`, `Option<T>`, `ListView<E>` and `BaseCollectionView<E, C extends Collection<? extends E>>`. Because foreign Java methods are declared in a PIE program with PIE DSL syntax, it is barely possible to interface with generic classes. For example, Figure 2.2a shows the generic class `Option`, which is not supported in PIE DSL 1. The only thing that can be declared is a data type that is specialized for a particular type argument, as is shown in Figure 2.2b. That is not enough for classes that are used with different type arguments, such as these library classes. Besides, these are not the only generic classes. Users may have their own generic classes, which PIE DSL 1 also does not support. In short, PIE DSL 1 does not support known generic classes from the PIE library, and it also does not support arbitrary user-defined generic classes.

Suppliers

The PIE framework defines `Supplier`, which allows passing tasks and other computations as values. Suppliers are anonymous, zero-argument functions, and are equivalent to Java `Supplier`, except that they are `Serializable`, which is required for inputs to PIE tasks. Suppliers are useful for incrementality because they potentially allow for quicker checks if some-

[2]Tasks are functions in the context of higher order functions, so the PIE `require` method is a higher order function.

thing is up-to-date by performing the check on the input of the previous task. For example, a parse task could take a string, which is obtained by reading a file. Instead of checking whether the string is still equal, which can be a lengthy operation for large files, we know that the string almost certainly changed when the file does. By using a supplier of the string from the reading task, the parse task no longer has to check the entire file contents and compare it to the value stored in the cache, but can instead compare the file itself, which can be as simple as the last modified date.

PIE DSL 1 does not support suppliers: not only are they type-parametric over their return type, which is not supported in general as explained in 2.1.1, the PIE DSL also does not support higher order functions, so it has no way to pass a task as a supplier.

Injected values

Not all values that are required to execute a task can be a regular argument. Some values are not serializable or not immutable, and they cannot be used as task arguments. These values are not passed as regular arguments when the task is executed but are fields in the task. They are passed to the task constructor in a pattern called *dependency injection*. An example is the `StrategoRuntime` that is used to execute Stratego strategies. This runtime is specific to each language and is added to the task as an instance variable.

PIE DSL 1 does not have any way to specify such an instance variable. We would like to extend the language with new syntax, semantics and code generation to handle these injected values.

Class inheritance

Even though it is possible to declare supertypes for data types, those supertypes are only used for subtyping. The methods declared in the supertype are not available on the subtype. In Java, the semantics are that a method with the same name and a override equivalent signature in the subtype overrides the definition in the supertype. If two methods have the same name but the signatures are not override equivalent, it is an overload instead. In Java, methods can also collide due to type erasure. Type erasure removes the type parameters from types, so `List<String>` and `List<Integer>` both become the raw type `List`. This happens because the JVM does not know about generics for backward compatibility reasons. A collision occurs when two methods have signatures which are not override equivalent before type erasure, but which are override equivalent after type erasure.

Lack of modularity

Finally, lack of modularity is a problem which straddles the line between a specification issue and an implementation issue. There are two problems that fall under a lack of modularity: multi-file PIE projects and name collisions.

Multi-file projects The first problem is caused by the fact that it is impossible to refer to other files in PIE DSL 1. Because we want each Spoofax language project to have its own PIE file, multi-project setups are not supported. Multi-project setups are unavoidable. For example, compilers often need at least two language projects: a language project with the target language and a language project with the source language. The compiler code itself is then either included in the source language or defined in a separate third language project. This is not merely an implementation detail: it is unspecified what should happen when there are multiple PIE files.

```
p2j-exp: IntLit(i) -> $[new Integer([i])]
```

(a) A Stratego rule to transform a PIE integer literal to a Java Integer using string interpolation. It uses the variable `i` and interpolates it into a template of Java code.

```
p2j-ast-exp: IntLit(i) -> NewInstance(None(), [], Id("Integer"), [], None(), [Deci(i)])
```

(b) A Stratego rule to transform a PIE integer literal to a Java Integer AST. It uses a `NewInstance` constructor, with no method type arguments, no annotations, the name `Integer` (unqualified), no class type arguments and the single formal argument `Deci(i)`, where `i` is a string representing a decimal number.

Figure 2.3: A Stratego rule to transform a PIE integer literal AST to a Java Integer. Both versions omit the empty list of statements with side-effects for conciseness. Compiling to AST is more verbose in this case because there is almost no syntax and the constructor includes all elements that are omitted in the template version.

Name collisions A simple solution to multiple PIE files is to simply declare everything in the global scope. This leads directly to the second problem with modularity: name collisions. If everything is declared in the global scope, everything needs a unique name. However, it is unlikely that everything has distinct names if it was not explicitly designed to have unique names. For example, many language projects will likely use names such as `parse`, `analyze` and `compile`. The workaround is to prepend the name of your language to the tasks, like `tigerParse`. However, adding such a qualifier for every task and datatype is annoying and should not be necessary to prevent name collisions in the future, it should only be necessary when there is an actual name collision here and now. In conclusion, we want a principled way to refer to other PIE files and a way to modularize groups of related code such as Spoofax language projects.

2.1.2 Problems in the code base

Besides problems in the expressive power of PIE DSL 1, there are also several problems with the code base. The main goal is to let users implement pipelines, and this goal does not consider the codebase quality. However, a bad codebase slows down development considerably, because it makes it harder to make changes, results in more bugs which then take time to fix later, and makes the code difficult to work with, which affects morale of the language developers and ultimately also slows down development. Ultimately, a bad codebase results in less features and slower bugfixes for users, so problems in the codebase are important and should be addressed. This section lists four problems with the code base for PIE DSL 1: the compiler uses string interpolation, there are no tests, NaBL2 limits the static semantics, and there is no differentiation between syntactic and semantic types.

Compiler

PIE DSL 1 has two compilers. The first one compiles to Kotlin. Kotlin is a language that gets compiled to bytecode for the Java Virtual Machine (JVM), which means that it can interoperate with Java and thus with the PIE framework. Oracle did not want to introduce Kotlin to their environment and asked us to compile directly to Java.

The second compiler compiles to Java using string interpolation. String interpolation[3] takes variables and interpolates them into a template of the code to be generated. While this

[3]Note that many languages have a concept that is also called string interpolation, which allows adding a variable into a string without leaving the string syntax (e.g. `"Hello $name!"` instead of `"Hello " + name + "!"`). For code generation in compilers, it is called string interpolation regardless of whether the implementation uses string interpolation or string concatenation to actually get the variable in the string.

sounds reasonably sophisticated, it just means that there is a string that we concatenate values into: `String code = "System.out.println(\"Congrats "+ name + ", you interpolated a variable into a template!\");";`. An example of string interpolation in the PIE DSL 1 compiler is shown in Figure 2.3a. It takes the variable `i` from the input PIE AST `IntLit(i)`, and simply uses it in a string of Java code.

String interpolation has several problems. First of all, it is error-prone because the Stratego compiler cannot check that the string would form valid Java code, which means that typos do not get caught until Java compile time. The second issue is that it is inefficient to do post-processing on the generated Java code when that code is represented as a string. The generated Java string needs to be parsed before it can be used. So post-processing after string interpolation means compiling to a string, parsing that string to a Java AST, doing post-processing, and then printing to a string once again. It is more efficient to compile directly to a Java AST, do post-processing, and then print it to Java only once. Figure 2.3b shows an example of a rule that compiles directly to an AST. Compiling to an AST also somewhat solves the first issue. The Stratego compiler can catch typos in constructor names, but it does not check that the generated AST is a valid Java AST.[4]

All this leads to two requirements for the compiler: it needs to compile to Java or bytecode and it needs to compile to an AST.

Tests

Tests provide confidence that a language feature, bugfix or refactoring has been implemented well and prevent introducing regressions in the future. Testing for PIE DSL 1 is performed by manually compiling a large PIE file and checking that that does not give any errors. This is suboptimal for a multitude of reasons. First of all, not all features are tested this way, which means that regressions can occur in the features that are not tested, assuming they were implemented correctly in the first place. Secondly, this does not test each feature in isolation. A common issue is that code generation does not include all required imports, which is not caught because the import is generated by another element. Next, manually compiling a certain file is not a standardized way of testing and it is undocumented, so a new person would not know how to test the language. Furthermore, this manual test requires manually checking that the generated Java code is correct. This check is often skipped or forgotten, which leads to a lot of errors in the generated Java code. Additionally, if the file fails to compile, it is unclear where it failed without adding debugging statements to the compiler. Also, building the language with Gradle does not run any tests, so any issues from differences between the editor and Gradle (e.g. different Java or Spoofax version) are not caught immediately, but only much later, when the buggy feature happens to be used. Finally, the lack of good tests diminishes the confidence that a refactor will not accidentally introduce a regression, which means that small improvements to the code are often not implemented.

NaBL2 limits static semantics

Spoofax divides language specifications up into several domains, each with its own meta-DSL. Name binding and the static semantics of PIE DSL 1 is specified in NaBL2. NaBL2 is a declarative language which uses constraints to specify the static semantics of a language. It divides constraint gathering and solving into two distinct phases, that is, it first gathers all constraints and after that it solves them.

NaBL2 has several limitations that impeded development of the PIE DSL. First of all, the strict separation of gathering constraints and solving constraints makes it impossible to implement generics, which would be the most powerful solution to interoperating with generics in Java. NaBL2 also has two kinds of edges in the scope graph (regular edges and import

[4]Stratego 2 does check this, but is not used in the PIE DSL.

edges) with defined semantics, which prevents implementing import renaming. Furthermore, NaBL2 has limited static checks besides parsing and limited ways to provide debugging information, so development is tedious and slow.[5] Finally, NaBL2 is now deprecated, so it has no further development and maintenance is low priority.

No differentiation between syntactic and semantic types

The NaBL2 specification for the PIE DSL 1 does not differentiate between syntactic types and semantic types. Using a syntactic type where a semantic type is required works in many cases because often the syntactic and semantic type are equal. In some cases however (in particular custom data types, but also recursive types such as lists and tuples), the kinds are different, and require an explicit tranformation from syntactic to semantic. Such tranformations are performed ad-hoc when it is determined that they are required, but there is no principled location where this happens. In many cases the transformation is left implicit, which works with simple types during testing but breaks when the more complex types are used. Combined with lack of static analysis in NaBL2 and the difficulty of debugging, it often takes some time to find the error, determine that it is because something is still a syntactic type but should be a semantic type, and to find where it should have been tranformed to a semantic type.

2.1.3 User experience of the DSL: no user documentation

User documentation is important because it allows users of the PIE DSL to understand how to use the features of the DSL. It also allows users to discover features they did not know about, can show how to do certain things (e.g. how to set up the Gradle Spoofax plugin so that it works with PIE DSL), and can explain the reasoning for certain language choices. PIE DSL 1 does not have any user documentation.[6]

Without user documentation, users have four options. The first option is to try something and see if it works, which is inefficient and frustrating if it does not work within the first few attempts. The second option is to look at an example and try to replicate that, which is contingent on there being examples and knowing where to find them. Next, a user might ask the community for help. This can certainly be effective, especially for small, easily explained problems, but often that means waiting until someone reacts to your question.

The final option is to look at the implementation. In the case of the PIE DSL, the formal specification is grouped by processing step instead of by language feature. To learn about lists from the specification, one has to look up the syntax in SDF3, the static semantics in NaBL2, and what they are in the Java documentation, after looking up what Java code is generated in Stratego. Translating the implementation to the actual behavior of a language feature is cumbersome but often doable, thanks to the declarative nature of Spoofax metalanguages. Trying to understand the behavior of abstract concepts, such as nullability, is far more difficult.

Overall, the easiest option after just trying the first few solutions that come to mind is to read the documentation, so the PIE DSL should really have that documentation.

2.2 Background

This section explains the context for this thesis. *Spoofax* (2.2.1) explains the Spoofax ecosystem and is generally applicable to this work. *Scope graphs* (2.2.2) explains scope graphs, which

[5] Development was slow partially due to reusing syntactic types as semantic types, which is bad practice. Nevertheless, this would have had far less impact if the editor statically checked the code.

[6] Although nowadays it can be pieced together by reading the documentation of PIE DSL 2 and ignoring the new systems like modules and generics

are relevant for the implementation notes of generics and the module system.

2.2.1 Spoofax

Pipelines in Spoofax are one of the main use cases for the PIE DSL and the PIE framework. Spoofax (Kats and Visser 2010) is a language workbench, that is, an IDE for developing programming languages, in particular DSLs. The idea behind Spoofax (and language workbenches in general) is that most compiler components and editor services for languages are more or less the same for languages. So, they can be mechanically derived from the specification of a DSL. Spoofax has several meta-DSLs for specifying DSLs:

1. Syntax Definition Formalism 3 (SDF3) (Souza Amorim and Visser 2020) specifies the grammar. Spoofax derives signatures for ATerm sorts and constructors, a parser, styler, tokenizer, prettyprinter, placeholders and code completions from the grammar.
2. NaBL2 (Antwerpen, Néron, et al. 2016) or Statix (Antwerpen, Poulsen, et al. 2018; Rouvoet et al. 2020) specifies the name binding and static semantics. Spoofax uses this for reference resolution and showing type information when hovering over an expression.
3. The 'Flow Analysis Specification Language' FlowsSpec (Smits and Visser 2017; Smits, Wachsmuth, and Visser 2020) specifies data-flow analysis, which allows expressing analyses like live variable analysis and constant analysis.
4. Stratego (Visser, Benaissa, and Tolmach 1998) specifies rules and strategies for code transformations, such as optimizations and code generation.
5. SPoofax Testing language (SPT) (Kats, Vermaas, and Visser 2011a; Kats, Vermaas, and Visser 2011b) specifies black box tests for language specifications.
6. PIE DSL specifies the pipelines of a language project.
7. Editor Service (ESV) and the CFG language specify configuration and editor services such as actions in menus, syntax highlighting, on-save handlers, and the file extension for the language.

These meta-DSLs are partially specified in Spoofax as well, and there is an ongoing effort to specify all of them fully in Spoofax. The PIE DSL is specified in its entirety in SDF3 and NaBL2 for PIE DSL 1 and Statix for PIE DSL 2. The compiler for the PIE DSL is implemented in Stratego. Figure 2.4 shows an example of how the Spoofax meta-DSLs are used to specify PIE DSL if-else expressions.

```
Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>
```

(a) The SDF3 production for if-else expressions. `Exp` is a sort and `IfElse` is a constructor of that sort. The right hand side of the production is a template, which means it not only defines the sub-elements of an if-else expression but also defines the layout for pretty-printing, i.e. that there is no space between `if` and the opening bracket `(` and between the brackets and the expression within the brackets, but there are spaces around the other expressions and keywords. This production is used to derive a constructor `IfElse` of sort `Exp` with three parameters, which is used in the other meta-DSL fragments.

```
[[ e@IfElse(exp_cond, exp_true, exp_false) ^ (s, ty_func_def) : ty_false ]] :=
  [[ exp_cond ^ (s, ty_func_def) : ty_cond ]],
  ty_cond <? BoolTy() | error $[Type mismatch: expected boolean type, got [ty_cond]] @
      exp_cond,
  [[ exp_true ^ (s, ty_func_def)  : ty_true ]],
  [[ exp_false ^ (s, ty_func_def) : ty_false ]],
  // TODO: this should calculate the LUB of the expression types
  ty_true <? ty_false | error $[Type mismatch: expected [ty_false], got [ty_true]] @ e.
```

(b) The NaBL2 definition for if-else expressions, including one of the TODOs mentioned in 2.1.1: *Overly restrictive subtyping constraint in list literals, if-else expressions and comparisons*. It uses the `IfElse` constructor defined by the SDF3 production in 2.4a. By returning `ty_false`, the type of the false branch is not just propagated back up the call stack for the type system, but also saved on the `IfElse` AST term, which can then be used by the compiler.

```
typeOfExp : scope * Exp -> (scope * TYPE)
typeOfExp(s1, exp) = ty@typeOfExpImpl(s1, exp) :-
  @exp.type := ty.

typeOfExpImpl : scope * Exp -> (scope * TYPE)
typeOfExpImpl(s, IfElse(exp_cond, exp_true, exp_false)) = (s, ty) :-
  {s_if ty_true ty_false}
  expectAssignableTo(s, exp_cond, BoolType(), ExpressionKind()) == s_if,
  typeOfExp(s_if, exp_true) == (_, ty_true),
  typeOfExp(s_if, exp_false) == (_, ty_false),
  lub(ty_true, ty_false) == ty.
```

(c) The Statix definition for if-else expressions. It uses the `IfElse` constructor defined by the SDF3 production in 2.4a. `@exp.type := ty` saves the type of the expression on the AST term, which can then be looked up by the compiler.

Figure 2.4: Specifications of the PIE DSL if-else expression in different Spoofax meta-DSLs. *Continued on the next page.*

```
p2j-ast-exp:
  e@IfElse(condExp, trueExp, falseExp) -> result
  with
    (condStmts, condVal)   := <try-p2j-ast-exp> condExp
  ; (trueStmts, trueVal)   := <try-p2j-ast-exp> trueExp
  ; (falseStmts, falseVal) := <try-p2j-ast-exp> falseExp
  ; resultName             := <newname> "ifResult"
  ; trueBlockStmts         := <concat> [trueStmts, [ExpressionStatement (Assign(
      ExpressionName(Id(resultName)), trueVal))]]
  ; falseBlockStmts        := <concat> [falseStmts, [ExpressionStatement  (Assign(
      ExpressionName(Id(resultName)), falseVal))]]
  ; ty                     := <p2j-ast-type-sig> <pie-ast-type> e
  ; stmts                  := <concat> [
    condStmts,
    [
      LocVarDeclStm(LocalVarDecl([Final()], <java-classType-to-unannType> ty, [VariableDecl
          (Id(resultName))])),
      IfElse(condVal, Block(trueBlockStmts), Block(falseBlockStmts))
    ]]
  ; result                 := (stmts, resultName)
```

(d) The Stratego rule for compiling if-else expressions to a Java AST. It uses the `IfElse` constructor defined by the SDF3 production in 2.4a, and the type defined by the static analysis of NaBL2 in 2.4b or Statix in 2.4c.

```
test ifelse [[ if (cond) "yes" else "no" ]] parse succeeds
test ifelse missing keyword if [[ (cond) "yes" else "no" ]] parse fails
test ifelse missing keyword else [[ if (cond) "yes" "no" ]] parse fails
test ifelse no layout after if [[ if(cond) "yes" else "no" ]] parse succeeds
```

```
test ifelse [[ [[ if (value == 10) "hello" else "world" ]] ]]
  analysis succeeds run pie-ast-type on #1 to StrType()
test ifelse condition not boolean [[ if ([["true"]]) 1 else 5 ]] 1 error at #1
test ifelse true branch null [[ [[ if (value == 10) null else "a string" ]] ]]
  analysis succeeds run pie-ast-type on #1 to NullableType(StrType())
```

(e) A small sample of SPT tests for if-else expressions.

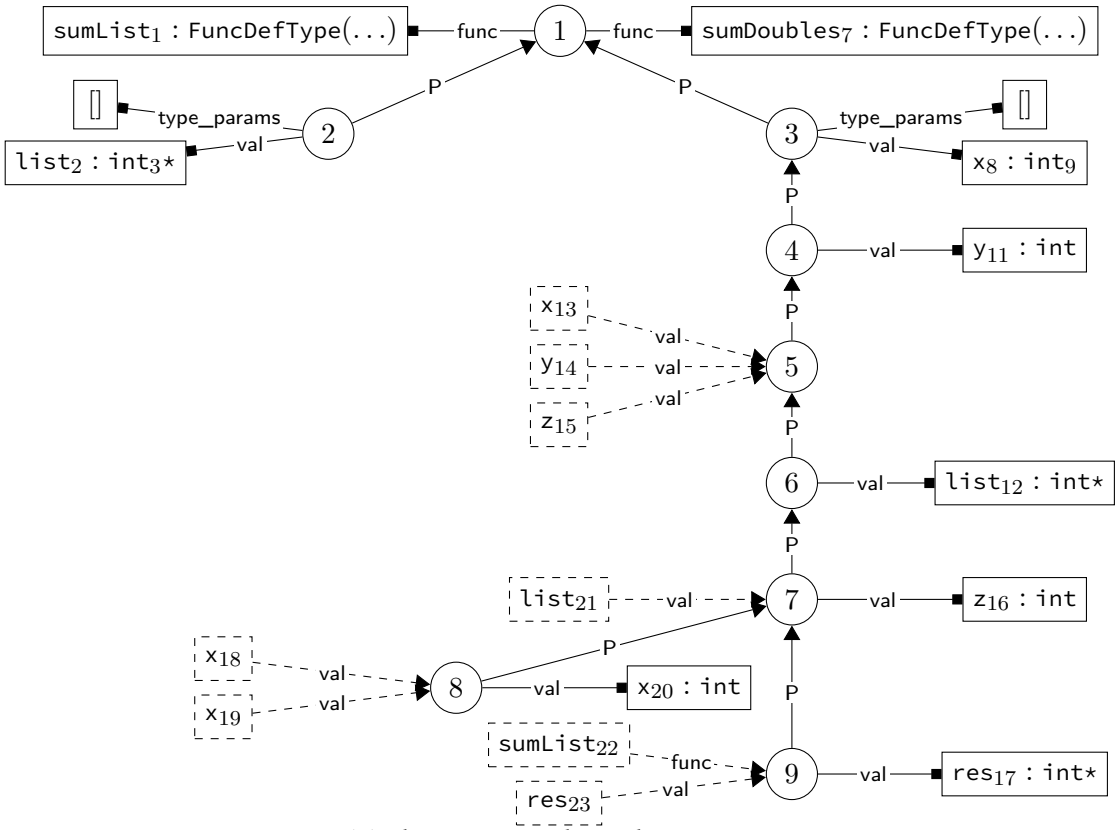
```
func fizzbuzz(num: int) -> string =
  if(divisibleBy(num, 15))
    "Fizz buzz"
  else if(divisibleBy(num, 3))
    "Fizz"
  else if(divisibleBy(num, 5))
    "Buzz"
  else
    "$num"
```

(f) An example of if-else expressions used to express the fizzbuzz problem in the PIE DSL.

Figure 2.4: *Continued from previous page.* Specifications of the PIE DSL if-else expression in different Spoofax meta-DSLs.

```
func sumList1(list2: int3*) -> int4 = foreign java Example5#sumList6
func sumDoubles7(x8: int9) -> int10 = {
  val y11 = 2;
  val list12 = [x13, y14, z15]; // error: z is used before it is declared
  val z16 = 3;
  val res17 = [x18 + x19 | x20 <- list21];
  sumList22(res23)
}
```

(a) An example PIE program. There is a use-before-declaration to show how that is represented by the scope graph.



(b) The scope graph for the program.

Figure 2.5: An example program and its scope graph. Identifiers have indices to differentiate uses of the same name. The following elements were omitted for clarity or brevity: the module statement and module tree, the full types of the functions, and the instantiated scope and reference from referencing `sumList`$_1$.

2.2.2 Scope graphs

Name resolution is the process of matching references to their declarations. Scope graphs are a model to represent name bindings of a program by associating each declaration with a scope, and scopes with each other. Scope graphs were introduced in Néron et al. (2015). Both Statix and NaBL2 use scope graphs as a representation of scoping and name binding of a program. Since NaBL2 is deprecated and Statix uses a simpler version of scope graphs, we will be explaining the scope graphs as used by Statix.

Scopes Figure 2.5 shows an example PIE program and its scope graph. Scope graphs have scopes, which are the vertices of the graph. In the Figure, scopes are represented as circles with a number in them. Scopes in scope graphs often correspond to lexical scopes or

namespaces in the program, but they do not have to match one to one. In the example, each declaration of a value introduces a new scope. These scopes do not correspond exactly to a lexical scope, they represent logical scopes. In this case, they are used to enforce that a name cannot be used before it is declared.

Edges Scopes are associated with each other by labeled, directed edges. Edge labels do not have inherent meanings within a scope graph, the meaning of labels is defined by the language. PIE has `P` edges (short for 'parent'), which go from one lexical scope to its enclosing lexical scope, from a file scope to the root scope, and from the scope of an instantiated datatype to the scope of its generic definition.

Declarations Declarations in Statix scope graphs are defined by named relations, which relate a scope to a value. Like edge labels, the meaning of relations is language dependent, but many lexical languages show similarities. Often, the value is a pair of a name and a value. If a scope `s`, a name `id` and a value `val` is in the relation `rel`, that often represents that `val` is defined in `s` with name `id` under the namespace `rel`. In this case, a namespace is not a language-user defined namespace such as C++ namespaces, but a language level namespace such as the namespace for functions, variables or types. As an example, in Figure 2.5b, the relation `func` has two entries, both from scope 1, with the values `("sumList`$_1$`", FuncDefType([list`$_2$ `: int`$_3$`], int`$_4$`))` and `("sumDoubles`$_7$`", FuncDefType([x`$_8$ `: int`$_9$`], int`$_{10}$`))`. Relations can also be used as a field on a scope, e.g. to set the name of a datatype (if types are represented as scopes), the ordered list of type parameters, or the modifiers of a date type. For example, in the Figure, scopes 2 and 3 set the relation `type_params`, which is used by function calls to look up the order of the type parameters.

Name resolution Name resolution resolves a reference to a declaration. In Statix, name resolution happens in the form of queries on the scope graph. The following is a quick summary of how to query the scope graph with Statix.[7] A query has a relation `rel`, a filter, shadowing policy, and a scope `s`, and results in a list of path-datum entries. The query starts in `s`, finds all declarations in `rel` that are reachable according to the filter, removes the declarations that are matched by the shadowing policy, and finally returns a list of each declaration in combination with the path to the declaration. References in the scope graph are represented with dashed squares and arrows. There are many examples of name resolution in Figure 2.5b. We will highlight three. The first one is a simple lookup: `sumList`$_{22}$ can be resolved via 9, 7, 6, 5, 4, 3 and 1 to `sumList`$_1$. The second example is `x`$_{18}$ in scope 8, which can reach both `x`$_{20}$ and `x`$_8$. In this case, the shadowing policy is that the shortest path wins.[8] The final example is `z`$_{15}$ in scope 5. This reference does not resolve to any names: the only declaration `z`$_{16}$ in scope 7 is unreachable from scope 5.

[7] For a full explanation, see `https://www.spoofax.dev/references/statix/queries/`.

[8] To be precise: the end of the path is preferred over a `P` edge.

Chapter 3

PIE Improvements

This chapter describes the improvements that were made to the PIE DSL. Like the problems and contributions, these improvements are subdivided into expressive power, the code base and the user experience. The improvements in this chapter briefly summarize the problem that they solve, more extensive explanations for the problems can be found in section 2.1.

3.1 Expressive power

PIE DSL 1 cannot express a lot of tasks and projects because they use one or more features that the DSL cannot express. To address this issue, we added generics, suppliers, injected values and a module system to PIE DSL 2. To implement generics, we made a Statix specification for the static semantics of the DSL (see subsection 3.2.3). Since we were writing the specification from scratch anyway, we also took the time to implement least upper bound and class inheritance. The rest of this section describes these features in more detail.

3.1.1 Least upper bound

Finding the least upper bound on two types is not possible in NaBL2. Since PIE DSL 1 is specified in NaBL2, it could not use least upper bound. The places where it would be expected, if-else expressions and list literals, add a constraint that one of the sub-expressions is a supertype of the other(s). For a more detailed explanation of the problem, see section 2.1.1

Statix can express least upper bounds, so we implemented it while moving to Statix. Most built-in types are fairly straightforward: the top type always results in the top type, the bottom type results in the other type, and most other built-in types require an exact match or become the top type otherwise. If one (or both) of the two types is nullable, the least upper bound of the inner types are taken, and then the result is made nullable if it is not already: `lub(NullableType(T1), T2)= makeNullable(lub(T1, T2)).`

Custom data types search for the closest common supertype. If none is found, the least upper bound is the top type. If there is a common supertype, all type arguments for that common supertype are merged, and the resulting type arguments are used to create a new instance of the common supertype.

Merging type arguments first checks for an exact match: if they match exactly, they are kept as is. It then checks if one of the two type arguments subsumes the other. If so, that type argument is kept. Otherwise, a wildcard upper bounded by the least upper bound of both type arguments is used as type argument.

Lists are backed by Java `ArrayList`, so their inner type is a type argument in Java. Merging the inner types should use the same logic as merging type arguments for custom data types, but at the moment it does not do that yet, and instead just makes it the top type. This is a bug, because it means the compiler generates invalid Java code.

3.1.2 Comparisons

Comparisons (the equality and inequality operators) have a TODO to use lub in the PIE DSL 1 NaBL2 specification. The idea is that it should only compare two expressions if their types `T1` and `T2` have overlapping values, i.e. $T1 \cup T2 \neq \emptyset$. For example, comparing an integer to a string for equality will always be false. As it turns out, that is not lub but glb. Types do not share values, except for `null` and empty lists. In these cases, an explicit equality check can be made instead. For example, when comparing an `int?` to a `string?`, the only way they are equal is if they are both `null`, so the equality check `e1 == e2` is equivalent to `e1 == null && e2 == null`. A similar pattern can be used with empty lists with different element types. We decided that checking for (in)equality in these cases should be done explicitly, because that avoids two kinds of mistakes where someone does not realize these types share values this way. In the first, someone accidentally compares types which are in principle incomparable, except that both could be `null` or both could be the empty list, so the type system would not give a warning for this type of mistake. For the second kind of mistake, existing code makes use of that property, but someone does not realize it and constant folds the code to `false` or `true`.

By disallowing these two edge cases, types are fully disjoint unless one of the two is a subtype of the other. Checking this is possible in Statix by having subtyping return a boolean and using boolean functions, so we implemented that while implementing the Statix specification.

It later turned out that we assumed that two expressions can only be equal if they yield the same value, and that that assumption is incorrect. This means that certain valid and sensible comparisons are currently disallowed. We unfortunately did not have time to fix this anymore, so it is future work. The problem and a possible solution are described in more detail in subsection 6.1.18.

3.1.3 Functions to access fields and enums

We did not implement support for fields and enums in PIE DSL. However, now that generics exist, it is possible to express the foreign Java signatures of functions that can access fields and enums using Java reflection. Since these functions use strings to represent field names and fully qualified class names, they are not type checked. Since they rely on Java reflection, they may also not work if the security settings of the JVM disallow reflection.

Disclaimer: the following has been type-checked in PIE and Java, but it has not been run yet. The PIE DSL standard library also does not exist yet. Testing that these function work and publishing them in a library may or may not be done as part of my thesis after the greenlight. Most likely not, as it is not required for the case study.

Fields There are two kinds of fields: static fields and instance fields. Static fields require a class, while instance fields require an instance. This means that there are two functions for field access. The foreign Java declarations are shown in Figure 3.1a. The corresponding Java implementations are shown in Figure 3.2b.

Instead of representing the class for the static field access as a fully qualified name, it would also have been possible to take a `Class[C]`. However, this just postpones the problem, as now we must get a `Class` somewhere. This 'somewhere' will likely turn out to be a function which creates one from a fully qualified name. In the end, since we see no other use case for Classes, we decided to inline this function and just take the fully qualified name as input.

Enums Enums are easiest: every enum in Java comes with a static `valueOf` method which takes a string and returns the enum value with that name. An example declaration and use is shown in Figure 3.2a.

```
module stdLib:java

func getField[I, R](instance: I, fieldName: string) -> R =
  foreign java mb.pie.dsl.stdLib.java.Util#getField
func getStaticField[R](qualifiedClassName: string, fieldName: string) -> R =
  foreign java mb.pie.dsl.stdLib.java.Util#getStaticField
```

(a) The PIE standard library file declaring the foreign Java functions.

```
    public static <I, R> R getField(I instance, String name) {
        final Class<?> clazz = instance.getClass();
        try {
            Field field = clazz.getField(name);
            try {
                return (R) field.get(instance);
            } catch (ClassCastException e) {
                throw new RuntimeException("Could not cast actual type " + field.
                    getGenericType() + " to declared return type R", e);
            }
        } catch (IllegalAccessException e) {
            throw new RuntimeException("Cannot access field " + name + " in " +
                clazz.getName() + " due to security settings", e);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException("Field " + name + " does not exist in " +
                clazz.getName(), e);
        }
    }

    public static <R> R getStaticField(String qualifiedClassName, String fieldName)
         {
        try {
            return (R) Class.forName(qualifiedClassName).getField(fieldName).get(
                null);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("Could not load class " + qualifiedClassName
                , e);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException("Class " + qualifiedClassName + " does not
                have a field " + fieldName, e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException("Missing permission to access field " +
                fieldName + " of class " + qualifiedClassName, e);
        }
    }
```

(b) The Java implementations of the filed access functions.

Figure 3.1: A proposal for standard library functions to access fields

```
1  module example:enums
2
3  data Severity = foreign java fully.qualified.name.of.Severity {}
4  func Severity(severity: string) -> Severity =
5    foreign java fully.qualified.name.of.Severity#valueOf
6
7  func log(Severity, message: string) -> unit = foreign java Example#log
8
9  func readOrFail(file: path) -> string = {
10   val text = read file;
11   if (text == null) {
12     val message = "Could not read '$file'";
13     log(Severity("ERROR"), message);
14     fail message
15   }
16   text!
17 }
18
19 // declare a standard library function instead?
20 func enumValue[Enum](qualifiedName: string, valueName: string) -> Enum =
21   foreign java mb.pie.dsl.stdLib.java#getEnumValue
22
23 func useValueOf() -> Severity = Severity("ERROR")
24 func useStdLib() -> Severity =
25   enumValue[Severity]("fully.qualified.name.of.Severity", "ERROR")
```

(a) An example of how to declare and use the function to access enum values. It also contains a generic foreign Java declaration, and a comparison in how these two are used.

Figure 3.2: An example of ways to access enum values. *Continued on the next page.*

It is possible to define a generic function for accessing enum methods too, which is shown on line 20. The Java implementation for this library function is shown in Figure 3.2b. The `valueOf` method does not take an unchecked string for the qualified name of the enum. While this might make `valueOf` seem is safer, it actually has that exact same string, but then in the foreign Java reference to the function.

In the end, they are both equally unsafe, the difference is in the ergonomics. `valueOf` requires that this function is defined alongside the enum. However, after that it can be used with minimal overhead. The library function can be used with a single import (or qualified reference), but has to define the type, the fully qualified name, and the field name. We recommend to use `valueOf`, as that contains all the boilerplate of the fully qualified name in a single place, and makes the actual reference to the value very ergonomic.

3.1.4 Generics

PIE DSL 1 cannot express type parameterized data types and functions, also known as generics. Not only does this mean that no generic tasks can be defined, generic functions, methods and data types cannot be declared or referenced at all. Workarounds within PIE DSL 1 exist in theory, but using them adds so much boilerplate that it defeats the entire purpose of using the DSL. They are also not as powerful as full generics.

There are some half-measure solutions, such as making the generic classes built into the DSL. This works for those classes, but does not work for any other classes, and obviously

```
8     public static <E extends Enum<E>> E getEnumValue(String qualifiedEnumName,
          String valueName) {
9         Class<?> clazz;
10        try {
11            clazz = Class.forName(qualifiedEnumName);
12        } catch (ClassNotFoundException e) {
13            throw new RuntimeException("Could not load class " + qualifiedEnumName,
                  e);
14        }
15        try {
16            if (!clazz.isEnum()) {
17                throw new RuntimeException("Class " + qualifiedEnumName + " is not
                      an enum.");
18            }
19            E res = Enum.valueOf((Class<E>) clazz, valueName);
20            try {
21                return res;
22            } catch (ClassCastException e) {
23                throw new RuntimeException("Cannot cast " + clazz.getName() + "." +
                      res.name() + " to " + valueName, e);
24            }
25        } catch (ClassCastException e) {
26            throw new RuntimeException("Cannot cast class " + clazz.getName() + "
                  to Class<E>");
27        }
28    }
```

(b) The Java implementation for the PIE DSL standard library enum value access function.

Figure 3.2: *Continued from previous page.* An example of ways to access enum values.

adding every generic class to the DSL does not scale as a solution.

In the end, we decided to actually fully implement generics in the DSL. Data types can now have type parameters, functions can have type parameters, and there is a new wildcard type `_`, which can only be used for type arguments (and inner types for lists). For examples, see Figure 3.3.

Syntax The generics in the PIE DSL mostly follow the semantics of Java, as it has to interoperate with Java and compile to Java. A major difference with Java is the syntax.

Brackets First of all, Java uses angled brackets as brackets for lists of type parameters and type arguments. This leads to ambiguities if the list of type arguments comes after the function name in combination with less-than and greater-than operators though. A well known example is `outer(inner<B, C>(1))`. This can be parsed as a function call to `inner` with type arguments `A` and `B` and formal argument 1, which is then used as single formal argument in the call to `outer`. The other interpretation is as a call to `outer` with two formal arguments, `inner < B` and `C > 1`. Not only is this a syntactic ambiguity, due to overloading in Java, it could be an actual ambiguity as well. In Java, this ambiguity is resolved by putting the type argument list before the function name (Bracha, Cohen, et al. 2004). The PIE DSL uses square brackets after the function name instead.

```
class Box<T> {
  private T value;
  public Box(T value) { this.value = value; }
  public T get() { return value; }
  public None set(T newValue) { value = newValue; }
  // equals, hashCode and toString omitted for brevity
}
```

(a) Examples of generics in Java

```
data Box[T] = foreign java Box {
  func get() -> T
  func set(T) -> unit
}
func createBox[T](value: T) -> Box[T] = foreign java constructor Box
```

(b) Examples of generics in the PIE DSL.

Figure 3.3: A comparison of generics in Java and the PIE DSL.

Bounds This decision enables another change in syntax: instead of using the keywords **`extends`** and **`super`** to express upper and lower bounds, the PIE DSL uses the symbols `<:` respectively `:>`. These are already used to mean 'is a subtype/supertype of', so using them to express bounds works pretty well. With angled brackets as brackets these become hard to read, but with square brackets they work really well. While introducing these symbols, we also changed the syntax for the supertype of a data type to use these as well, so it changed from `data Apple : Fruit` to `data Apple <: Fruit`. In general, the PIE DSL now uses `<:` when upper bounding a type, while `:` is used to type an expression.

Wildcards Finally, Java uses `?` for wildcards. The PIE DSL already uses `?` for nullable types and making an expression nullable. While this would not lead to any ambiguities, it would still be a little hard to read. Instead, the PIE DSL uses `_` as the symbol for a wildcard. For example, `_ <: Fruit` is a wildcard upper bounded by `Fruit`.

Static semantics While the core specification of generics in the PIE DSL closely matches Java, there are a few features that are or could be different.

Bounds for type variables Every type parameter and wildcard in the PIE DSL has a lower and an upper bound. If they are not specified, they default to the bottom type and the top type respectively.

While the type system could in principle support declaring both a lower and an upper bound, Java only supports declaring one bound. The PIE DSL ultimately needs to compile to Java. Instead of rushing a decision on how to make code generation for multiple bounds work, we have decided to leave this as future work. For now, declaring both a lower and an upper bound are disallowed in the DSL.

Implicit wildcard bounds In Java, wildcards implicitly copy the upper bounds of their type parameter. This is illustrated in Figure 3.4. In the PIE DSL, wildcard bounds must always be stated explicitly. While it is a convenient feature that bounds do not have to be stated explicitly, it is not required to make generics work, and as such we have not implemented it yet.

Type inference The third feature is type inference. Type interference allows the user to leave out some type arguments or bounds if they can be derived (or 'inferred') from

```
public class Result<T, E extends Exception> {}
public class Example {
  public void example(Result<String, ?> result) {}
}
```

Figure 3.4: An example of implicit wildcard bounds. `Result<String, ?>` does not specify a bound for the wildcard, even though an unbounded wildcard is not within the bounds of `E`. This unbounded wildcard is nevertheless valid because it implicitly copies the upper bound from `E`.

clues in the context. Java gets this type inference by using existential constraint solving, i.e. it searches for a set of types that satisfy all the constraints. The generics for PIE are implemented in Statix, which also uses constraint solving. However, Statix uses universal constraint solving, i.e. it only instantiates something when that is the *only* possibility. This means that general type inference using constraint solving like in Java is not easily supported in Statix: it would have to be implemented on top of Statix. Type interference was deemed non-critical, so for now it remains to be implemented: every type argument and every type bound has to be specified.

Limitations Besides the missing convenience features mentioned in section 3.1.4, the current implementation of generics in the PIE DSL still has some limitations.

Code generation for generic PIE tasks Code generation of PIE tasks with type parameters has not been implemented yet. Generic PIE tasks in pure Java are fairly easy: they can just define the type parameters on the `TaskDef` itself. It becomes more complicated when we try to combine generic tasks with the Dagger injection framework. A cursory glance at some articles online tells us that instantiating a generic class with Dagger is not easy. Since the case study only uses generic data types, not generic tasks, we decided to postpone code generation for generic tasks until later.

As a result, the specification currently gives an error on functions with type parameters and an implementation in the DSL. Even though this makes type parameters in function bodies impossible, there are nevertheless tests for it to prevent regressions.

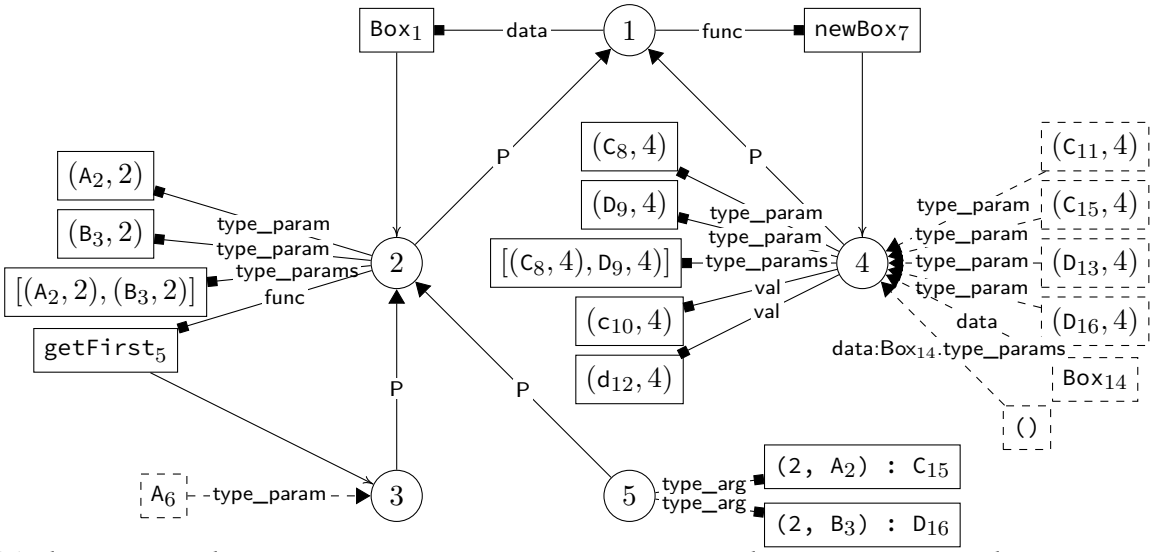
Soundness A formal proof of the soundness of the PIE DSL is out of scope for this thesis. Even without a formal proof, we can confidently say that the PIE DSL is most definitely unsound. There are a few known bugs where the specification of the DSL does not match our intended semantics. While we hope that this unsoundness is limited to these known and the undoubtedly existing unknown bugs, it may be that the intended semantics are fundamentally unsound.

This could be more likely than it seems. The generics in the PIE DSL follows those of Java. Surprisingly, generics in Java are unsound, as is shown by Amin and Tate (2016). The original Java file that demonstrates this unsoundness and our translation to the PIE DSL can be found in Appendix C. While we believe that this particular example would not be accepted by the PIE DSL because the DSL does not propagate constraints on type parameters like Java does, we were unfortunately unable to verify this because of one of the aforementioned bugs.

If issues like these are present in Java after years of intensive use and multiple studies into the formal semantics of the Java type system, it seems unlikely that the type system for the PIE DSL is entirely sound. On the other hand, the DSL does have far less edge cases such as implicit nullability, raw types, arrays and primitive types, so perhaps this simplicity means that the intended semantics for the DSL are indeed sound. While we lack a formal

```
data Box1[A2, B3] = foreign java Box4 {
  func getFirst5() -> A6
}
func newBox7[C8, D9](c10: C11, d12: D13) -> Box14[C15, D16] = foreign java constructor Box17
```

(a) The PIE program. It has a method which references the type parameter `A`, and a constructor which references the data type `Box` in its return type, and provides type arguments `C` and `D`.



(b) The scope graph. Types of declarations are omitted because they are not useful. The reference `()` in relation `data:Box.type_params` first resolves the scope of `Box` in relation `data` to scope 2, and then resolves `type_params` from scope 2.

Figure 3.5: An example of a generic data type `Box` with two type parameters `A` and `B`. The indices do not affect the name of an identifier, they are there to make the origin of each identifier in the scope graph clear. The type parameters for the constructor are named `C` and `D` to make it easier to differentiate them from `A` and `B`.

proof of soundness, we do have hundreds of tests for generics alone, so it seems unlikely that there are many bugs or unsound semantics that would come up in practice.

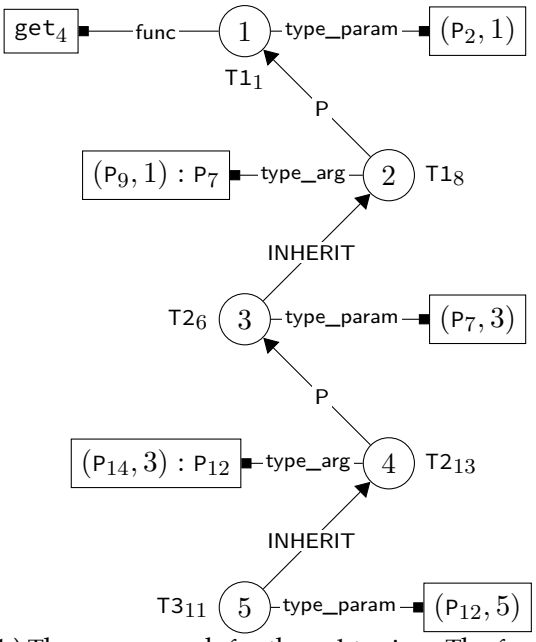
Implementation This is the first time that anyone has implemented generics for a full language with Statix, and consequently, using scope graphs in general. As such, the specific implementation could be interesting, particularly for those who want to implement generics for another language.

Scope graph layout for types Figure 3.5 shows an example of a PIE program with a generic data type and function. The Statix specification of PIE uses scopes as types (see section 3.2.3). This means every data type definition has a scope `s_data_def`, scope 2 in the figure. To define the type parameters, each type parameter is defined in `s_data_def` separately in a relation `type_param` mapping the name and scope to a lower and upper bound (the upper and lower bound are not shown in the figure). These separate type parameter declarations are used when resolving a reference to a type parameter, for example with A_6. Additionally, there is a single declaration `type_params` which has all type parameters in an ordered list. This is used when resolving a reference to the data type or function, as that will have an ordered list of type arguments without the associated names. This can be seen with the reference `data:Box14.type_params` in scope 4, which first resolves the data type Box_{14} to the declaration Box_1, and then looks up `type_params` in its associated scope 2.

```
data T1₁[P₂] = foreign java T1₃ {
  func get₄() -> P₅
}
data T2₆[P₇] <: T1₈[P₉] =
  foreign java T2₁₀ {}
data T3₁₁[P₁₂] <: T2₁₃[P₁₄] =
  foreign java T3₁₅ {}

func use₁₇(t₁₈: T3₁₉[int₂₀]) -> int₂₁ =
  t₂₂.get₂₃()
```

(a) A PIE program with generics and subtyping.



(b) The scope graph for the subtyping. The function scopes, global scope and references are omitted.

Figure 3.6: An example of instantiating type parameters when there is subtyping.

The function `newBox` has an associated scope which declares type parameters and has references just like the data type `Box`. References to data types may now have type arguments, which must be available for lookup when looking up members of the type. For example, the type of `getFirst` in the expression `newBox(int, int).getFirst()` is `int`, but in the expression `newBox(string, int).getFirst()` it is `string`. These type arguments are created in a new scope `s_data_instance`, which declares the type arguments in the relation `type_arg` as the name and scope of a type parameter and the type for that parameter. It uses the scope as well as the name of the type parameter to uniquely identify[1] the type parameter. This is useful for the instantiation of types, which is explained in the next paragraph. Such an `s_data_instance` scope is shown for the reference $\texttt{Box}_{14}$, which creates a new scope 5 with the type arguments $\texttt{C}_{15}$ and $\texttt{D}_{16}$ for the type parameters $\texttt{A}_2$ and $\texttt{B}_3$ respectively.

Instantiation First, a note on terminology. In the literature, finding the actual value of a type parameter is known as 'substitution,' because the literature often uses rewrite rule semantics, so the type parameter is substituted with the actual value. Term rewriting only does one substitution at a time. If that substitution contains more type variables, another substitution is performed, and so on until all substitutions are done. This makes substitution iterative. The PIE DSL code base uses 'instantiation,' which instantiates recursively until instantiation is done. For the purposes of explaining the instantiation algorithm, this difference does not matter. In practical terms, recursive instantiation might be slightly faster than iteratively substituting, but if that difference exists it is should be fairly minimal. The main difference is that recursively instantiating is a bit more straightforward to express in Statix. To be consistent with the code base, we will use 'instantiate' in this thesis.

To find the type of `t.get()`, we first take the type of `t`, which is `T3[int]`. After that we resolve `get` from `T3`, which resolves to `T1.get()`. We then need to instantiate the type parameter `T` from `get` to its actual type `int`, which will be the final type of `t.get()`.

[1] In legal PIE programs. An illegal program may have multiple type parameters with the same name for a single type or function. This is accounted for in the Statix specification, but currently with an incorrect error.

That last step can be done in two ways. The first one is to explicitly follow the supertypes and instantiate type parameters every time. This would first instantiate $\texttt{P}_{12}$ with the actual type $\texttt{int}_{20}$. Then it would instantiate $\texttt{P}_7$ with the type of $\texttt{P}_{14}$, which was instantiated to `int`. We would again follow the supertype and instantiate $\texttt{P}_2$ the type of with $\texttt{P}_9$, which is `int`. Finally, we are in the scope of the data type with the `get` method, so we can now instantiate the return type $\texttt{P}_5$ with the actual type of $\texttt{P}_2$, `int`. This instantiation method is eager: while following the supertypes, it eagerly instantiates every type parameter. This is inefficient when type parameters are not required for the final result. If `T1`, `T2` and `T3` all had more type parameters which were not required for type checking `get`, they would still all be instantiated. Similarly, if the class `T2` would use `T2[P] <: T1[string]`, then the type of `P` up till that point does not matter at all. Additionally, from this explanation, it is hopefully clear that explicitly following the supertypes is quite involved: it needs to keep track of the current scope, and it needs to keep track of which type parameters actually are part of the current scope.

The second instantiation method is far easier. Every type parameter includes a name and the scope it was defined in. We now do not need to explicitly follow every supertype when instantiating type parameters. We can simply instantiate the return type of the method from the current scope[2], and that will recursively instantiate type parameters until all instantiations are done. This method is used in the PIE DSL.

Until now, we have simply stated that instantiation and substitution run until they are 'done'. There is a fairly obvious definition of done, which is that there are no more type parameters to instantiate. However, there is the issue of non-termination of instantiation. If the substitution of a type parameter transitively contains that same type parameter (possibly after a few more substitutions), instantiation will not terminate. To avoid this, we keep track of which type parameters have been seen before in this instantiation. If we try to instantiate a type parameter we have seen before, we bail by leaving the second instance of the type parameter as is, with no further instantiation.

3.1.5 Injected values

The syntax for injected values can be seen in Figure 3.7b. This syntax hits several design goals.

First of all, it is part of the PIE function implementation, so it cannot be used in functions with foreign Java declarations, where injected values do not make sense. An earlier design made them part of the signature, but that required a bunch of warnings when they were used for foreign Java declarations.

Secondly, it is at the start of the function body, so it is clear that they cannot be declared at any point in the function. An alternative design has an `inject` expression that injects a value directly at that location, but that gives the impression that injection will happen at that point in the execution, while it actually already happens before the task starts.

Next, this keeps injected values somewhat separate from other language constructs. This is good, because they are essentially an implementation detail for PIE tasks.

Finally, the `inject in` can be omitted when there are no injected values. This means that this feature adds no boilerplate when it is not used. It also makes it backwards compatible.

The static semantics are simple: injected values use the same namespace as parameters and values. The injected values can be referred to in the body as regular parameters. The values must have proper names, so the name cannot be omitted and it cannot be an anonymous value.

Finally, code generation for the injected values generates code that is almost exactly the original Java code, seen in Figure 3.7a. The only difference related to injected values is that

[2] a helper scope that can access both the type parameters of `use` and those of `T3`

```
@TigerScope
public class TigerStyle implements TaskDef<Supplier<Option<JSGLRTokens>>, Option<Styling>>
    {
  private final TigerStyler styler;

  @Inject
  public TigerStyle(TigerStyler styler) { this.styler = styler; }

  @Override
  public String getId() { return getClass().getName(); }

  @Override
  public Option<Styling> exec(ExecContext context, Supplier<Option<JSGLRTokens>> tokens)
      throws IOException {
    return context.require(tokens).map(t -> styler.style(t.tokens));
  }
}
```

(a) The current Java code for `TigerStyle`.

```
func style(tokensSupplier: supplier[Option[JSGLRTokens]]) -> Option[Styling] =
  inject styler: TigerStyler in {
    val tokens = tokensSupplier.get();
    if(tokens.isEmpty())
      none[Styling]()
    else
      some[Styling](styler.style(tokens.unwrap().getTokens()))
  }
```

(b) `TigerStyler` in the PIE DSL.

Figure 3.7: The current Java code and the replacement in the PIE DSL for `TigerStyle`.

the generated code does not use unqualified names but qualifies them with **`this`**, for example **`this`**`.styler`.

3.1.6 Suppliers

Suppliers are built into the language. There is a special supplier type with a type parameter, and there are three expressions involving suppliers.

- Create a supplier from a task: `someTask.supplier[TypeArgs](args)`. This is special syntax. It returns a supplier with the return type of the task. Getting the value of the supplier will execute the task (or get a cached value).

  `identifier.supplier(args)` would be ambiguous between creating a supplier from a task and calling a normal method. This is intentional: creating the supplier is supposed to feel like calling a method on the task itself. To resolve this ambiguity, `supplier` is a reserved keyword for function names, which means that it cannot be parsed as a method call.

  Because creating a supplier from a task cannot be expressed in terms of existing PIE DSL constructs, it has its own rules in the grammar, Statix and the compiler.

- Create a supplier with a literal value: `supplier(arg)` or `supplier[Type](arg)`. From a user perspective, this is just like a normal function call, except that it can derive the type argument. Because 'supplier' is a reserved word, this has its own rule in the grammar

to still parse 'supplier' as a function name. It therefore also has its own rules in Statix and the compiler.

- Get value from a supplier: `supplierExp.get[]()`. This is just a normal method call on expressions with the type `Supplier[_]`. It does not have its own rule in the grammar: it is parsed as a regular method call. In the Statix specification, there is a special case for calls to a method named `get` on an expression with the type `Supplier[_]`. The Statix specification also specifies that this is a built-in method. In the code generation, there is also a special case for this built-in method, because getting the value from a supplier also requires the `execContext`, which cannot be expressed in the DSL.

3.1.7 Class inheritance

PIE DSL 1 has supertypes for data types but uses these only for nominal subtyping, not class inheritance. We added class inheritance to PIE DSL 2 as part of the move to Statix. Since the only data type implementation right now is the foreign Java implementation, class inheritance follows the semantics from Java. The current implementation does not include method overloading, so methods from super classes are shadowed by methods with the same name in subclasses. Method overloading has not been implemented because we want to make a principled design for method overloading in PIE DSL.

Given a class `Sub` which extends a class `Super` with a method $\texttt{m}_{super}$, there are four options:[3]

- `Sub` does not have a method with the same name as $\texttt{m}_{super}$. In both Java and PIE, `Sub` *inherits* $\texttt{m}_{super}$ from `Super`.
- `Sub` has a method $\texttt{m}_{sub}$ which is a *subsignature* of $\texttt{m}_{super}$. The signature of $\texttt{m}_{sub}$ is a subsignature of $\texttt{m}_{super}$ iff they are the same signature, or if $\texttt{m}_{sub}$ is the same signature after erasure of $\texttt{m}_{super}$. The erasure matching was added as a compatibility feature for interfacing with pre-generics code: it allows a generic method to override a non-generic method (Bracha, Odersky, et al. 1998).

  In both Java and PIE, $\texttt{m}_{sub}$ *overrides* $\texttt{m}_{super}$. $\texttt{m}_{sub}$ must be *return-type-substitutable* for $\texttt{m}_{super}$, or an error is emitted. In Java, this has several (sometimes intentionally unsound) edge cases related to primitive types and backwards compatibility. In PIE, these edge cases do not exist, and all that we require is that the return type is covariant, i.e. the return type of $\texttt{m}_{sub}$ is assignable to that of $\texttt{m}_{super}$.
- `Sub` has a method $\texttt{m}_{sub}$ with the same name, but which is not a subsignature of $\texttt{m}_{super}$, and $\texttt{m}_{sub}$ is not equal to the erasure of $\texttt{m}_{super}$. In Java, $\texttt{m}_{super}$ is inherited and available as an overload of $\texttt{m}_{sub}$. In the PIE DSL, $\texttt{m}_{sub}$ shadows $\texttt{m}_{super}$.
- `Sub` has a method $\texttt{m}_{sub}$ with the same name, which is not a subsignature of $\texttt{m}_{super}$, but the signatures are equal after erasure.[4] In this case, in both Java and PIE, $\texttt{m}_{sub}$ *collides* with $\texttt{m}_{super}$. $\texttt{m}_{sub}$ does not override $\texttt{m}_{super}$, and an error is emitted about the signature collision after type erasure.

In summary, Java and the PIE DSL behave identically, except when a method with the same name exists but does not override $\texttt{m}_{super}$. In that case, $\texttt{m}_{super}$ is inherited in Java, but is shadowed in PIE.

As can be seen in these specifications, checking whether a method inherits, overrides or collides with a super method requires type erasure. We implemented type erasure following

[3] for details, see the Java specification: `https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.8`

[4] This is a special case of *override-equivalence*. Because the PIE DSL only allows one method with the same name per data type, that rule simplifies to both signatures being the same after erasure

definitions.pie

```
module org:example:thesis:moduleSystem:imports:definitions
function theTask() -> int = 0
```

uses.pie

```
module org:example:thesis:moduleSystem:imports:uses

import org:example:thesis:moduleSystem:imports:definitions:theTask
import org:example:thesis:moduleSystem:imports

function useIt() -> int = {
  val x = org:example:thesis:moduleSystem:imports:definitions:theTask();
  val y = theTask();
  val z = imports:definitions:theTask();
  x+y+z
}
```

Figure 3.8: An example of referencing a task in another module. `x` uses a fully qualified reference of the task. `y` uses a direct import. `z` uses a qualified reference from an imported module.

the Java specification. We did not add new constructors for erased types. Instead, if an existing constructor has a type parameter built in (e.g. `Type -> ListType`), it uses `UnitType()` as type argument.

3.1.8 Module system

At the start of this thesis, the PIE DSL used single-file analysis, which means that a PIE program could not refer to things outside the file. For single projects, that essentially means that everything has to be done in a single file.[5] Multi-project setups were unsupported.

The obvious solution is to enable multi-file analysis, which analyzes multiple files as a single PIE program. However, just enabling multi-file analysis means that everything is put into the global namespace, which can lead to name collisions. For example, many language projects will use tasks such as `parse` and `analyze`. The solution is to introduce namespaces, which leads to a module system.

The semantics of the module system The PIE DSL module system makes every file a named module. Modules can be nested, e.g. `module org:example:exampleModule`. Each file must start with a module statement, which specifies a unique fully qualified module name. For compilation, the module name is translated directly to a Java package name, so the example would become **`package`** `org.example.exampleModule`. The parent modules do not need to exist, so in the example, there does not need to be a file that declares the module `org` or `org:example`.

Functions and types from modules can be referenced directly with a fully qualified reference (e.g. `org:example:tasks:someTask`) or by importing the function, type or module and then referencing the imported element. An import makes the name it imports available, so `import org:example:foo` declares the name `foo` in the file of the import. An example can be found in code listing 3.8.

Imports also have two more advanced features. The first is renaming, which makes an element available under a different name. For example, `import org:example:foo as bar` de-

[5] unless the functions in both files are entirely separate, which rarely happens

```
import mb:common:util:ListView
import mb:common:util:createListView
import mb:common:util:createEmptyListView
```

(a) Three simple imports for the type `mb.common.util.ListView` and two of its constructors

```
import mb:common:util:{ListView, createListView, createEmptyListView}
```

(b) The equivalent multi-import

Figure 3.9: Three simple imports and the equivalent multi-import

```
import a:{b1, b2}:c
```

(a) An import with a multi-import in the middle

```
// Not equivalent to the multi-import
import a:b1:c
import a:b2:c
```

(b) Incorrect decomposition of the multi-import

```
import a:b1:c as b1:c
import a:b2:c as b2:c
```

(c) Correct decomposition of the multi-import

Figure 3.10: An import with the multi-import in the middle, and its decompositions according to two possible semantics for decompositions of multi-imports. The first decompositions, in Figure 3.10b, does not work, because it creates two imports that both declare a name `c`. To avoid that, the actual semantics defines that the name in the multi-import is used as well. This is shown in Figure 3.10c.

clares `bar` which refers to `org:example:foo`. The new name can be in a module as well, for example, `import org:example:foo as newModule:bar` creates a local declaration for the module `newModule` with the element `bar`, which refers to `org:example:foo`. Other elements could be added to `newModule` by making more imports that rename there, like `import org:example:baz as newModule:baz`

The other feature is multi-imports, which import multiple elements in a single import statement. They are essentially syntactic sugar for multiple simple import statements. Unlike Java, constructors are not part of the type in the PIE DSL, so each constructor needs to be imported separately. This leads to many imports from elements of the same package. For example, figure 3.9a shows three simple imports for the `ListView` type and two of its constructors. Multi-imports allow writing these as one import, as can be seen in figure 3.9b.

The logic for the names a multi-import makes available is not as straightforward as it may seem at first glance. The counter-intuitive part comes from making the middle of the import a multi-import. The multi import in code listing 3.10a is not decomposed into 3.10b, because that would declare two elements named `c`. Instead, the name in the multi-import is part of the name, so the correct decomposition of the multi-import can be found in 3.10c.

Implementation The module system is implemented in Statix, which uses scope graphs as representation for scopes and name binding. The implementation of the module system takes the set of fully qualified module names and imports from the PIE files and declares them in the scope graph. It also makes Statix functions available that resolve a fully qualified reference to a module scope in the scope graph.

Declaring trees in the scope graph Conceptually, the module system forms a tree. The implementation creates multiple trees. The first tree is the *module tree*, which represents all concrete modules. This tree is defined in the project scope, and as such is visible in every file. The other trees are *import trees*, which represent pseudo-modules created by imports. Each file (concrete module) has one import tree. In every file, the import tree for that file and the module tree are visible and accessible as if there was only one tree.

Require: A set M of MODULEs and a scope s to declare the tree in
  $tree \leftarrow$ `TreeRootNode([])`
  for all $module \in M$ **do**
    $tree \leftarrow addToTree(tree, module)$
  end for
  $declareInScopeGraph(s, tree)$

Figure 3.11: The tree building algorithm. A MODULE is a three-tuple of a fully qualified module name, associated scope, and whether it is a concrete module or a pseudo module. The $addToTree$ function takes a tree constructor and a MODULE, and returns a new tree constructor with the module added. It is called for each module, the updated tree is passed along with each call. $declareInScopeGraph$ takes the final tree constructor and declares the tree in the scope graph.

Statix has a concept called *permission to extend*.[6] This limits which scopes can still be extended by adding declarations or edges to other scopes. Only scopes that were created in the current function or which were passed as a parameter to the current function have permission to be extended. This means that it is not possible to extend a scope that was obtained with a query on the scope graph. That in turn makes it impossible to declare the trees by declaring the concrete modules one by one, all submodules must be known or derivable when creating the scope for the module. This is handled by the tree building algorithm.

The tree building algorithm can be found in Figure 3.11. It takes a set of MODULEs, which consist of the fully qualified module name, a scope, and whether the module is a concrete module or a pseudo module. It also takes a scope from the scope graph where it will declare the tree. It turns the MODULEs into a tree using constructors, and then declares that tree in the scope graph.

The tree building algorithm is used for both the module tree and the import tree. For the module tree, the qualified names are the names of concrete modules. These are part of the parse forest. To get them all, each file declares the fully qualified module name and the corresponding file scope in the project scope. The project analysis then queries the root scope for all qualified names and calls the tree building algorithm.

Creating the import tree To build the import tree, first all qualified module names for the pseudo modules have to be extracted from the imports. The full import tree algorithm can be found in Figure 3.12. It first normalizes all imports. A normalized import is a single renaming from the fully qualified name of the original declaration to the fully qualified name of the declaration in the importing file. Multi-imports are normalized to multiple normalized imports. For example, `import a:b:{c1, c2}` becomes `{RENAMING("a:b:c1", "c1"), RENAMING ("a:b:c2", "c2")}`. Then, for each normalized import, it resolves the import, creates the declarations in the current file, adds errors as needed, and adds MODULEs and SUBMODULEs to the set M. Finally, it calls the tree building algorithm on M to build the import tree and declare it in de scope graph. Each MODULE in M becomes a pseudo module, while each SUBMODULE M points from a pseudo module (or directly from the import tree root scope) to a concrete module in the module tree.

It should be noted that the implementation for imports in the PIE DSL creates new declarations in the current file for imports. This is not standard practice, but is required to support renaming of imports. Since renamed imports would have to be re-declared anyway, it was deemed simpler to just redeclare every import.

[6] See `https://www.spoofax.dev/references/statix/scope-graphs/#permission-to-extend` for the full documentation

Require: A set of imports I, the file scope s_{file}.
  $M \leftarrow \varnothing$
  for all $i \in I$ **do**
    $I_n \leftarrow normalize(i)$ ▷ $normalize(i)$ returns a set of normalized imports
    for all $RENAMING(fq_{orig}, fq_{new}) \in I_n$ **do** ▷ fq stands for 'Fully Qualified'
      $qualifier_{orig} \leftarrow getQualifier(fq_{orig})$ ▷ $getQualifier("a:b:c") = "a:b"$
      $s_{mod} \leftarrow resolveModule(s_{file}, qualifier_{orig})$
      if $s_{mod}.status \neq OK$ **then**
        $addError("Undefined\ module\ \{qualifier_{orig}\}")$
        continue
      end if
      $name_{orig} \leftarrow getName(fq_{orig})$
      $name_{new} \leftarrow getName(fq_{new})$
      $res_{mod} \leftarrow resolveMod(s_{mod}, name_{orig})$
      $res_{data} \leftarrow resolveData(s_{mod}, name_{orig})$
      $res_{func} \leftarrow resolveFunc(s_{mod}, name_{orig})$
      if $res_{mod} \cup res_{data} \cup res_{func} = \varnothing$ **then**
        $addError("Undefined\ element\ \{name_{orig}\}\ in\ \{qualifier_{orig}\}")$
        continue
      end if
      if $|res_{mod}| \geqslant 2$ **then**
        $addError("BUG:\ duplicate\ definition\ for\ module\ \{name_{orig}\}")$
      end if
      if $|res_{data}| \geqslant 2$ **then**
        $addError("Duplicate\ definition\ for\ data\ type\ \{name_{orig}\}")$
      end if
      if $|res_{func}| \geqslant 2$ **then**
        $addError("Duplicate\ definition\ for\ function\ \{name_{orig}\}")$
      end if
      if $getQualifier(fq_{new}) = ""$ **then**
        $s_{decl} \leftarrow s_{file}$
      else
        create new scope s_{proxy}
        $M \leftarrow M \cup \{MODULE(s_{proxy}, fq_{new})\}$
        $s_{decl} \leftarrow s_{proxy}$
      end if
      if $res_{mod} \neq \varnothing$ **then**
        $M \leftarrow M \cup \{SUBMODULE(res_{mod}, qualifier_{orig})\}$
      end if
      for all $decl \in res_{data}$ **do**
        $declareData(s_{decl}, name_{new}, decl.declarationInfo)$
      end for
      for all $decl \in res_{func}$ **do**
        $declareFunc(s_{decl}, name_{new}, decl.declarationInfo)$
      end for
    end for
  end for
  $treeBuildingAlgorithm(M, s_{file})$

Figure 3.12: The import tree algorithm takes all imports in a file, resolves them, declares them, and declares the import tree.

Require: A fully qualified reference $ref = [q_1, ..., q_n, name]$.
  if resolving the original declaration for an import **then**
    $S_{cur} \leftarrow \{s_{module\_tree\_root}\}$
  else
    $S_{cur} \leftarrow \{s_{module\_tree\_root}, s_{import\_tree\_root}\}$
  end if
  for all $q_i \in getQualifier(ref)$ **do**
    $S_{cur} \leftarrow \{\, s \mid s_{cur} \in S_{cur} \wedge s = resolveMod(s_{cur}, q_i) \,\}$
    if $S_{cur} = \varnothing$ **then**
      $addError("Undefined\ module\ \{q_i\}\ in\ module\ \{q_1, ..., q_{i-1}\}")$
      return E_UNDEFINED_MODULE
    end if
  end for
  $Res \leftarrow \{\, res \mid s_{cur} \in S_{cur} \wedge res = resolve(s_{cur}, name) \,\}$
  if $Res = \varnothing$ **then**
    $addError("Undefined\ \{namespace\}\ \{name\}\ in\ module\ \{q_1, ..., q_n\}")$
    return E_UNDEFINED_NAME
  else if $|Res| \geqslant 2$ **then**
    $addError("Duplicate\ \{namespace\}\ definition\ \{name\}\ in\ module\ \{q_1, ..., q_n\}")$
    return E_DUPLICATE_NAME
  end if
  $\{res\} = Res$
  return res

Figure 3.13: The qualified reference resolution algorithm. The namespace to resolve $name$ in is derived from the syntactic position of the reference.

Resolving qualified names Resolving a qualified reference is fairly straightforward. The pseudo code for the algorithm can be found in Figure 3.13. In short, it starts in the root scopes of both the module tree and import tree, resolves the qualifier to a module scope from there, and finally resolves the name in that module scope.

The start point is either just the module tree root scope or the module tree root scope and import tree root scope. The import tree root scope is not used when resolving the original declaration for an import, because that would lead to a circular dependency where declaring the imports depends on the import tree.

The namespace ('data', 'func' or 'mod') to resolve the final name is determined based on the syntactic position of the reference. For example, in `a:b:c[d:e:f]()`, the syntax identifies this as a function call. This means that `a:b:c` is a function and must be resolved in the function namespace, while `d:e:f` is a type argument and must be resolved in the data namespace.

3.2 Code base

To enable the changes we made to the PIE DSL and to improve the experience of working on the DSL, we made several improvements to the code base. First, we changed to compiler to compile to ASTs instead of using string interpolation. We added tests to prevent regressions, as documentation and as a way to think about edge cases for new features. We switched from NaBL2 to Statix to enable us to implement generics (described in subsection 3.1.4). Finally, we split the constructors for types from PIE DSL 1 into syntactic and semantic constructors, which allows Statix to statically error when using a syntactic type where a semantic type should be used. The following sections describe these improvements in more details.

3.2.1 Compile to ASTs

The compiler compiles PIE DSL code to Java code. Spoofax automatically generates a parser that parses a PIE DSL program to an AST. In PIE DSL 1, this AST is compiled to Java using string interpolation.

The new compiler compiles the PIE DSL AST to a Java AST, which is mostly statically checked for typos by Stratego and allows optimization of the generated Java code. Spoofax generates a prettyprinter from the SDF3 specification for Java. This prettyprinter takes the Java AST to generate the final Java code.

While this change in the compiler backend is an improvement, it does not change the PIE DSL, it only makes it easier to maintain and improve the compiler in the future.

3.2.2 Add tests

Tests provide confidence that a language feature, bugfix or refactoring has been implemented well and prevent introducing regressions in the future. Testing of PIE DSL 1 is performed by manually compiling a large PIE file. This manual testing leads to a lot of mistakes which slows down development. It also results in a lack of confidence that refactorings can be done quickly without introducing regressions, which means that many small improvements are not implemented.

To mitigate these issues, we have implemented automated tests. SPT is a declarative DSL for testing language specifications. We use SPT tests to unit test the grammar and static semantics, and have well over 1500 tests.[7] An example of SPT tests for if-else expressions is shown in Figure 3.14. The grammar tests test the grammar and generated parser. Figure 3.14a shows the grammar tests for if-else expressions. Testing the grammar is mostly the same for every language feature. The grammar can be tested with a correct example (line 1), with missing keywords or syntactic elements (lines 2-3), with layout within keywords or operators (not shown), with layout between elements that typically have no layout between them (not shown) and without layout between elements that typically do have layout between them (lines 4-7).

The semantics tests test the static semantics. What to test for is highly dependent on the language feature under test. The semantics tests for if-else expressions can be found in Figure 3.14b. If-else expressions require that the condition has a boolean type (line 3) and takes the least upper bound of both branches for the overall type of the expression. This is tested with simple types (line 1-2 and 17), with nullability (lines 4-5), user-defined datatypes (lines 6-7), generics (lines 8-16) and with unresolved types (lines 18-19).

Unit testing the compiler is brittle because the input would be an ASTs and an analysis result, which do not have a stable application programming interface (API). Instead, we implemented end-to-end tests in Java to test the compiler.[8] Figure 3.15 shows the relevant fragments of the end-to-end test for calls to foreign Java functions. The full files can be found in Appendix B. An end-to-end tests consists of a full PIE program (3.15a), optionally Java code with definitions (3.15b), Java code to run the generated Java code from the PIE file and check the output in a JUnit test (3.15c) and scaffolding code for Dagger, an injection framework (3.15d and 3.15e). The PIE program is parsed, analyzed and compiled to Java, and then the Java code runs the JUnit test to verify that it is also semantically correct.

While all these tests cost a lot of time to write, they were definitely worth it, because it gives us the confidence to make changes without fear of introducing a regression.

[7] The full SPT test suite can be found at `https://github.com/metaborg/pie/tree/develop/lang/lang/test`.

[8] The full end-to-end test suite can be found at `https://github.com/metaborg/pie/tree/develop/lang/lang.test`.

```
test ifelse [[ if (cond) "yes" else "no" ]] parse succeeds
test ifelse missing keyword if [[ (cond) "yes" else "no" ]] parse fails
test ifelse missing keyword else [[ if (cond) "yes" "no" ]] parse fails
test ifelse no layout after if [[ if(cond) "yes" else "no" ]] parse succeeds
test ifelse no layout before else [[ if (cond) "yes"else "no" ]] // todo parse fails
test ifelse no layout after else [[ if (cond) "yes" else"no" ]] parse fails
test ifelse no brackets [[ [[ if value == 10 "hello" else "world" ]] ]] parse fails
```

(a) SPT tests for the grammar of if-else expressions. The tests in the file are configured to start parsing from the `Exp` sort, which permits us to write a simple fragment without writing a full PIE program.

```
test ifelse [[ [[ if (value == 10) "hello" else "world" ]] ]]
  analysis succeeds run pie-ast-type on #1 to StrType()
test ifelse condition not boolean [[ if ([["true"]]) 1 else 5 ]] 1 error at #1
test ifelse true branch null [[ [[ if (value == 10) null else "a string" ]] ]]
  analysis succeeds run pie-ast-type on #1 to NullableType(StrType())
test ifelse branches least upperbound [[ [[ if (value == 0) bar else bak ]] ]]
  analysis succeeds run pie-ast-type on #1 to DataType(_)
test ifelse branches equal with type arguments [[ [[
  val res: Generic[Foo, Baar, string] = if (value == 0) generic1 else generic1
]] ]] analysis succeeds run pie-ast-type on #1 to DataType(_)
test ifelse branches simple type equal with different type arguments [[ [[
  val res: Generic[_ : Foo, Baar, _] = if (value == 0) generic2 else generic1
]] ]] analysis succeeds run pie-ast-type on #1 to DataType(_)
test ifelse true branch subtype with different type arguments [[ [[
  val res: Generic[_ : Foo, Baar, _] = if (value == 0) genericSub else generic1
]] ]] analysis succeeds run pie-ast-type on #1 to DataType(_)
test ifelse branch type mismatch [[ if (value == 0) "hello" else 10 ]] 1 error
test ifelse error in dead code [[ if (true) "hello" else [[not_defined]] ]]
  error like "resolve" at #1
```

(b) SPT tests for the static semantics of the 'if-else' expression. Not all semantic tests for if-else expressions are included here, this is just a sample to show the range of things that are tested. Semantic tests require a full PIE program because Statix cannot start at an arbitrary AST node. To avoid repeating the code around these fragments, SPT offers a *test fixture*. This test fixture contains common surrounding code. In this case, it defines the names and types of the values, and the subtyping relations between Foo, Bar, Baar, etc.

Figure 3.14: SPT tests for the grammar and static semantics of if-else expressions.

3.2.3 Specify the static semantics in Statix

As was discussed in *NaBL2 limits static semantics* (2.1.2), the static semantics of PIE DSL 1 were limited due to NaBL2. In summary, NaBL2 had two distinct phases for gathering constraints and solving constraints, and language-defined semantics for its two kinds of scope graph edges, which meant that it was impossible to express a generic type system and import renaming. It was also slow and tedious to work in, and is now deprecated.

Statix is the successor of NaBL2. It is a functional meta-DSL that also uses constraints to specify the static semantics of a language. Unlike NaBL2, Statix interleaves constraint gathering and constraint solving, which greatly enhances the expressive power compared to NaBL2, and allows it to express generics. It also does not define semantics for edges, which allows it to express renaming imports. Finally, Statix code is type checked in the editor, which means that errors are caught statically in the editor instead of dynamically at runtime. This is a huge improvement over NaBL2, because it enables faster language development and prevents a lot of frustration.

The change to Statix meant a complete reimplementation of the static semantics of the PIE DSL. During this reimplementation we took the opportunity to implement some tasks

```
module mb:pie:lang:test:call:foreignFunc:generic

func func[C, D](c: C, d: D) -> D =
  foreign java mb.pie.lang.test.call.Bar#func

func main_generic() -> (string, bool) = {
  func[int, (string, bool)](217, ("generic", true))
}
```

(a) The PIE file that is compiled to Java.

```
    public static <C, D> D func(C c, D d) {
        return d;
    }
```

(b) The Java file `Bar.java` with supporting definitions for the PIE file in 3.15a.

```
class GenericTest {
    @Test void test() throws ExecException {
        assertTaskOutputEquals(DaggergenericComponent.class, new Tuple2<String,
            Boolean>("generic", true));
    }
}
```

(c) The Java fragment to test the Java code generated from the PIE code.

```
@mb.pie.dagger.PieScope
@Component(modules = {PieModule.class, PieTestModule.class}, dependencies = {mb.log
    .dagger.LoggerComponent.class, mb.resource.dagger.ResourceServiceComponent.class
    })
public interface genericComponent extends PieComponent {
    main_generic get();
}
```

(d) The Dagger Component, which tells Dagger to generate a class that provides an instance of `main_generic`.

```
@Module
abstract class PieTestModule {
    @Provides @mb.pie.dagger.PieScope @ElementsIntoSet
    public static Set<TaskDef<?, ?>> provideTaskDefs(
        main_generic generic
    ) {
        final HashSet<TaskDef<?, ?>> taskDefs = new HashSet<>(1, 1);
        taskDefs.add(generic);
        return taskDefs;
    }
}
```

(e) The Dagger Module, which declares the generated task `main_generic` for the PIE runtime.

Figure 3.15: Fragments of the end-to-end test for a call to a generic foreign Java static function. The full files can be found in Appendix B.

that were marked as TODO in the NaBL2 code. We also implemented some new features for the static semantics of the PIE DSL to solve issues that we ran into during this master thesis. Overall, the move to Statix came with many small and a few large improvements over the NaBL2 implementation.

Changes discussed elsewhere There are a few changes that were already mentioned in this chapter.

- *Least upper bound* (3.1.1)
- Allow comparisons where the left hand side is a subtype of the right hand side: *Comparisons* (3.1.2)
- *Suppliers* (3.1.6)
- *Class inheritance* (3.1.7)
- Differentiate between syntactic and semantic types: *Different constructors for syntactic and semantic types* (3.2.4)
- *Module system* (3.1.8)
- *Generics* (3.1.4)

Scopes as types In NaBL2, types are represented with occurrences, which are particular declarations in the scope graph. Subtyping is declared in a *relation*, which is built into the language. Constraints on this relation can either succeed or fail, but there is no way to use the result of that constraint.

Antwerpen, Poulsen, et al. (2018) introduces *scopes as types*, which is exactly what it says on the tin: types are represented by scopes instead of declarations. This enables more complicated subtyping rules by creating edges in the scope graph and performing queries.

While moving to Statix, we started using scopes as types as well. This enables our implementation of generics, which uses alternating `P` and `INHERIT` edges to express subtyping. It also makes it possible to perform more complicated queries than just 'is `A` a subtype of `B`?' For example, lub finds the closest supertype of a type `A` that is also a supertype of `B`. Because this is a query, and no longer a constraint, the result can be used in further computations.

Appending `null` to lists It is now possible to append `null` literals to lists with non-nullable elements. The list element type is made nullable as part of this operation. For example, `[1, 2, 3] + null` results in the list `[1, 2, 3, null]`, which changes the list from `int*` to `int?*`. This likely would have been possible in NaBL2, but it was simply never implemented in PIE DSL 1.

Better editor message Spoofax supports three types of messages: errors, warnings and notes. PIE DSL 1 does not have any warnings or notes. While implementing the Statix specification, we made many changes to messages. Existing errors were made clearer, we added new errors, warnings and notes, and we implemented the specification to avoid cascading errors in certain cases.

Reworded existing errors The errors in PIE DSL 1 often do not explain in much detail what is wrong. Most errors have the form *Type mismatch: expected [type kind], got [actual type]*, where `type kind` is a description of what type was expected, and `actual type` is the type that was provided. For example, when trying to call a method on a nullable type, PIE DSL 1 gives the error *Type mismatch: expected callable type, got [ty_data]*. While this describes what the error is it does not describe how to resolve the error. The error does provide a hint in the sense that it says that it expected a callable type. It also shows the actual type, so presumably that is not a callable type then. However, it does not explain how to make that type callable.

The PIE DSL 2 error is *Cannot call method on nullable type [exp_ty], make it non-nullable first*. This is more specific about what exactly the error is, and it explains how to solve the error: make the type non-nullable.

In the previous example, both the NaBL2 and the Statix specification have an explicit error. This is not always the case. For example, supertypes cannot be cyclical. This means that a type cannot (indirectly) be its own supertype. In NaBL2, this is enforced with the built-in subtyping relation, which is required to be non-cyclical. If that constraint is broken by creating a cyclic dependency, NaBL2 has a very cryptic error on one of the supertype declarations: *Adding violates anti-symmetry*. This error is basically only understandable if you know the implementation details of supertypes. Because relations are built into NaBL2, there is no way to customize the error.

Statix does not have relations in the same sense as NaBL2.[9] The check if supertypes are cyclical is implemented manually. Statix has a powerful scope graph model, so that check is not too difficult. Because we implemented the check by hand, we can use a custom error message: *Cannot (indirectly) inherit from self. [super_name] is already an (indirect) subtype of [name].*

New messages Some mistakes are not caught by PIE DSL 1. The obvious ones are for new features, such as from the module system and generics. However, there are also a few errors in PIE DSL 1 that were not caught. Examples are using a variable before it is declared and having the same modifier multiple times for a data declaration.

There are now also warnings and a note. A new warning that is also applicable in PIE DSL 1 is a double nullable on types, e.g. the type `int??`. This is allowed, but it is simply equivalent to the much clearer `int?`, so it is a warning. There are also warnings for new features. First, the module system brings warnings for empty multi-imports, multi-imports with a single element, renaming to the same name and making a name available that was already available under that name (e.g. `import mb:common:result:Result as mb:common:result:Result`). Additionally, bounds for type variables have warnings in case the bound only includes a single type, or only a single type and the bottom type. Next, anonymous values give warnings when every element of an assignment is anonymous and on anonymous parameters on foreign Java declarations. Finally, injected values give a warning if the section is left empty, e.g. `func empty()-> int = inject in 8`.

There are two kinds of note. The first note is shown on the type parameter with the same name as an existing type: *Type parameter [name] shadows data type [name]*. As the message notes, the type parameter will shadow that type. This is allowed, but gives a note, as it is normally not something you would want to do. The second note is for anonymous parameters on a PIE DSL task. The message is rather long, but it should be self explanatory: *Anonymous parameter _ cannot be referred to but will be included in the generated task signature. If this is not required, this parameter can be removed. If this is intended, it is recommended to add a comment explaining why this is necessary. To refer to this parameter, use a different name.*

[9] Statix does have relations, but those are analogous to NaBL2 namespaces instead.

Avoid cascading errors Cascading errors are errors which are the result of other errors. There is one case where the Statix specification has been specifically written to avoid cascading errors. A function call `someFunc(1, 2, 3)` has 3 constraints: the function exists, the arguments must be valid expressions, and the argument types must be subtypes of the parameter types. If `someFunc` does not exist, that is an error. However, the constraints that the arguments match the parameters now also fail, because the parameters cannot be looked up. In PIE DSL 1, this was not accounted for, and leads to cascading errors. In PIE DSL 2, the Statix specification is written to avoid these errors: if the function does not resolve, the arguments are type checked, but it is not checked if they match the parameters.

Statix has a feature where it will try to detect and suppress cascading errors: if a constraint failed due to an unbound variable which is used in an earlier failing constraint, it is assumed that the variable is unbound *because* the earlier constraint failed, and no error for the later constraint is generated. However, this feature came after the Statix spec for the PIE DSL had largely been written, so this specification has a few patterns that prevent the cascading error detection from working everywhere. Right now it is unclear how much the cascading error suppression really helps, because we try to write correct code, and when the suppression works it only means that something does not have an error, which is hard to spot.

An explicit type for empty lists Empty lists have the type `ListTy(TopTy())` in PIE 1. While this allows assigning elements to the list (which is often the intention for empty lists), it does not allow assigning the list to anything that does not take a list of the top type. PIE 2 has a specific type for empty lists, which allows it to append or concatenate to an empty list and have it update the element type. Since there is an explicit type for empty lists anyway, we also implemented warnings for nonsensical operations, such as concatenating an empty list to another list or a list comprehension over an empty list.

Bottom type This is required for the lower bounds of type variables. Like the top type, this type does not have syntax, so it is impossible to express that a value has the bottom type within the DSL.

Since the bottom type exists, there were also some other cases where we considered using it. In two cases it is already implemented, in two other cases it could be implemented, and in the last case it has been rejected for now.

List comprehension over empty list The first case is as element type of the input list in a list comprehension over an empty list. Because the element type for the list is unknown, we use the bottom type to check the body of the list comprehension.

Base case of `listLub` The second case is as return value of the base case of `listLub`, which calculates the lub of a list of types. The base case is `listLub([])`, i.e. take the lub of zero types. `listLub` previously required at least one element, so this is certainly a cleaner design.

Type of `return` and `fail` We are considering using the bottom type as type of the `return` and `fail` statements, but this is low priority and we have yet to consider the full ramifications of these changes.

Element type of empty lists Finally, instead of creating a special type for empty lists, we first considered a regular list with the bottom type as element type (i.e. `typeOf(s, ListLit([]))= ListType(BottomType())`). However, that does not work out when list types are not

covariant in their element type. Without such covariance, the list of bottom cannot be assigned to any other list type, so `val ints: int* = []` would be an error. We are considering adding covariance for lists, so this is planned to be implemented afterward.

3.2.4 Different constructors for syntactic and semantic types

As explained in section 2.1.2, the specification for the static semantics of PIE DSL 1 did not differentiate between syntactic and semantic types. This worked for many simple built-in types, but not for custom data types. To solve this, we introduced different constructors for static and semantic data types. Because functions on types (such as `isAssignableTo` and `lub`) only take semantic types and Statix statically checks this, it is now impossible to use a syntactic type if a semantic type is required.

3.3 User experience: reference documentation for types and expressions

The PIE DSL is specified in meta-DSLs, which means that the formal specification of the PIE DSL is also the implementation of the PIE DSL. At the start of this thesis, the PIE DSL had no documentation besides the formal specification in the meta-DSLs. However, a formal specification is not suitable as documentation for a language. Formal specifications are precise, but they are not easy to read. In the case of the PIE DSL, the formal specification is grouped by processing step instead of by language feature. For example, to learn about lists from the specification, one has to look up the syntax in SDF3, the static semantics in Statix, and what they are in the Java documentation, after looking up what Java code is generated in Stratego. And that process is still relatively straightforward compared to more abstract language features. For example, the module system is spread over the module statement, imports, and type and function references, and the type system interacts with literally every other language feature, and also has to interoperate with the Java type system.

The solution is to not use the formal specification as documentation. Instead, we wrote prose documentation, grouped by language feature, and with cross-references to other language features as appropriate.[10]

[10]The documentation is available on the Spoofax website, `www.spoofax.dev/references/pipelines/`. Note: The 'www' is important

Chapter 4

Evaluation

The PIE DSL was made to improve on Java for implementing PIE pipelines. Java is a general language, which means it can be used in many domains, but also that it does not perform excellently in all domains. In the case of PIE, Java has a lot of boilerplate and lacks domain knowledge that could be used to prevent some bugs. The PIE DSL *is* specialized towards the PIE framework, so it should be able to reduce boilerplate and prevent bugs related to it. These objectives of the PIE DSL lead to the question: 'Is the PIE DSL better than Java for expressing PIE pipelines?'

This chapter divides this question into three evaluation questions in section 4.1, describes the case study that is used to evaluate the questions in section 4.2, and answers the questions in section 4.3.

The case study focuses on the capabilities of PIE DSL 2 and how they compare to Java. Section 4.4 compares PIE DSL 2 on a theoretical basis to PIE DSL 1 within the context of the Tiger pipeline from the case study. It looks at the language features each task uses and whether they can be expressed or worked around in each version of the DSL.

4.1 Evaluation questions

To evaluate this question we will answer the following questions:

1. Does the DSL have less boilerplate than Java?
2. Does the DSL provide less opportunities for mistakes than Java?
3. Is the DSL easier to understand than Java?

The following paragraphs justify and elaborate upon the evaluation questions, and describe our methods for evaluating these questions.

Question 1: Does the DSL have less boilerplate than Java? Boilerplate is difficult to rigorously define. As an operational definition "code that does not convey any information to the programmer but which is necessary to make the program work" is close enough. Because boilerplate does not convey any useful information to the programmer, it is useless for understanding the program, yet it still wastes time every time someone reads it before figuring that out. It also makes it harder to find the useful parts of the program. As such, language design should aim to minimize boilerplate.

Because boilerplate is ill defined, it is impossible to measure the amount of boilerplate in a project directly. However, it is possible to compare two projects to see which one has more boilerplate. We can use the fact that we implement the same pipeline in both the PIE DSL and Java with the same underlying model and framework, therefore both projects express

the same information. Any differences in code size do not change the amount of information, so they must be the result of boilerplate. We measure code size in multiple ways: lines of code and amount of characters. Both of these are counted including and excluding layout, and including and excluding library code.

Layout is whitespace, comments and JavaDoc. Layout can separate identifiers and keywords, but consecutive layout does not serve any purpose for the compiler. Most consecutive layout is from indentation. Java and PIE DSL have different conventions for indentation: Java uses 4 spaces per tab, PIE uses 2. This difference conveys no information to programmers, so we also measure the code size excluding layout. We exclude useless layout by counting consecutive layout as a single character. For example, using ␣ as layout character, 1␣2␣3␣␣␣4␣␣5 counts as 8 characters: 5 digits and 3 "characters" of layout.

We also differentiate between library and project code. In Java, library code is literally library code and is simply all Java code that is not part of the project. As such, it is not counted either way. In the PIE DSL, Java library code still needs to be declared in the DSL as foreign Java functions and data. However, these declarations can be reused between projects and are more stable than regular code. Furthermore, we hypothesize that it is a good idea to implement a tool that can generate these declarations from the Java code. Measuring the amount of code that is used for libraries provides data for this hypothesis, which can be used to decide whether to implement such a tool.

Question 2: Does the DSL provide less opportunities for mistakes than Java? Mistakes during coding are inevitable, and one of the jobs of the IDE, language design and the ecosystem is to prevent or point out as many of these mistakes as early as possible. For this question, 'mistakes' refers to general mistakes, and they can be defined as 'unintentionally suboptimal code'. This question is answered on a theoretical basis by comparing the possible mistakes in these languages. Languages are evaluated based on the number of mistakes and the stage where each mistake is detected. We will focus on differences between languages, so mistakes with the same severity that are detected at the same stage in both languages are not considered. Mistakes can be detected at different stages of developing the code, and when they are detected they can be reported with varying severities.

Severities of mistakes Most editors consider three levels of severity: errors, warnings and notes. Languages agree on what errors should be used for, but there are different ideas on warnings and notes.

An error can be defined as 'a mistake which prevents successfully compiling or running the code'. Errors are almost always derived from illegal code as defined by the language specification.

Java uses warnings for things that are not checked by the type checker, but which the developer would typically expect to be checked. For example, conversion from a raw type to a parameterized type will give a warning, because this operation is unsound and will not be checked (not even at runtime). It has an annotation `@SuppressWarnings` that suppresses the specified warnings in a certain part of the code. The Java language specification does not specify anything about notes, but IntelliJ treats them as suggestions to improve the code. For example, on `x == true` it will show a note with the suggestion to use `x` instead.

The PIE DSL takes a different approach. False positives erode the trust of the developer in warnings, especially when they cannot be turned off, as is the case for the PIE DSL. Additionally, statements for suppressing warnings may become outdated without being removed. One of the pillars of the PIE DSL is to only give warnings if code is definitely absolute nonsense. This means that there can be no false positives, and every warning is something that can and should be fixed. Notes are used to note things, i.e. bring something to the attention of the developer, not necessarily a mistake but something to consider. For example, the type

```
data Foo[Foo] = foreign Java Foo {
  // Within these curly braces, Foo refers to
  // the generic parameter, not the datatype
}
```

Figure 4.1: PIE DSL code that will generate a note.

checker gives a note when declaring a generic parameter with the same name as a visible type. Figure 4.1 shows an example of code that would generate such a note. It declares a datatype named `Foo` with a generic parameter also called `Foo`. The generic parameter hides the declaration of the datatype `Foo` within the definition of that datatype, so it might be better to use a different name for the generic parameter.

Stages of mistake detection There are four stages where mistakes can be detected, in order of earliest to latest: in the editor, compile time, runtime or undetected. In general, the earlier a mistake is detected, the cheaper and easier it is to fix.

Mistakes in the editor are detected in the editor. The user gets notified about them as they are typing, at the location of the mistake. These mistakes have very little cost to correct.

A compile time mistake is caught by the compiler at build time. The user has started a build and likely has to wait a few minutes for the result. Solving the mistake may take some time, depending on how well the compiler reports what the cause of the mistake is and how well it explains how to resolve it. For Java, this stage is rare, almost all mistakes are either caught in the editor or not caught until runtime. For the PIE DSL, these mistakes can occur due to legal PIE DSL code for which the PIE DSL compiler generates illegal Java code, after which the Java compiler will give an error.

A runtime mistake is an exception thrown at runtime. These mistakes are time consuming to solve, because they require building and running the system once to detect the bug, and then again while fixing the mistake. It may also be hard to determine the cause of and the solution to the mistake.

Finally, a mistake that is undetected is called a bug. The code runs to completion, but it returns incorrect results. Bugs may be entirely inconsequential or cause tremendous damage. Finding the root cause of observed incorrect behavior becomes harder and costlier the further away it is from the observed behavior.

Question 3: Is the DSL easier to understand than Java? This compares the DSL to Java. Objectively evaluating this would require a user study. That is outside the scope of this thesis. Instead, we will compare the DSL code and the Java code in the case study and make a subjective judgement of whether the DSL is easier than Java.

4.2 Case study

Tiger is a small functional language introduced in Appel (1998).[1] Spoofax 3 is the third major version of the Spoofax language workbench, and uses PIE as its build system. A Tiger language specification project is used as an end-to-end test for Spoofax 3. It contains several PIE pipelines with tasks for commands in Spoofax, for example, a command to analyze a file with Statix and show the scope graph.

[1] A language reference manual for Tiger can be found at `http://www.cs.columbia.edu/~sedwards/classes/2002/w4115/tiger.pdf`. The Spoofax language specification for Tiger can be found on Github: `https://github.com/MetaBorgCube/metaborg-tiger/tree/master/org.metaborg.lang.tiger`

The tasks in this project are implemented in Java. For this case study, we created a new version where the tasks are implemented in PIE DSL 2 instead. This allows us to compare the PIE DSL to Java to answer the evaluation questions in the next section.

There are two options for the DSL version: implement as many of the tasks as possible in the DSL, or only implement in the DSL when it makes sense to do so. The latter is decided by a subjective judgement, which is mostly reached by trying to minimize boilerplate without sacrificing too much code quality. While expressing as many tasks as possible in the DSL is interesting for the expressive power of the DSL, for the evaluation we want to know if the DSL is better than Java for expressing pipelines, so we need the best possible version of both projects. A simple Java class with some boilerplate is better than a simple PIE DSL task with a lot of boilerplate in Java helper classes.

When the case study was executed, the DSL did not have generics and injected values yet. After these were added to the DSL we unfortunately did not have time to update the case study. This means that several more tasks should now be expressible in the DSL than were expressible in the case study. Section 4.4 has a detailed analysis of the language features each task uses and what tasks should be expressible now that generics and injected values are implemented.

This case study has 19 tasks. One task was implemented in the DSL. One task uses resource dependencies (see subsection 4.4.5) in a way that cannot be expressed. The other 17 tasks use injected values (see section 2.1.1 and subsection 3.1.5), which could not be expressed or worked around at the time of the case study. The data from the case study can be found in appendix A.

The code for the case study can be found at github: `https://github.com/MeAmAnUsername/spoofax-pie/releases/tag/tiger-case-study-v1`. The case study itself can be found in `example/tiger/manual`. This includes `tiger.spoofax`, the version with the full Java code, and `tiger.newpie.spoofax`, the PIE DSL version. The code to count the boilerplate can be found in `example/evaluation`.

4.3 Answers to evaluation questions

This section answers the evaluation questions. It also answers the main question: 'Is the PIE DSL better than Java for expressing PIE pipelines?'

4.3.1 Question 1: Does the DSL have less boilerplate than Java?

```
module org:spoofax:interpreter:terms

data IStrategoTerm = foreign java org.spoofax.interpreter.terms.IStrategoTerm {}
```

Figure 4.3: The library file that re-declares `org.spoofax.interpreter.terms.IStrategoTerm` from Java in the PIE DSL.

Data We could only re-implement one task in the PIE DSL. An overview of the result of moving that one task is shown in Table 4.2, the full data of the case study can be found in Appendix A. Whether the amount of lines and characters decreased depends on whether we include libraries in the PIE DSL. These libraries consist entirely of foreign java declarations for Java libraries. An example is shown in Figure 4.3. As can be seen, this library file is entirely boilerplate. Library files like these greatly increase the boilerplate for the PIE DSL.

Value	Difference	Difference (%)
total lines including libraries and layout	+8	+0.34 %
total lines including libraries, excluding layout	+1	+0.05 %
total lines excluding libraries, including layout	-19	-0.82 %
total lines excluding libraries and layout	-16	-0.83 %
total characters including libraries and layout	+570	+0.63 %
total characters including libraries, excluding layout	+639	+0.79 %
total characters excluding libraries, including layout	-486	-0.53 %
total characters excluding libraries and layout	-407	-0.51 %

Table 4.2: An overview of the reduction in boilerplate due to switching from pure Java to also using the PIE DSL when appropriate. 'Excluding layout' means that consecutive layout is counted as one character, and lines with only layout are excluded for the line count. Libraries are always excluded for Java, 'including libraries' only applies to the PIE DSL. Note that for the percentages, the baseline value is the total amount in Java. Since this changes when including or excluding layout, the percentage values including and excluding layout have different baselines and cannot be meaningfully compared.

Because writing these libraries with foreign java declarations is required for now, we will consider the counts including libraries as the 'main' counts.

When we look at the percentages with the libraries included, both the line and character counts increase slightly. This leads to the answer that the PIE DSL does not have less boilerplate than Java. In fact, a reasonable answer based on just the data is that the PIE DSL adds boilerplate. This answer comes with several caveats, which are discussed in *Caveats*. We do not expect this trend to continue when moving multiple tasks to the DSL, and discuss our reasoning in *Expectations when moving multiple tasks*. Our overall conclusion for this evaluation question can be found in *Conclusion*. Finally, the next paragraph looks into the data when we exclude libraries from the measured code size.

When we disregard code in the libraries, there is a modest reduction in code of 0.82%. If we extrapolate this data to 19 tasks[2], this becomes $19 \times 0.82\% = 15.58\%$, which is a respectable reduction in code. We used the data including libraries because the libraries currently have to be written by hand. These data show that automatically extracting the PIE DSL foreign declarations from Java would bring a large reduction to the boilerplate that has to be written by hand.

Caveats There are four caveats, from two perspectives, for the answer to evaluation question 1. The first two relate to our sample size of 1 task, the other two to including libraries for the code size.

First, the particular task we moved did not have a huge amount of boilerplate. The boilerplate that was removed includes imports for PIE classes and nullability[3], the signature of the task, the identifier, and the constructor. While this reduces the amount of lines from 38 to 19, this is not enough to offset the overhead of declaring the types and functions that the task uses in separate libraries. If, for example, it would have had more than one parameter, it would have needed a nested `Input` class to aggregate parameters. Such an input class easily takes 21 lines in Java[4], while taking 0 additional lines in the PIE DSL. It also did not use injected values, which necessitate a field and constructor for the task in Java, so 5 lines

[2]Extrapolating here is safe, because we expect the reduction in code size to become more pronounced, and we do not expect that there is much Java overhead that can be removed when the last task is removed from Java.

[3]`@Nullable`, an annotation to track nullability, and `Objects`, to call `Objects.requireNonNull`

[4]These 21 lines are divided as follows: class name: 1, fields: > 2, constructor: > 4, hashCode: 3, equals: > 5, toString: > 4, closing bracket: 1, newline: 1

in Java vs 1 line in PIE (it can be done on the same line, but in most cases the line with the function declaration becomes too long and needs to be split up).

That means that every element is used only by that task, and the overhead in the form of module statements is averaged over just this one task. We cannot conclude from the data what will happen if more tasks are moved from Java to the PIE DSL. Moving more tasks may require adding more library files or adding new declarations in the existing files, but presumably the tasks do not deal with completely isolated areas, so some of the declarations should be shared between tasks. This should mean that the overhead of the libraries should not be as severe when using multiple tasks. Therefore, we expect that moving multiple tasks will result in a slight decrease in line and character counts, instead of the current slight increase we see in this case study.

Furthermore, libraries are arguably not part of the project, they are separate projects. If the creator of a Java library also provides a PIE library with foreign java declarations, the boilerplate for users of that library becomes less than a line per element used from the library.[5] Even if the provider of the library does not provide a PIE DSL version with the Java version, it takes only one person to write and maintain such a library. Presumably there are multiple people who want to use the library: if there were not, it would not need to be a published library.

Finally, the boilerplate in the libraries carries less negative consequences than boilerplate within the task file. We care about boilerplate because it impedes quick reading and easy modification of code, but for libraries, these concerns are alleviated. First, libraries are more stable than other code, so you rarely have to modify them, only read them or add more elements. Next, when just browsing the library, these PIE libraries provide an alternative to browsing the Java library. The Java library has implementations and JavaDoc, while the PIE DSL library can provide an overview of the available functions and methods. It is still possible to browse the Java library, so users can just open the version of the library that is best for their use case. Finally, to use an element from the library when you already know the name and location, you can just add an import without ever opening the library.

Expectations when moving multiple tasks The line count including libraries and layout increases by 8 lines, 0.34% of the total. This is a slight increase, but not that significant overall. However, we moved just one out of 19 tasks to the DSL. If we take that into account and extrapolate to moving all 19 tasks with a 0.34% increase in line count, we would grow the code by $0.34\% \times 19 = 6.46\%$ in line count. This value would be a noticeable increase in code size, but extrapolating like this is almost certainly not a good representation of what would actually happen.

We expect that moving more tasks from Java to the PIE DSL will not increase, but decrease boilerplate. We expect this due to two compounding factors. First of all, we expect that other tasks have more boilerplate that can be removed compared to the task we implemented now. The 19 tasks have 5 Input classes, each of which could be replaced with a simple parameter list. Many tasks are not implemented in the PIE DSL in this case study because they use dependency injection and generics. Dependency injection also adds some boilerplate, so when these tasks are implemented in the DSL, that is removed too.

The second reason that we expect a reduction in boilerplate is because the DSL has a small amount of overhead. This overhead does not need to be added again when moving more tasks to the DSL, so that means that the slight increase in lines can become a slight decrease in lines.

To give some tentative experimental evidence for these claims, Appendix D shows ideal DSL implementations for 16 of the 19 tasks. This takes 321 lines of code. The foreign Java

[5] There is some small, constant overhead to include the library in the PIE DSL code of your project. There is also at most one line per imported element, less if you use multi-imports.

declarations take another approximate 400 lines of code. The three tasks which are not implemented are 132 lines of code. Finally, let's be generous and say that helper methods in Java take an additional 300 lines of code. That is approximately 1150 lines of code, only half of the 2327 lines of the full Java implementation.

Conclusion The data shows that when only moving a single task, and including libraries and layout, the code size increases by eight lines. Thus, for this data there is more boilerplate when using the PIE DSL.

However, we do not expect this trend to continue with multiple tasks. This is because the particular task we implemented with the DSL had a below average amount of boilerplate, and because the overhead of moving to the DSL is now borne in its entirety by this single task. Moreover, there are arguments to be made to exclude libraries when measuring the code size. First, library code can be shared between multiple projects, so it is arguably not part of this project. Second, library code carries less negative consequences than boilerplate within the task file. The main reasons to avoid boilerplate are less pertinent in library code. If we exclude libraries from our measurements, there is less boilerplate. We expect this effect to become more pronounced with multiple tasks.

4.3.2 Question 2: Does the DSL provide less opportunities for mistakes than Java?

Mistake	The Java language		IntelliJ & Gradle		The PIE DSL	
	detection stage	severity	detection stage	severity	detection stage	severity
Incorrect foreign Java declarations	not applicable in Java				Java compiler	error
Using a null value when a non-null value is required	runtime	exception	runtime	exception	editor	error
Using a nullable element without checking for null	undetected	-	editor	warning	editor	error
Comparing two types that cannot be equal	undetected	-	editor	warning	editor*	error
List comprehension over empty list	undetected	-	editor†	warning	editor†	warning
Concatenating an empty list	undetected	-	editor†	warning	editor†	warning
Incorrect imports	compiler	error	editor	error	editor	error
Missing imports	compiler	error	editor	error	editor	error
Duplicate imported names	compiler	error	editor	error	editor	error
Duplicate imports	not applicable - Java does not allow renaming imports				undetected	-
Unused imports	undetected	-	editor	warning	undetected	-
Empty multi-imports	undetected	-	editor	warning	editor	warning
Multi-imports with single element	undetected	-	undetected	-	editor	warning
Renaming import to the same name	not applicable - Java does not allow renaming imports				editor	warning
Unused elements	undetected	-	editor	warning	undetected	-
Duplicate task IDs	runtime	exception	runtime	exception	runtime‡	exception
Use a language keyword as name in a non-ambiguous position	compiler	error	editor	error	allowed by language design	
Use a language keyword as name in an ambiguous position	compiler	error	editor	error	editor	error
Use a Java keyword as name	compiler	error	editor	error	Java compiler	error
Warnings from the Java ecosystem	undetected	-	editor	error	undetected	-
Generated source code	IntelliJ can generate some of the boilerplate, which reduces opportunities for mistakes					

* The PIE implementation is incorrect: it disallows valid comparisons
† It does not detect all cases, so there are false negatives, but no false positives
‡ This is not a mistake by the developer but a bug in the compiler

Table 4.4: Overview of possible mistakes in Java and the PIE DSL where the detection stage or severity differ. The Java language refers to the Java language as specified by the Java Language Specification. IntelliJ and Gradle refers to the Java ecosystem, which includes IntelliJ, Gradle, the Checkerframework Gradle plugin, and the Java compiler.

Table 4.4 provides an overview of the possible mistakes and whether they are caught by Java as specified by the Java language specification, Java and its ecosystem, and the PIE DSL.

If we assume that the foreign Java declarations are imported correctly and ignore known and unknown bugs, the PIE DSL is safer than pure Java. It has mostly the same semantics as Java, but explicit nullability and more restrictive semantics for comparisons. Java allows comparing anything to anything, PIE only if there are overlapping values.

This has three caveats. First, we assume that 'foreign Java declarations are imported correctly', which will almost certainly not be the case when manually writing the declarations for a large amount of Java elements. Secondly, there are known bugs in the implementation of the PIE DSL, and there are almost certainly unknown bugs as well. Java has been used by by billions of devices every day over many decades, most bugs that are likely to be hit have been resolved already. Finally, the more advanced type system features in the PIE DSL that are not specified by the Java language are added by tools in the Java ecosystem. For example, we use a compiler plugin that checks nullability, and IntelliJ gives warnings when a comparison seems fishy.

The rest of this section will elaborate on all mistakes in Table 4.4. For each mistake, we will explain how it can happen and compare the stage where it is detected and its severity between Java, the Java ecosystem, and the PIE DSL.

Foreign Java declarations Foreign Java declarations are required to interoperate with Java. They declare the signature of data, functions and tasks with the fully qualified name in Java. However, neither the signature nor the fully qualified name are checked by the PIE DSL. It just assumes that they are correct, and if they are not, it will generate invalid Java code. This invalid Java code will lead to compile errors in the Java compiler.

Explicit nullability Explicit nullability in Java is added by compile time annotations from `org.checkerframework:checker-qual-android`. By adding explicit annotations like `@Nullable` or `@NonNull`, nullability is made explicit and checked by the framework. If nullability is violated, the framework gives a compile time warning, which is also shown in the editor if it has integration with the build system (as is the case for us with Gradle and IntelliJ). This is less strict than the PIE DSL, which gives an error. The framework also has some more advanced notions of nullability, such as `@MonotonicNonNull`, which specifies that a field or variable will never become null after it has had a non-null value. This is not applicable in the PIE DSL because all variables are immutable and fields do not exist.

The framework has a few shortcomings compared to the PIE DSL. First of all, it is more verbose because nullability is tracked via annotations with full names, while the DSL just uses a question mark (`?`) to indicate that a type is nullable. The framework does allow setting a default nullability, but that only removes annotations for a single nullability.

Another shortcoming is that the nullability does not properly work with generic types. For example, PIE sessions have a method `<O extends @Nullable Serializable> O require(Task<O> task)`. Because `O` is declared `extends @Nullable`, the framework gives a warning when using the return value without checking if it is null. However, with `Task<@NonNull String>` the return value will never be null, yet the framework still gives a warning if we do not check for null[6].

Finally, there are bugs in the implementation of the framework. For example, the field `parse` in `mb.jsglr.pie.JsglrParseTaskInput.Builder` should be non-null when the `Builder` builds the `JsglrParseTaskInput`. This is checked by `checkBuildPossible`. Even when adding a condition to that method that specifies that `parse` is non-null after that method, there are still warnings that method calls on `parse` may produce a `NullPointerException`. These false positives are a bigger problem than they may appear at first glance, because false positives like this results in developers ignoring or suppressing warnings. An example is a bug in

[6]Looking at the documentation at `https://checkerframework.org/releases/3.16.0/manual/#polymorphism`, it seems like this behavior is unintended

```
// static type of xs is empty*
val xs = [];
[1, 2] + xs; // warning
```

(a) There is a warning when the static type is an empty list.

```
// static type of xs is int*, not empty*
val xs: int* = [];
[1, 2] + xs; // no warning
```

(b) There is no warning when the type is specified explicitly.

Figure 4.5: The PIE DSL only gives a warning on appending an empty list if the type is an empty list.

the code we wrote where a null check was not properly renamed, which meant that there really was a possibility for a `NullPointerException` but the warning for it was suppressed. We explain this example in more detail in *Warnings from the Java ecosystem*.

In the PIE DSL, nullability is part of the type system and violations result in errors. False positives should only be possible in case of bugs, but even when there are, there is no way to ignore it, it has to be explicitly handled. Explicitly handling it requires just the single character '`!`', or 3 characters if the offending expression has to be enclosed in parentheses. When the code is no longer deemed nullable (either because the false positive becomes a true negative or because the code is updated), the non-null check becomes an error as well, which promotes clean code.

Comparisons Java compares Objects with the `equals(Object)` method, which takes any Object and compares it to **`this`**. The PIE DSL has more restrictive semantics for comparisons. The Java expression `obj1.equals(obj2)` is expressed in the PIE DSL as `obj1 == obj2`. This is only allowed when the type of `obj1` and the type of `obj2` share some values, excluding `null` and the empty list. Multiple inheritance has not been implemented in the PIE DSL yet, which means that two types only share values (besides `null` and the empty list) if one is a subtype of the other. The current implementation for comparability assumes that that means that two types can only be equal if one is a subtype of the other or vice versa, but that is incorrect. For example, Java Lists are equal when their elements are equal, regardless of the generic arguments or even the specific implementing class. Java does not give any warnings when comparing values. IntelliJ gives warnings when comparing two types that do not share values.

Empty lists The PIE DSL keeps track of empty lists for implementation reasons. This enables it to give warnings on nonsensical operations with empty lists. The first case is a list comprehension over an empty list, for example `val xs = []; [x+1 | x <- xs]`. The equivalent Java expression, `ArrayList<Integer> xs = new ArrayList<>(); xs.stream().map(x -> x+1)` also gives a warning in IntelliJ, although there it is a warning about never writing to the empty ArrayList `xs`.

The second case where the DSL gives warnings is concatenating an empty list to another list, as is shown in figure 4.5a. The PIE DSL does not use data flow analysis, which means that it will not give warnings for empty lists if the type is set to something else, as shown in figure 4.5b.

IntelliJ does not keep track of empty lists, but it does check if a list is updated. This has the same effect if the empty list is assigned to a variable, as shown in figure 4.6a. However, if there is no variable then there is no warning, which can be seen in 4.6b The check for whether a list is updated is not perfect either. For example, if the empty list is used in a stream it does not give a warning, as is shown in figure 4.6c. This is probably because a stream *can* update the list, even though it is bad practice to update the source of a stream from within the stream.

```java
final List<Integer> xs = Arrays.asList(1, 2, 3);
final List<Integer> ys = new ArrayList<>(); // warning
          // Contents of collection 'ys' are queried, but never updated
final List<Integer> merged = new ArrayList<>(xs);
merged.addAll(ys);
```

(a) IntelliJ gives a warning if a variable holding a list stays empty.

```java
final List<Integer> merged = Arrays.asList(1, 2, 3);
merged.addAll(new ArrayList<>()); // no warning despite this clearly not doing anything
```

(b) IntelliJ does not give a warning when an empty list is appended to another list.

```java
final List<Integer> xs = Arrays.asList(1, 2, 3);
final List<Integer> ys = new ArrayList<>();
final List<Integer> merged = Stream.of(xs, ys)
  .flatMap(Collection::stream)
  .collect(Collectors.toList());
```

(c) IntelliJ also does not give a warning when appending the two lists via stream.

Figure 4.6: IntelliJ gives a warning for certain cases of appending an empty list.

Imports There are a few types of mistakes with imports. All warnings and errors in this paragraph appear in the editor.

Incorrect imports The first type is an incorrect import, which tries to import an element that does not exist: `import org:example:cs:complexity_theory:printProofThatPEqualsNP`. These errors are caught by both Java and the PIE DSL.

Missing imports The second type is a missing import, where an element is used without being imported (this results in an undefined element error). These errors are also caught by both Java and the DSL. Additionally, IntelliJ will suggest a quick-fix to add an import for the element.

Duplicate imported names The PIE DSL gives an error if two imports declare the same name in the same namespace. Figure 4.7a shows an example where two imports rename to the same name.

Java does not give an error when multiple imports make the same name available. It only gives an error when the name is used and the use is ambiguous. The code in figure 4.7b may or may not give an error, depending on the code with the declarations. When classes have a method `test` without any parameters as in figure 4.7c, the code gives an error "Ambiguous method call. Both test() in A and test() in B match". When the two methods have different number of parameters as in 4.7d, the call is not ambiguous, and there is no error.

Duplicate imports Duplicate imports import an element multiple times under different names, for example `import org.example.{cs as compsci, cs as computerScience}`. This is not caught by the DSL. Renaming is not possible in Java, so this issue cannot occur in Java.

Unused imports Unused imports result in warnings in IntelliJ. They are not caught by the PIE DSL.

Empty multi-imports The PIE DSL has multi-imports, which import multiple named elements with a single import statement. Java does not have multi-imports, but it does have

```
import org.example.dataTypeA as someType
import org.example.dataTypeB as someType
```

(a) Two PIE DSL imports that make the same name available.

```
package org.example

import static org.example.A.test
import static org.example.B.test

class Test {
  public static void example() {
    test();  // error if this call is ambiguous
  }
}
```

(b) Java code with a possibly ambiguous call to an imported method. Whether the call is ambiguous depends on whether the declaration code is 4.7c or 4.7d below.

```
package org.example

class A {
  public static void test() {}
}
class B {
  public static void test() {}
}
```

(c) Declaration code that results in an error in 4.7b.

```
package org.example

class A {
  public static void test() {}
}
class B {
  public static void test(int num) {}
}
```

(d) Declaration code that does not result in an error in 4.7b.

Figure 4.7: Examples of duplicate imported names in the PIE DSL and Java.

the star import, which imports everything in a class. The PIE DSL gives warnings for empty multi-imports. The equivalent in Java is a star import from a class with no elements. Because the class has no elements the import does nothing, and the import is therefore necessarily unused. IntelliJ does not give a warning that the import imports from a class without elements, but it does give a warning that the import is unused.

Multi-imports with a single element The DSL also gives warnings for multi-imports with a single element. IntelliJ does not give any warnings for a star import for a class with a single element.

Renaming to the same name Finally, the PIE DSL allows renaming imports. It gives a warning when renaming an import to the name it already had, e.g. `import org:example:foo as foo` or `import org:{example:bar as example:bar}`. Java does not allow renaming imports, so mistakes in renaming are not applicable to Java.

Unused elements Unused imports, variables, parameters, functions and data types are all caught by IntelliJ, but not by the PIE DSL. The scope of imports, variables and parameters is restricted to within the file, so analysis can determine that the element is definitely not used. That is not the case for functions and data types. The PIE DSL does not have a set entry point, so there is no way to know which functions are meant to be called as main task. Additionally, functions and data types can be imported from other modules. This means that functions and data types might be intentionally unused, for example with libraries. The PIE DSL does not give warnings on code that might be correct, therefore unused functions or data types do not give warnings. Note that this same problem occurs in IntelliJ, which means that libraries need to mark everything with `@suppresswarnings("unused")` to suppress the warnings.

Task IDs The PIE framework requires that every task definition has a unique identifier, the task ID. This is checked at runtime, while building the PIE instance. A common mistake in Java is not to use the fully qualified name, but just the simple name of the class: `getClass().getSimpleName()` instead of `getClass().getName()`. This is mostly caused by IntelliJ, which will suggest `getSimpleName` before `getName`. Task definitions with the same name but in different packages can now not be used in the same project. This can happen in projects that use multiple Spoofax languages, where every language project has tasks such as 'Parse' and 'Analyze'. The PIE DSL does not have a way to specify the task ID. Instead, it is simply the task name. This means that it suffers from the same problem as Java when simple names are used, with the added complication that this is a bug in the compiler[7], not a mistake by the developer. A workaround is to rename the task, but that is annoying and not always possible.

Keywords Keywords are words with special meaning within the language. They are often prohibited at certain locations because they would introduce ambiguity. Java has a list of reserved words, which includes all keywords, plus some words reserved for future use or to avoid confusion. Reserved words cannot be used as names of variables, fields, methods, classes etc., even when their use would not be ambiguous. For example, naming a variable `throws` does not introduce any ambiguities, but it is still disallowed.

The PIE DSL allows using keywords as names, unless that would result in ambiguities. For example, `module` is a keyword but its use is not ambiguous, so it can be used as any name. `return` is not ambiguous as a module name, so that is allowed. However, a function named `return` is ambiguous, because it would be unclear whether the following is a return expression or a function call: `return (47)`.

The PIE DSL does not assign special meaning to Java keywords. Assigning special meaning to Java keywords would couple the design of the DSL to Java, which we want to avoid where possible. Instead, it is the responsibility of the compiler to compile the name to a name that is valid in Java. For example, renaming a variable `class` to `_class`. This is currently not implemented in the compiler. It will keep the same name, which results in an error at Java compile time. This is not a mistake on the part of the pipeline developer, but a bug in the compiler. Nevertheless, for now, it falls on the pipeline developer to work around this bug. The workaround is to not use Java keywords as names in PIE DSL code. Luckily, using a Java reserved word as name is rare, so this rarely comes up practice.

Warnings from the Java ecosystem The Java ecosystem is far more mature than the PIE DSL. IDEs like IntelliJ will highlight issues that are not defined on the Java language itself. It gives warnings on variables that are read, but never written to (flags that are never changed, collections that are never updated and thus stay empty). It gives warnings on variables that are never read. It also gives warnings for things that are possibly intended, but likely bugs. For example, unused classes and methods, method parameters that are always called with the same argument, and collections that are read from but never have anything added to them. Finally, it gives notes with suggestions for better ways to express certain things, for example, `Arrays.asList()` can be replaced by `Collections.emptyList()`. This is different from the PIE DSL, where warnings are always mistakes.

Warnings and notes on correct Java code can be suppressed with an annotation. This has the positive effect that the editor can warn for things that are likely mistakes, not just when it is certainly a mistake. A negative consequence is the possibility that something is suppressed, but later the code is changed in such a way that the suppression is no longer justified. This change does not necessarily have to be in the code where the expression is suppressed. An

[7] There is an open issue to use the fully qualified name, see https://github.com/MeAmAnUsername/pie/issues/65

```
Result<LineCounts, @NonNull Exception> lineCounts = context.require(countLines.createTask(
    input));
if (lineCounts.isErr()) {
    //noinspection ConstantConditions  Safe because isErr() returned true
    return Result.ofErr(lineCounts.getErr());
}

return Result.ofOk(new ProjectEvaluationResult(lineCounts.get()));
```

(a) Original code with a correct warning suppression.

```
Result<LineCounts, @NonNull Exception> lineCounts = context.require(countLines.createTask(
    input));
if (lineCounts.isErr()) {
    //noinspection ConstantConditions  Safe because isErr() returned true
    return Result.ofErr(lineCounts.getErr());
}
Result<ProjectCounts, @NonNull Exception> characterCounts = context.require(countCharacters
    .createTask(input));
if (lineCounts.isErr()) { // BUG! Should be characterCounts.isErr()
    //noinspection ConstantConditions  Safe because isErr() returned true
    return Result.ofErr(characterCounts.getErr());
}

// bug is suppressed here
//noinspection ConstantConditions  Safe because isErr() returned false
return Result.ofOk(new ProjectEvaluationResult(lineCounts.get(), characterCounts.get()));
```

(b) Updated code with an outdated warning suppression.

Figure 4.8: An example of a correct Java warning suppression annotation that becomes incorrect due to a code change somewhere else.

example that we actually ran into can be found in figure 4.8. The original code, shown in 4.8a, is fine. Either `isErr` returns true and we use `getErr`, or it returns false and we use `get`. The problem came when we made a copy-paste error, shown in 4.8b. We forgot to change the `isErr` check from `lineCounts` to `characterCounts`. This meant that we could call `get` when `characterCounts` holds an error, which would have resulted in a `NullPointerException`. The annotation still suppressed that warning, so we were not notified of the mistake.

Generated source code As determined in evaluation question 1, the PIE DSL has approximately the same amount of code as Java. The Java boilerplate that the PIE DSL avoids is mostly generated by IntelliJ with a few keystrokes, which means that this boilerplate in Java does not have many mistakes. Except for the task IDs, every mistake in this generated code either does not matter (e.g. unused imports) or will be caught by the type system in the editor.

The PIE DSL does not have any features to generate boilerplate for source code.[8] All code in the DSL is handwritten, therefore it does not benefit from the editor generating correct code.

Conclusion The PIE DSL is theoretically safer than the Java language. It has stricter semantics for nullability and comparisons. However, compiler plugins and IntelliJ will in many cases catch many more mistakes than are specified by the Java language specification. All caught mistakes and suboptimal code are caught in the editor by tools in the Java ecosystem,

[8] It generates Java code, but that is not source code

which makes the development experience far smoother than with the PIE DSL. The Java ecosystem is also far more mature than that of the PIE DSL, so it catches more mistakes and has many more features, such as tools to correctly generate boilerplate.

Overall, the only class of mistakes that the PIE DSL handles better than the Java ecosystem is nullability.

4.3.3 Question 3: Is the DSL easier to understand than Java?

Since there is only one task implemented in the PIE DSL, we can examine it in detail.

File setup First of all, because tasks are generally small (less than 20 lines), the convention is to put all tasks from a Java package in the same module. In this case study, the task `TigerCompleteTaskDef`, which is part of the package `mb.tiger.spoofax.task.reusable`, can be found in module `mb:tiger:spoofax:task:reusable` in file `reusable.pie`. This could make development slightly easier when a few small tasks are put together. It could also make development slightly harder because the file does not just contain the tasks you are interested in, but also unrelated tasks. In Java, all tasks have their own files, so one can just open editors for only the tasks that are relevant.

The other part of the file system setup is the files with foreign Java declarations. Foreign Java declarations are a form of boilerplate because they are not required to understand the code. Since they are rarely relevant, it was deemed better to put them in separate files. This also makes them reusable and easily replaced with a library of foreign java declarations. The current convention is to follow Java and put every PIE file in a directory structure corresponding to the module name, and the module name corresponds to the Java package name. For example, a function `mb.common.style.StyleName#fromString` is put in the module `mb:common:style` in the file `src/main/pie/mb/common/style.pie`.

File locations do not influence anything in the PIE DSL, so an alternative file setup would be a directory `pie` with subdirectories `main` and `libraries`. We do not have enough experience to know which file system setup is better. It likely does not matter too much.

Foreign Java declarations Declarations for foreign Java functions are consistent with normal PIE declarations by design, and the implementation is a reference to a Java class. Not all Java features exist in the PIE DSL. For example, **`void`** and **`throws`** are unsupported by the PIE DSL. However, correctness of foreign Java declarations is not checked, so it is possible to smuggle a bit and use a different signature.

For example, generics can sometimes use the instantiated version of a class. In the case study, the Java function `public static <E> ListView<E> of(List<? extends E> list)` contains the generic type `List<? extends E>`. This type is replaced with `CompletionProposal*` in the DSL. The function is declared as `func listViewOf(proposals: CompletionProposal*)-> ListView = foreign java mb.common.util.ListView#of`. This works because no code is generated for foreign Java functions, which means that the DSL does not generate code with that type. It only checks that the actual type of an argument of a call to `listViewOf` is a subtype of the declared type `CompletionProposal*`. The declared type in the DSL is also a subtype of the declared type in Java, so the declaration is type safe too, even though this is not checked. Knowing that this substitution works requires thinking about how the type will be used by static analysis and knowledge of the implementation of the static analysis.

As a negative example, it is impossible to replace **`void`** functions with `unit`. The compiler assumes that every function has a return value, and always generates code that saves the result of a call to a variable, even if the result is unused. This results in compile errors in Java when the return type of the method is **`void`**. Knowing that this substitution does not work requires knowledge of the compiler implementation.

These exceptions and their requirement for knowledge of implementation details makes foreign Java declarations a tricky part of the DSL.[9] Java does not have such a feature mismatch with Java, and as such can easily use classes and methods from Java files.

Imports

```java
import mb.common.style.StyleName;
import mb.common.util.ListView;
import mb.completions.common.CompletionProposal;
import mb.completions.common.CompletionResult;
import mb.pie.api.ExecContext;
import mb.pie.api.Supplier;
import mb.pie.api.TaskDef;
import mb.tiger.spoofax.TigerScope;
import org.checkerframework.checker.nullness.qual.Nullable;
import org.spoofax.interpreter.terms.IStrategoTerm;

import javax.inject.Inject;
import java.util.Objects;
```

(a) The imports in Java.

```
import mb:common:style:{StyleName, styleNameFromString}
import mb:common:util:{ListView, createEmptyListView}
import mb:completions:common:{CompletionProposal, createCompletionProposal,
    CompletionResult, createCompletionResult}
import org:spoofax:interpreter:terms:IStrategoTerm as Term

func listViewOf(proposals: CompletionProposal*) -> ListView = foreign java mb.common.util.
    ListView#of
```

(b) The imports in the PIE DSL.

Figure 4.9: The imports of `TigerCompleteTaskDef` in Java and in the PIE DSL.

Because all tasks are in the same file, imports for all tasks are merged together. In Java, `CompletionProposal` and `CompletionResult` require two lines to import, as is shown in figure 4.9a. The same two classes require four imports in the PIE DSL, because the constructors are not automatically included in the imports. The PIE DSL includes multi-imports, which allows grouping the data types and functions from a class or package. This allows us to combine the four imports into a single multi-import, as can be seen in figure 4.9b.

The DSL also allows renaming an import, which is used to rename `IStrategoTerm` to `Term`. This saves some typing, is easier to remember, and is just as clear as (if not clearer than) `IStrategoTerm`.

Overall, the PIE DSL does have more features for imports than Java, and those features do make the code a little nicer to read. However, imports are rarely actively used during development, so it does not really have much effect on the difficulty of the PIE DSL.

Task declaration

The Java declaration of the task is shown in figure 4.10a. It has 9 lines, with some duplicate and redundant information. The PIE DSL definition is shown in figure 4.10b. In the DSL, the declaration is a single line with no duplicate information, which makes the declaration in the DSL far easier than Java.

[9] In fact, they are so tricky that we, despite having implemented both the static analysis and the compiler, did not figure this out by thinking about it but just by trying and seeing if it works.

```
@TigerScope
public class TigerCompleteTaskDef implements TaskDef<Supplier<@Nullable IStrategoTerm>,
    @Nullable CompletionResult> {
    @Inject
    public TigerCompleteTaskDef() {}

    @Override
    public String getId() {
        return this.getClass().getName();
    }

    @Override
    public @Nullable CompletionResult exec(ExecContext context, Supplier<@Nullable
        IStrategoTerm> astProvider) throws Exception {
```

(a) The declaration of the task in Java.

```
func TigerCompleteTaskDef(astProvider: supplier<Term?>) -> CompletionResult? = {
```

(b) The declaration of the task in the PIE DSL.

Figure 4.10: The declaration of `TigerCompleteTaskDef` in Java and the PIE DSL.

The `@TigerScope` annotation is for Dagger, the injection framework, and specifies the lifetime of instances of this taskdef. Such an annotation is not generated by the PIE DSL, but that is not a problem when only a single instance is created. The next line specifies the name, that it is a task definition, and the input and output of the task. The nullability for the input and output have to be specified explicitly with annotations. In the DSL, this is done with a single question mark, which is more concise and does not lose much readability[10]. The next 4 lines specify the task ID, which for Java source code is always the fully qualified name of the class. Then there is a line of layout, and finally the last two lines specify the method that will provide the implementation of the task. Given that the second line already specified that this class is a task definition with the input and output, these last two lines provide exactly zero new information.

The PIE DSL is definitely more concise and with better syntax, which makes it far easier to write and to read.

Task implementation

The task implementation for both Java and the PIE DSL are shown in figure 4.11. The implementations are mostly the same, but there are five small differences. First of all, the type of the variable `maybeAst` does not need to be specified in the DSL. The type of getting from that supplier is already specified in the line above: `astProvider: supplier<Term?>`, so repeating it is not necessary.

The second difference is the return statement on the last expression. The PIE DSL does not require (or allow) a return statement, it just returns the last expression in the block. This is just a syntactic decision, it does not really matter for readability.

Next, **`new`** `CompletionResult` and **`new`** `CompletionProposal` are replaced with `createCompletionResult` and `createCompletionProposal` respectively. Not having the concept of constructors simplifies the design of the PIE DSL. Names of the form `createClassName` are slightly harder to read than **`new`** `ClassName`, but overall not making a distinction between functions and constructors makes the DSL easier than Java.

The `StyleName.fromString` and `ListView.of` calls are replaced with specific function calls as well. While these are like the constructor calls in that they create a new instance, in this

[10] If a developer does not know what that syntax means they can look it up in the documentation

```java
@Nullable IStrategoTerm ast = astProvider.get(context);
if (ast == null) return null;   // Cannot complete when we do not get an AST.

return new CompletionResult(ListView.of(
    new CompletionProposal("mypackage", "description", "", "", "mypackage", Objects.
        requireNonNull(StyleName.fromString("meta.package")), ListView.of(), false),
    new CompletionProposal("myclass", "description", "", "T", "mypackage", Objects.
        requireNonNull(StyleName.fromString("meta.class")), ListView.of(), false)
), true);
```

(a) The implementation of the task in Java.

```
val maybeAst = (astProvider.get());
if (maybeAst == null)
  return null;
createCompletionResult(listViewOf([
  createCompletionProposal("mypackage", "description", "", "", "mypackage",
      styleNameFromString("meta.package")!, createEmptyListView(), false),
  createCompletionProposal("myclass", "description", "", "T", "mypackage",
      styleNameFromString("meta.class")!, createEmptyListView(), false)
]), true)
```

(b) The implementation of the task in the PIE DSL.

Figure 4.11: The implementation of `TigerCompleteTaskDef` in Java and the PIE DSL.

case they do it via a static methods instead of constructors. The `StyleName.fromString` becomes `styleNameFromString`, which is the same but without the dot. The Java name feels slightly easier to read because the dot separates it into two smaller words. `ListView.of` gets replaced by two function calls, `listViewOf` and `createEmptyListView`. The PIE DSL did not support generics, so the declaration of the `ListView.of` method specifies the element type of the listview. This makes it hard to reuse the declaration. We could have used `listViewOf([])` to create an empty listview, but that is slightly less efficient than createEmptyListView, and does not show the intend as clearly. Additionally, the plan is to eventually generate PIE DSL libraries from Java libraries semi-automatically, so then both functions would be available anyway. The Java version also uses the specialized method to create an empty ListView, so there is no difference in how the code works, only the name of the function is different. Again, Java's `ListView.of()` and `ListView.of(List)` are slightly easier to read than the DSL's `createEmptyListView` and `listViewOf`.

Finally, Java uses `Objects.requireNonNull` to check that `StyleName.fromString` does not return null, while the PIE DSL simply checks this with `!`. This exclamation mark is easy to miss, but that does not matter because the PIE DSL keeps track of nullability, which means that the editor will tell you if you do it wrong. That in turn means that a developer does not have to actively keep track of the nullability of types, which certainly makes development easier.

Conclusion The DSL is easier than Java for doing simple tasks. It is a small and focused language by design. In the cases where it has more features than Java, using the advanced features is optional. This makes the DSL really easy when the logic you are trying to implement is supported. The issue is when the required functionality is not supported by the DSL, in which case either the task has to be implemented in Java or some workaround has to be used. This often happens when interacting with Java classes that were not designed with the limited PIE DSL feature-set in mind. At the moment, this happens rather often, so while the PIE DSL is simpler than Java, it is not easier for most use cases.

4.3.4 Conclusion

We asked the question 'Is the PIE DSL better than Java for expressing PIE pipelines?' We split this question into three sub-questions and answered each of them separately. The answers to the three sub-questions are summarized below.

1. *Does the DSL have less boilerplate than Java?*
   With only a single task, the reduction in boilerplate for the task definition cannot overcome the overhead from foreign Java declarations in libraries. We expect that the DSL *does* reduce boilerplate when there are more tasks that can be expressed in the DSL, but this expectation could not be confirmed or refuted with the data we have.

2. *Does the DSL provide less opportunities for mistakes than Java?*
   The Java ecosystem is far more mature than that of the PIE DSL. Nullability works better in the DSL despite Java's maturity, because it is built into the PIE DSL, while the implementation in Java still leaves some things to be desired.

3. *Is the DSL easier to understand than Java?*
   The PIE DSL is certainly simpler than Java. However, this only translates into an advantage if the task to be implemented can be expressed in the more restricted feature-set of the DSL. When the task uses features only available in Java, using the PIE DSL requires understanding both the required Java features *and* the workarounds to make it work in the PIE DSL, which makes just doing it in Java far easier.

In conclusion, the PIE DSL is not better than Java for expressing PIE pipelines in its current state, unless your use case only uses features supported by the DSL.

4.4 Comparison between PIE DSL 1 and 2

Chapter 3 provides a detailed look into the changes from PIE DSL 1 to 2 and what problems they solve. However, the changes are described abstractly, without specific references to the case study. This section aims to show how the improvements of PIE DSL 2 enable us to make the PIE DSL better than Java, and why these changes are necessary, by showing how these changes affect the case study. We also use the summary of this analysis to answer the question *how does PIE DSL 2 compare to PIE DSL 1?*

The goal of the DSL is to be better than Java for implementing PIE pipelines. It should achieve this by making tasks easier and more concise to express. As was shown in the case study, it does not achieve this: most tasks cannot be expressed in the DSL. Any features for making task definitions easy and concise achieve nothing if tasks cannot be expressed. On the other hand, the ability to express tasks achieves nothing if expressing the tasks in the PIE DSL is not better than expressing them in Java.

These two observations bring us two goals for the PIE DSL:

- The DSL should support enough language features that tasks can be expressed.
- The DSL should make it more concise to express PIE projects.

For the DSL to be better than Java, it needs to achieve both these goals. The next subsections explain what has been done and what is still to come for both of these goals.

Tasks cannot be expressed because they use language features which are not supported in the DSL. Table 4.12 shows an overview of the language features that each task uses.

	Task name	PIE DSL 1	PIE DSL 2	Generics	Supplier	Subclassing	Injected values	Resource dependencies	Nested class	Higher order functions	Exception handling	Field and enum access	Instanceof	Arrays
1	TigerAnalyze	×	×	•	-	◆	•	-	■	-	†	-	◗	-
2	TigerAnalyzeMulti	×	×	•	-	◆	•	-	-	-	†	◗	◗	-
3	TigerCompleteTaskDef	×	✓	-	•	-	-	-	-	-	■	-	-	-
4	TigerListDefNames	×	~	•	•	-	•	-	-	○	○	○	-	=
5	TigerListLiteralVals	×	~	•	•	-	•	-	-	○	○	○	-	=
6	TigerParse	×	×	•	-	◆	•	◆	-	-	‡	-	-	-
7	TigerStyle	×	~	•	•	-	•	-	-	◗	■	○	-	-
8	TigerCheck	×	~	•	•	-	•	◆	■	◗	■	○	-	-
9	TigerCheckAggregator	×	×	-	-	-	■	‡	-	‡	■	-	-	-
10	TigerCheckMulti	×	×	•	•	-	•	‡	-	‡	■	○	-	-
11	TigerCompileDirectory	×	×	•	•	-	•	‡	■	◗	■	○	-	◗
12	TigerCompileFile	×	×	•	•	-	•	‡	■	○	†	○	-	◗
13	TigerCompileFileAlt	×	×	•	•	-	•	‡	■	○	†	○	-	◗
14	TigerIdeTokenize	×	~	•	•	-	•	-	-	-	■	-	-	-
15	TigerShowAnalyzedAst	×	~	•	•	-	•	◆	-	◗	■	○	-	-
16	TigerShowDesugaredAst	×	~	•	•	-	•	◆	-	○	○	-	-	=
17	TigerShowParsedAst	×	~	-	•	-	•	◆	-	◗	■	-	-	-
18	TigerShowPrettyPrintedText	×	~	-	•	-	•	◆	-	○	○	-	-	=
19	TigerShowScopeGraph	×	~	-	•	-	•	◆	-	○	○	○	-	=
	Total	0	11	14	15	3	18	12	5	14	19	11	2	3

×/✓ The task cannot/can be expressed in the given version of the PIE DSL.
~ The task can theoretically be expressed in the given version of the PIE DSL, but this has not been tested in the case study.
- The feature is not used by the task.
= The feature is not used by the task. It cannot be expressed in the DSL, which prevents a workaround for another feature.
■ The feature is used by the task and expressible in both PIE DSL 1 and 2.
◆ The feature is used by the task. It is not expressible in either PIE DSL 1 or 2, but has a workaround in both.
• The feature is used by the task and expressible in PIE DSL 2.
◗ The feature is used by the task. It is not directly supported by the DSL, but a workaround exists in PIE DSL 2.
○ The feature is used by the task and is not expressible in PIE DSL 2. The use case of the feature can be implemented in a helper method in Java, which can be expressed as a call to a foreign Java function.
† The feature is used by the task. It is not expressible in PIE DSL 2 and does not have a workaround. It is possible to omit the feature with some loss of functionality. See explanation for this feature and this task in text for more information.
‡ The feature is used by the task and is required to express the task.

Table 4.12: An overview of the language features each task in the case study uses.
We consider a workaround better than a Java helper method, so if a feature works with either it will be listed as 'workaround'. Lightgray symbols already worked in PIE DSL 1, gray symbols work since PIE DSL 2, and black symbols still do not work.
'Subclassing' means that the task does not directly implement `TaskDef`, but instead extends a class which does. 'Injected values' are values which are added to the `TaskDef` when it is constructed. 'Resource dependencies' are dependencies on resources, such as files, open editors in an IDE, Uniform Resource Locators (URLs), and resources loaded in the JVM.
Tasks 1 and 2 use `ExecContext`, which cannot be expressed or worked around.

4.4.1 Generics

In the case study, 14 tasks use generics. One of the major use cases for generics are abstract data types (ADTs). Tasks use `Option<T>` and `Result<T, E>` as return values. `Option<T>` models a value which may or may not be present, and is a type-safe version of Java's **`null`**. It is used by 1 task, `7 TigerStyle`. `Result<T, E>` is described in section 2.1.1. In short, it models exceptions as part of the return type instead of as an external interface to functions. This forces pipeline developers to consider how to handle exceptions, and means that exception handling has some support in the DSL without explicit support for Java exceptions. `Result` is used by 10 tasks: 4, 5, 6, 8, 10, 11, 12, 13, 14 and 15.

Another use case for generics is `Provider<T>`. Injected values are built into a task at task construction (see also subsection 4.4.4). A `Provider<T>` injects a value that has not been constructed yet, but will be constructed when its `get` method is called. They can be used to avoid cyclic dependencies or when constructing a value is resource-intensive but might not be required. Providers are used by 3 tasks: `1 TigerAnalyze`, `2 TigerAnalyzeMulti` and `16 TigerShowDesugaredAst`.

The final use case within the case study is for higher order functions. For example, both `Result` and `Option` have the method `map`, which takes a function as argument and executes that function on the value if that value is present.

PIE DSL 1 does not support generics. Full support for generics was implemented as part of the work for this thesis. The implementation is described in subsection 3.1.4. Since generics are fully supported, all uses in tasks can be expressed in PIE DSL 2.

4.4.2 Supplier

Suppliers are higher order tasks. Their main use case is so that a task depends on another task instead of on the output of that task. This speeds up the equality checks which are used to determine if a cached value can be re-used.

In the case study, 15 tasks use suppliers. Suppliers are generic over their output type, so they are not supported in PIE DSL 1. PIE DSL 2 adds support for suppliers, which includes syntax to create a supplier from a task. Since suppliers are supported, PIE DSL 2 can express all uses of suppliers.

4.4.3 Subclassing

PIE tasks are defined in Java by implementing the `TaskDef` interface. 'Subclassing' refers to tasks which implement `TaskDef` transitively instead of implementing it directly. In other words, they do not implement `TaskDef` themselves but they extend a class which implements `TaskDef`. This is used in Spoofax by parse (`6 TigerParse`) and analysis (`1 TigerAnalyze` and `2 TigerAnalyzeMulti`) tasks, which have large parts in common between languages but have some elements specific to the language itself. To avoid code duplication, these tasks are implemented as an abstract class with all the common parts defined, and an abstract method for the part that changes. The language specification then extends the class to get a PIE `TaskDef`.

Since the PIE DSL currently always generates the **`implements`** `TaskDef`, such subclassing cannot be expressed in either version of the DSL. However, the subclassing was done to avoid code duplication between tasks. The tasks can be expressed in both versions of the DSL by duplicating the common parts so that everything is expressed in a single task.

Supporting the subclassing as it is done in Java seems like it would unnecessarily complicate the PIE DSL. It would add inheritance to tasks, which is only useful for this specific use case. In principle, inheritance can be modeled as composition with higher order functions. A design that is more in line with the DSL requires defining regular functions (not PIE tasks) in the DSL, and higher order functions. That would allow defining a function with

the common parts, and takes the abstract method with the changing part as a parameter. A task could then be defined as a call to that function. While this requires two new features (defining regular functions and higher order functions), these two features have use cases besides extending common tasks, so they seem like a better fit for the DSL. For examples of other use cases of higher order functions, see subsection 4.4.7.

It is also possible to implement the common function in Java and then declaring it as a foreign Java function. In that case the code de-duplication would be achieved with only higher order functions in the DSL. New parse and analysis tasks could then be expressed in a few lines in the DSL instead of the at least 40 lines each subclass in Java uses.

4.4.4 Injected values

Injected values are dependencies of a task which are built into the task itself.[11] For example, the Spoofax parse tasks depend on the parse table for that language. Instead of passing the parse table as an argument to the task, the parse table is provided to the JSGLR parser, which is an internal field of the task.

In fact, one very common example of such a built-in dependency are other tasks which are called. This specific case of a built-in dependency was already supported by PIE DSL 1. Other injected values are only supported since PIE DSL 2. This is further explained in subsection 3.1.5.

Table 4.12 shows there is one task, `3 TigerCompleteTaskDef`, that does not use any kind of injected value. There is also another task, `9 TigerCheckAggregator`, which only uses an injected task, which is supported in both versions of the DSL. Finally, the remaining 17 of the 19 tasks use injected values. Since there was no workaround before this was supported, this language feature was one of the three major ones to include in PIE DSL 2.

4.4.5 Resource dependencies

The PIE DSL has first class support for file system paths. However, projects may use different kinds of resources as well. For example, it could use a file from a zipped folder that has been extracted in memory, an open editor in an IDE, or a class, file or other resource loaded in the JVM. The PIE API and runtime support these, but the DSL does not have specific support for them. All 12 out of 19 tasks that use resource dependencies use these different kinds of resources.

A workaround is to declare all of these resources as foreign Java data types, and use them like a regular Java API. That generally works, but loses out on the conciseness of the expressions and operations for built-in paths. It also fails when other unsupported language features are used as part of the resource dependency, which happens surprisingly often.

There are 2 tasks, `9 TigerCheckAggragator` and `10 TigerCheckMulti`, which walk a directory and recursively set dependencies for the subdirectories. This is not directly supported in the DSL: the `walk` expression only operates on files, so it cannot set dependencies on directories. It also does not set these dependencies automatically. And it cannot be expressed with the API directly because it uses higher order functions, which are not supported (see *Higher order functions*).

Tasks 9, 10 and 11 use a feature which is not supported in the DSL and for which using the Java API directly does not work: they depend on a directory. In Java, this is implemented as a call to `ExecContext#require`. The `ExecContext` is a parameter that is provided by the PIE runtime to every call to a task. However, the `ExecContext` is hidden in the PIE DSL, which means that the `require` method cannot be called. The PIE DSL has the built-in operator `requires`, which marks a file or directory as required. However, that operates on a built-in `path`, but we

[11] In practice, they are not hardcoded dependencies but provided to the task at its construction. This pattern is called *dependency injection*.

have a `HierarchicalResource`. There is no way in the DSL to convert a `HierarchicalResource` to an `FSPath`, so we cannot use `requires`.

Finally, tasks 11, 12 and 13 all write to a file. This is not natively supported in the DSL, as `read` does not have a counterpart `write`. The Java code uses `WritableResource#writeBytes`, which takes a byte array. Arrays are not supported in the DSL, so this method cannot be expressed. However, there is a `writeString` method, which can be expressed. Unfortunately, both `writeBytes` and `writeString` throw an `IOException` if writing fails. This cannot be expressed in the DSL (see *Exception handling*). Additionally, similarly to requiring a dependency, marking the file that is written as generated by the task uses `ExecContext#provides`, which also cannot be expressed.

Overall, 12 of the 19 tasks use resource dependencies, and 5 use them in a way that cannot be expressed. None of the tasks that use resource dependencies can use the built-in `path` type, which indicates that the current design of this feature requires some serious overhaul. Due to time constraints, this remains future work.

4.4.6 Nested class

There are two types of nested classes: static nested classes and non-static nested classes, also known as inner classes. Static nested classes and inner classes of non-generic outer classes can be referred to with simple dot syntax: `fully.qualified.path.to.Outer.Nested`. Referring to an inner class of a generic outer class also uses the dot syntax, but requires the generic arguments for the outer class: `fully.qualified.path.to.Outer<Number>.Inner`.

The PIE DSL does not have specific support for nested classes. The syntax without generics can be expressed using foreign Java declarations, because a nested class has the same syntax as a regular class in a package. Referring to an inner class of a generic outer class is impossible.

In the case study, 5 tasks use nested classes, all of which are static. The task `8 TigerCheck` uses `JsglrParseTaskInput.Builder`, which is a class to build an `JsglrParseTaskInput`. All other nested classes are data classes. Because Java higher order functions cannot have a variable amount of parameters, PIE `TaskDefs` formally all have one parameter. Tasks that need more parameters use a static data class to hold these parameters instead.

There are also 5 other tasks (the TigerShow tasks, i.e. task 15-19) that use a data class, `TigerShowArgs`. Because all these tasks use the same parameters, this data class is an independent class shared by all tasks instead of a nested class. When expressing these tasks in the PIE DSL, they will be expressed as regular parameters, and as such the compiler would generate a separate nested data class for each task, instead of a single independent class as in Java.

In conclusion, the PIE DSL does not have specific support for nested classes. However, it can still express static nested classes and some inner classes, and inner classes are rarely used. As such, 5 out of 19 tasks use nested classes in the case study, and a further 5 would use them but it got refactored out while removing code duplication. All uses and would be uses are supported in both versions of the DSL.

4.4.7 Higher order functions

Higher order functions are not supported by the DSL. Suppliers are an exception, as they are built into the language. Since suppliers have their own category, they are not included in this language feature.

14 out of 19 tasks use higher order functions. The tasks `9 TigerCheckAggregator` and `10 TigerCheckMulti` use `HierarchicalResource#walkForEach`. This is unsupported and has no workaround. This particular use of higher order function is described in more detail in *Resource dependencies*.

The task `8 TigerStyle` uses the `map` method on `Option`, and the remaining 11 tasks use similar methods on `Result`. These methods all provide concise syntax to operate on the inner value of a `Option` or `Result`. The alternative would be to check if it has a value, unwrap, operate on it, and then create a new `Result` or `Option`. While this is more verbose, it is supported in PIE DSL 2 with the introduction of generics, so these uses of higher order functions can be worked around.

One method operating on the inner values of `Results` is `Result#mapCatching`, used by 7 of the tasks. It is the same as `Result#map`, except that it takes a `ThrowingFunction` as argument. If the function throws an exception, it will be caught and wrapped into a `Result`. In the case study, `12 TigerCompileFile` and `13 TigerCompileFileAlt` use it to catch errors when writing the compiled file. These exceptions cannot be avoided without introducing a function in Java for writing files that returns a `Result`. We might implement this in the future, as writing files definitely falls under the domain of the DSL. Nevertheless, it is currently not supported without writing a helper method.

The other 5 tasks (tasks 4, 5, 16, 18, 19) use `mapCatching` when invoking Stratego strategies, which throw an exception when they fail. While catching the exception is still not supported, we can look at the implementation of `StrategoRuntime#invoke` in Figure 4.13a. We see that `invoke` is simply a wrapper around `StrategoRuntime#invokeOrNull` that throws an exception when `invokeOrNull` returns **`null`**. While we could call `invokeOrNull` to avoid the exception handling, this would no longer report the Stratego stacktrace in the `CommandFeedback` if the strategy fails. There is no way to construct the exception ourselves within the DSL, as the exception constructor takes a `String[]`. Arrays are not supported in the DSL, so this exception cannot be constructed (see *Arrays*). Additionally, using `invokeOrNull` would allow catching strategy failure, but would not catch other exceptions that could be thrown, such as an error or explicitly calling the exit function. What we can do instead of using `invokeOrNull` is create a helper method `invokeStrategoStrategy` that returns a `Result` with the exception instead. Since all the functions used in `invoke` are public, we can reimplement that method but wrap the exception into a `Result` instead of throwing it. The implementation for that is shown in Figure 4.13c. While this is an independent helper method for now, it could easily be a method that is integrated into `StrategoRuntime` or provided as a PIE DSL library for Spoofax or Stratego.

Overall, 14 of the 19 tasks use higher order functions. 5 of those 14 can be worked around so that they are fully expressible in PIE DSL 2. 7 of the tasks require a helper method in Java to be expressible in the DSL. Both of there helper methods would be fine as library methods in the future. The final 2 cannot be expressed and have no workaround. They either require specific features from the `walk` construct built into the DSL or support for higher order functions to call the Java API directly.

4.4.8 Exception handling

The PIE DSL does not have specific support for exceptions. Instead, exceptions are modeled as special return values with `Result`, and have to be explicitly handled or passed back up the call stack. While this works in isolation, many Java APIs use exceptions for control flow. For example, `java.nio.file` throws exceptions when files do not exist, cannot be written to or cannot be opened (tasks 11, 12 and 13). The `StrategoRuntime` by default throws exceptions to signify that a strategy failed or abruptly exited (e.g. because a `with` block failed) (tasks 4, 5, 16, 18, 19). And the `JSGLRParser` throws an exception when a program fails to parse (task 6).

As can be seen in Table 4.12, every task uses some kind of error handling. However, 9 of the 19 tasks (3, 7, 8, 9, 10, 11, 14, 15 and 17) only use exception handling to rethrow exceptions from their bodies. This is supported by both PIE DSL 1 and 2, since it generates **`throws`** `Exception` for every task body.

```
public IStrategoTerm invoke(String strategy, IStrategoTerm input) throws StrategoException
    {
  @Nullable final IStrategoTerm result = invokeOrNull(strategy, input);
  if (result == null)
      throw StrategoException.strategyFail(strategy, input, hybridInterpreter.
          getCompiledContext().getTrace());
  return result;
}
```

(a) `invoke` throws an exception on strategy failure. The exception includes the stratego stacktrace.

```
public @Nullable IStrategoTerm invokeOrNull(String strategy, IStrategoTerm input) throws
    StrategoException {
  hybridInterpreter.setCurrent(input);
  hybridInterpreter.setIOAgent(ioAgent);
  hybridInterpreter.getContext().setContextObject(contextObject);
  hybridInterpreter.getCompiledContext().setContextObject(contextObject);

  try {
    final boolean success = hybridInterpreter.invoke(strategy);
    return success ? hybridInterpreter.current() : null;
  } catch(InterpreterException e) {
    throw StrategoException.fromInterpreterException(strategy, input, hybridInterpreter.
        getCompiledContext().getTrace(), e);
  }
}
```

(b) `invokeOrNull` returns null if the strategy fails. This loses the Stratego stacktrace. This method has been simplified, the actual method also takes arguments for the stratego strategy and handles passing those to the `HybridInterpreter`.

```
public static Result<IStrategoTerm, StrategoException> invokeStrategoToResult(
    StrategoRuntime runtime, String strategy, IStrategoTerm input) {
  try {
    @Nullable final IStrategoTerm result = runtime.invokeOrNull(strategy, input);
    if (result == null)
      return Result.ofErr(StrategoException.strategyFail(strategy, input, runtime.
          getHybridInterpreter().getCompiledContext().getTrace()));
    return Result.ofOk(result);
  } catch(StrategoException ex) {
    return Result.ofErr(ex);
  }
}
```

(c) `invokeStrategoToResult` returns a result, which will be a `StrategyFail` exception if the strategy fails. This keeps the Stratego stacktrace. This method is static because it is not implemented in the `StrategoRuntime` class, but is a standalone helper method. It therefore also takes a `StrategoRuntime` as argument.

Figure 4.13: Three methods for calling a Stratego Strategy.

7 tasks (4, 5, 12, 13, 16, 18 and 19) use `Result#mapCatching`. 5 of those (4, 5, 16, 18 and 19) use `StrategoRuntime#invoke` to invoke a Stratego strategy. As was explained in *Higher order functions*, this method throws an exception when the strategy fails. However, we can implement a helper method that handles the exceptions and wraps them into a `Result`.

That leaves 2 other tasks, `12 TigerCompileFile` and `13 TigerCompileFileAlt`, which use `mapCatching` to catch `IOException`s from writing files. We can unwrap the `Result` to avoid the higher order functions from `mapCatching`. That leaves us with the need to catch the `IOException`s. Since `11 TigerCompileDirectory` also writes a file and simply rethrows the exception, we will assume that that is fine for `12 TigerCompileFile` and `13 TigerCompileFileAlt` as well. This means that failing to write the file will now throw an exception instead of being returned as a `Result`. Failing to write the file is not expected, even in the setting of an IDE, so throwing this as an exception is fine.

Next, the super classes of the two analysis tasks, `ConstraintAnalyzeTaskDef` for `1 TigerAnalyze` and `ConstraintAnalyzeMultiTaskDef` for `2 TigerAnalyzeMulti`, catch a `ConstraintAnalyzerException`. `2 TigerAnalyzeMulti` catches the exception directly in a catch block. `1 TigerAnalyze` uses `Result#mapCatchingOrRethrow`, which functions like `mapCatching`, except that it only catches a specific exception instead of all exceptions. Catching the exception cannot be done by a helper method because the code that can throw the exception requires access to the `ExecContext`. Since this is not available in the DSL, it cannot be passed to the helper method, so a helper method would not work. Another option is to just not catch these exceptions. While that loses functionality, these exceptions should only be thrown in case of bugs, so it might be deemed acceptable to rethrow them. If this is deemed unacceptable, the tasks cannot be expressed in the DSL.

Finally, the task `6 TigerParse` catches an exception from the `JSGLRParser`. This cannot be expressed in the PIE DSL. Furthermore, the `JSGLRParser` throws the exception to indicate that the file could not be parsed, which in the setting of an IDE is very much an expected situation: the parser runs on every keystroke[12], so the program is unlikely to always be syntactically valid. It is therefore also not acceptable to just rethrow the exception as we did with the `TigerCompileFile*` tasks above. Since this task is just a wrapper around a call to the parser that catches exceptions and returns them as a `Result`, it also does not make much sense to do this in a helper method. In conclusion, this task cannot be expressed in the PIE DSL.

Overall, all 19 tasks use exception handling. 9 only (re-)throw exceptions, which is already supported by PIE DSL 1. The rest of the tasks catch exceptions in the Java implementation of the case study, which is not supported by the DSL. 5 need a helper method, but they can share the helper method between them. 4 can omit catching an exception with some loss of functionality, but since the exception does not signify an expected condition it could be deemed acceptable. Finally, the task `6 TigerParse` must catch an exception, which cannot be expressed.

4.4.9 Field and enum access

Fields and enums access are not directly supported by the DSL. However, with the support for generics in PIE DSL 2, it is now possible to access fields and enums using Java reflection. subsection 3.1.3 describes how this can be done.

There are three kinds of field and enum access: instance field access, static field access, and enum value references.

The instance field access is used to access the fields of data classes. 6 tasks use this: `7 TigerStyle` uses `tokens` from `JSGLRTokens`, `9 TigerCheckAggregator` and `10 TigerCheckMulti` use `messages` from `JsglrParseOutput`, and tasks 8, 9, 10, 15 and 19 use several fields from `ConstraintAnalyzeTaskDef.Output`.

[12] More or less. There is a delay of a few hundred milliseconds to avoid starting and restarting the parser unnecessarily during continuous typing.

```
private ConstraintAnalyzerContext getConstraintAnalyzerContext(ExecContext context,
    ResourceKey resource) {
  final @Nullable Serializable obj = context.getInternalObject();
  if(obj instanceof ConstraintAnalyzerContext) {
      return (ConstraintAnalyzerContext)obj;
  }
  return new ConstraintAnalyzerContext(false, resource);
}
```

Figure 4.14: The function used by `ConstraintAnalyzeTaskDef` to either get an existing `ConstraintAnalyzerContext` or to construct a new one.

The static field access is used to access constants. 5 tasks use this: `4 TigerListDefNames` and `5 TigerListLiteralVals` use `Integer.MAX_VALUE` for the maximum width of the printed ATerm. `11 TigerCompileDirectory`, `12 TigerCompileFile` and `13 TigerCompileFileAlt` use `StandardCharsets.UTF_8` as the encoding when writing a file.

Finally, enum access is used to access enum values. Enums come with their own accessor method, so they can be accessed using a workaround instead of a helper method. 4 tasks use enum access: `ConstraintAnalyzeMultiTaskDef` (the superclass of `2 TigerAnalyzeMulti`), `8 TigerCheck`, `10 TigerCheckMulti` and `11 TigerCompileDirectory` use `Severity.Error` as the severity of messages.

Overall, 11 tasks access fields or enum values. It is not directly supported by the DSL, but can be worked around using generics and Java reflection. For now these helper functions have to be defined per project, but these functions can be declared in a PIE DSL standard library in the future.

4.4.10 Instanceof

The **`instanceof`** keyword in Java checks at runtime whether a value is a subclass of a specific class. This is used in the case study for one specific purpose. The PIE API allows setting an 'internal object' for the current task. This object is cached just like input and output values, but is internal to the task itself. It can be used to cache some internal state of the task. In the case study, `1 TigerAnalyze` and `2 TigerAnalyzeMulti`[13] use it to cache an analysis state, the `ConstraintAnalyzerContext`. This can then be used in the next call to the task to analyze incrementally instead of re-executing the full analysis from scratch.

Figure 4.14 shows the method used to either get the `ConstraintAnalyzerContext` from a previous call to the task, or to create a new one. The previous internal object could have come from an earlier version of this task, in which case it might be non-null but still not a `ConstraintAnalyzerContext`, so it uses `obj instanceof ConstraintAnalyzerContext` instead of `obj == null`.

While the **`instanceof`** keyword is not supported in the PIE DSL, since PIE DSL 2 it is possible to express the `Class#isInstance` method, which works exactly the same but is a generic method instead of a built-in keyword. Overall, two tasks out of 19 tasks use the **`instanceof`** keyword, and they can use the `isInstance` method as a workaround.

4.4.11 Arrays

Arrays in Java are Objects, but they have special syntax and semantics. Since the PIE DSL does not have specific support for them, there is no way to generate this syntax, so they cannot be expressed in the DSL.

[13] technically it is the two superclasses they extend, `ConstraintAnalyzeTaskDef` and `ConstraintAnalyzeTaskDef`. See *Subclassing* for an explanation of this pattern.

The 3 `TigerCompile*` tasks, 11, 12 and 13, use arrays. They write a string to a file using `HierarchicalResource#writeBytes`. This method takes a an array of bytes (`byte[]`), which cannot be expressed in the DSL. However, these bytes are obtained from a String using `getBytes`. Instead of first getting the bytes and then writing them manually, we can also use `HierarchicalResource#writeString`. This avoids the need to express the byte array in the DSL. Note that while this solves the byte array problem, both `writeBytes` and `writeString` throw an `IOException`, which is caught by `mapCatching`. Since both methods throw this exception, the workaround works around this feature without regressing in another feature, so we mark this as solvable with a workaround. See *Exception handling* for an explanation of how this exception is handled.

There are also 5 other tasks, 4, 5, 16, 18 and 19. They all call `StrategoRuntime#invoke`. This method throws an exception. Since the PIE DSL does not support catching exceptions, a workaround would be to instead call `StrategoRuntime#invokeOrNull`. However, this method loses the Stratego stack trace when the strategy fails. Recovering this stack trace requires using `strategoRuntime.getHybridInterpreter().getCompiledContext().getTrace()`, but `getTrace` returns `String[]`. Since this cannot be expressed, this workaround fails, and invoking a Stratego strategy instead requires a helper method. See also *Exception handling*. As is noted there, this workaround would not catch every exception, so a helper method might have been required anyway.

Overall, there are 3 tasks that directly use this feature, and 5 tasks that could have used this feature as a workaround for another problem. The 3 tasks that use the feature actually have an alternative function that avoids the need for arrays. The 5 tasks that could have used it for a workaround also might have required a helper method anyway.

4.4.12 Partially qualified task names

Finally, the last language feature is actually one that is missing in Java, but implemented in the PIE DSL. Every task name starts with 'Tiger'. This is to avoid name collisions in Java imports when there are other language projects which also define tasks such as `parse` and `analyze`. While that would not make tasks inexpressible, it would require fully qualifying at least one of the tasks, and to write clear code, both need to be qualified. Since Java does not have partial qualification (i.e. `tiger.Parse`), both tasks need to be fully qualified (i.e. `mb.tiger.spoofax.task.reusable.Parse`), which adds boilerplate. By adding the project name to the task as a form of ad-hoc partial qualification, the need to use fully qualified references is avoided. However, this means that this ad-hoc partial qualification is always included, even when only one language project is used and no qualification is required. This is visible in the case study: Tiger is the only language project, but it still has the partial qualification baked into the task names.

In PIE DSL 1, the issue of name collisions has to be solved at the foreign declaration site by giving the task a different name then its name in Java. For example, if this is the only parse task, it can be named `parse`: `func parse(file: path)-> Term = foreign mb.tiger.spoofax.task.reusable.TigerParse`. If there are multiple, it can be named `parseTiger`: `func parseTiger(file: path)-> Term = foreign mb.tiger.spoofax.task.reusable.TigerParse`. Since PIE DSL 1 is single file, every file has to redeclare every task it uses. That is a lot of boilerplate, but in this case it means one can decide whether to qualify the task names for each task.

In PIE DSL 2, this is no longer an issue thanks to the module system. If there is only one parse task, the name can be imported unqualified as `parse` with `import mb:tiger:spoofax:task:reusable:parse`. If there are multiple parse tasks, they can be qualified as `tiger:parse` using `import mb:tiger:spoofax:task:reusable:parse as tiger:parse` and `cpp:parse` from `import org:example:lang:cpp:parse as cpp:parse`. In this case, both fully qualified names still have to be mentioned again at the import. If both projects have the same package struc-

ture, both tasks can be imported at the same time to avoid the common package parts: `import mb:{tiger, cpp}:spoofax:task:reusable:parse` will make both `tiger:parse` and `cpp:parse` available. If we want to access multiple tasks from a single language, one can import multiple tasks at the same time to avoid fully qualifying each of them: `import mb:tiger:spoofax:task:{reusable:{parse, style}, check, compileFile}`. Java can do something similar with a star import: `import mb.tiger.spoofax.task.reusable.*; import mb.tiger.spoofax.task.*`. It uses less code (particularly when more tasks are imported), but it needs a separate statement for each sub-package. It also does not support importing just a few classes from the package, it imports every class in the package. To import some but not all classes, every class needs to be imported separately. Finally, if there are two languages with multiple tasks to be imported, one can use a partial import instead of importing tasks separately: `import mb:tiger:spoofax:{task as tiger, task:reusable as tiger:reusable}`.

Overall, name collisions are a problem, and partially qualified names are the ideal solution. Because name collisions are a global problem, every task is affected. Java lacks partial qualification and needs to use a workaround by either using fully qualified names or by partially qualifying the name at the declaring project. This workaround is in some ways worse than workarounds for most other missing features, because it affects code even when the problem is not present in the current project. The decision to use the workaround has to be made at the declaration site, so using it will affect cases where it would not be required, but not using it means it is unavailable when it is required. The case study does not have name collisions, but because the decision has to be made at the declaration site, the workaround is still used in Java. Partially qualified names are also not available in PIE DSL 1, but it has a workaround that can be decided for each project, and because it is single-file the boilerplate for that workaround has to be incurred anyway. Partially qualified names do exist in PIE DSL 2. Additionally, it has some other features to help express imports efficiently.

4.4.13 Conclusion

We can analyze what language features each task uses from two perspectives: the tasks or the language features. The following paragraphs both summarize the findings of this section for their perspective. They also answer the question for this section: *how does the PIE DSL 2 compare to PIE DSL 1?*

Tasks Overall, none of the tasks could be expressed in the PIE DSL 1. In the case study we only had suppliers, only a single extra task could be expressed: `3 TigerCompleteTaskDef`. Now that we have injected values, suppliers and generics, we should in theory be able to express many more tasks: 3 tasks without helper methods, 8 tasks with helper methods and 8 tasks which can still not be expressed in the DSL.

While 12 tasks require helper methods, they actually only use 4 helper methods. Two methods allow accessing static fields and instance fields. The two other methods, one to write files and the other `invokeStrategoToResult`, catch exceptions that are thrown by the functions they wrap.

With these 4 helper methods, 11 of the 19 tasks can be expressed, and the remaining 8 not yet. This is much better than the 0 that can be expressed in PIE DSL 1 and the 1 task that could be expressed in the case study. However, while the analysis in this section strongly suggests that these tasks can now be expressed, this has not been tested.

Even though we had hoped that more tasks could be expressed in PIE DSL 2, going from 0 expressible tasks to 11 expressible tasks means PIE DSL 2 is a marked improvement over PIE DSL 1.

Language features There are 4 new major language features that have full support: generics, suppliers, injected values and the module system. The first 3 enable enable 15 tasks to

be expressed. The module system avoids name collisions but does not enable more tasks.

Static nested classes are already expressible in PIE DSL 1. Inner classes are still inexpressible but are unused in the case study, and we expect them to be very rare in other use cases.

Subclassing and `instanceof` are not supported by the DSL, but they have workarounds. They will not be supported as they should not be used in the DSL and are discouraged in Java.

Resource dependencies currently prevent expressing the most tasks. They are part of the core domain of PIE and have no workaround (not even a helper method), so updating them has high priority.

Finally, higher order functions, exception handling and field and enum access are all commonly used and lack some support. Exception handling will almost certainly get some support so that throwing functions can be expressed and handled in the DSL. Higher order functions and field and enum access are common, but they are not used *that* often and will add non-trivial bloat to the language design, so we will look into whether the tradeoffs are worth it.

Overall, support for three more language features was implemented in PIE DSL 2: suppliers, generics and injected values. While this is not too many, these three did also enable workarounds and helper methods for 5 other features: higher order functions, exception handling, field and enum access, `instanceof` and arrays. This means that PIE DSL 2 is again a marked improvement compared to PIE DSL 1.

Chapter 5

Related work

This chapter looks at related work. PIE is a relatively new framework, so there is not much literature on it. It does mean that we can go over each work in detail.

PIE: A Domain-Specific Language for Interactive Software Development Pipelines The PIE framework, runtime and DSL are introduced in Konat, Steindorfer, et al. (2018). It uses the PIE DSL in two case studies. The first one was also for Spoofax, but it implements generic parsing tasks and tasks for keeping editors up to date, instead of custom tasks for a specific language project as we do for Tiger. All of these tasks are now generated by a Spoofax Gradle plugin.

The second case study is on live performance testing of a set of Java data structures with the benchmarking suite Criterion. The original implementation run every combination of options every time, which takes about 2 days. The re-implementation with PIE makes use of the built-in incrementality of PIE to only run benchmarks if the subject or the benchmark changes.

It concludes that the PIE framework and DSL result in a 6x code size reduction in the first case study. This seems to be in stark opposition to our conclusion that the PIE DSL does not reduce code size. However, this difference is likely caused by this paper comparing its DSL code to code written without the PIE framework, while in our case study we use code that was already written with the PIE framework as our baseline.

Scalable incremental building with dynamic task dependencies In Konat, Erdweg, and Visser (2018), the PIE runtime and API is extended with the option to execute builds bottom-up. A bottom-up build takes a set of changed files and directories. The tasks that depend on these changes are then re-executed, and it continues re-executing tasks until all tasks are up-to-date.

While this paper is about PIE and the change is useful for PIE, it did not affect this thesis as we did not execute the code as part of our evaluation, we only analyzed it statically. The goals, methods and results of this paper are completely orthogonal to those of this thesis, therefore they cannot be meaningfully compared.

Task Observability in change driven incremental build systems with dynamic dependencies The master thesis Sol (2019) introduces the concept of *observability* to PIE. A user can explicitly mark some tasks as observable, and the dependencies of observable tasks are then also recursively marked as observable. Tasks that are not explicitly or implicitly observed can be removed, which solves the issue that tasks which are no longer required for a build stay in the dependency graph.

Again, while Sol's thesis improves PIE, it is orthogonal to this thesis.

Precise, Efficient, and Expressive Incremental Build Scripts with PIE Konat, Sol, et al. (2019) briefly summarizes the findings of the previous two papers and thesis.

Chapter 6

Future work

This section lists future improvements to the PIE DSL and its ecosystem. The main goal of the DSL is to allow its users to express pipelines. This goal can be divided into three categories: extend the expressive power of the DSL, improve the User Experience (UX) of the DSL, and improve the code base of the DSL. We also discuss our roadmap in the last section of this chapter.

6.1 Increase the expressive power of the DSL

One of the main limitations we found in our case study is interoperating with Java. In particular, many tasks and functions could not be expressed in the PIE DSL, not even as a foreign Java declaration. Fully expressing these tasks in the PIE DSL should provide some benefit in the future after the DSL has been improved some more. The other big issue was foreign Java declarations, which are error prone and add a lot of boilerplate. Generating these automatically should help the DSL reduce boilerplate and avoid a lot of mistakes.

6.1.1 Generics

While generics are implemented for the DSL, there are a few possible points of improvement. First of all, as also mentioned in subsection 3.1.4, there are bugs, in particular with wildcards. While they are likely not fundamental design issues and not show-stopping for using the DSL, it would still be good to resolve them. Secondly, we learned very late in the implementation of generics that Java uses 'use-site variance', while other languages use 'declaration-site variance'. Declaration-site variance seems a lot simpler than use-site variance On the other hand, use-site variance has greater expressive power. Additionally, Java uses use-site variance, so compiling declaration-site variance to Java will be non-trivial. Moreover, adding extra features makes the language more bloated and harder to learn, even if the feature itself is a simpler alternative to an existing feature. Nevertheless, it seems a good idea to at least investigate adding declaration-site variance to the PIE DSL.

6.1.2 Suppliers

Suppliers are built-in in the PIE DSL. This is because they were added before generics were added. Now that generics exist as well, suppliers can be almost entirely expressed as a regular foreign data type. It would be nice if suppliers could be removed as built-in feature, as that would make the language simpler. There are two parts that still require language support.

The first part is getting a value from a supplier. This requires access to the `ExecContect`, which is not available in PIE DSL bodies. There are two options to get a value when suppliers are not built-in. The first one is to make `ExecContect` available somehow. This could be as

easy as defining a keyword `context` that evaluates to the `ExecContect`. Another option is to make `get` a built-in function, method on suppliers, or even to define special syntax to get the value from a supplier. Both of these options require that we figure out how to refer to a type which is not built into the DSL. That is also required for some methods on built-in types and to make certain language constructs customizable (e.g. to allow a custom `Matcher` for `list` or `walk` expressions). However, right now it is not clear what the best design is: referring to a specific PIE DSL datatype seems unsatisfactory, and so does referring to a specific Java class.

The second part that still requires some language support is getting a supplier from a task or function. Right now, this has special syntax. There are again two options to do this. The first option is to define that task suppliers return a `Supplier[T]`, where `Supplier[T]` is not a built-in type. The special syntax remains, but the type itself is no longer built-in. This again requires referring to a type that is not built into the DSL. The alternative is to implement higher order functions. Higher order functions have other use cases (see also *Higher order functions*), so this is just one more reason to implement them. If they were implemented, task suppliers could be implemented without special syntax for suppliers, so then suppliers could be fully expressed with existing language features.

6.1.3 Subclassing

Subclassing can be expressed by making normal task declarations where the common parts from the abstract superclass are duplicated. This workaround should work in general and is not too bad. Moreover, extending the DSL with some kind of subclassing seems like it would go against its sort-of functional style. So we do not want to add support for this particular pattern. However, the idea of factoring out the common parts of tasks into reusable elements is good. Instead of expressing this with subclassing and inheritance, it can also be expressed by creating a freestanding function and calling that function from the task. This fits better with the functional style. To support this, the PIE DSL would need to support defining functions that are not tasks. While this is certainly doable, there are still a few open design questions, such as where these functions should be generated, and what syntax would be used to signify that this is a function instead of a task.

6.1.4 Injected values

Injected values have not been tested in an actual use case, but they should work. Either way, there is a feature that is currently not supported by the DSL: injecting qualified values. Normally, the injection framework can work out what value should be injected based on the type of the value. However, sometimes that is not enough, and more specific information is required. In that case, a use site (i.e. the task we generate) can specify a qualified value to be injected. Methods that provide injected values can then specify that they provide that specific qualified instance. For example, package IDs in Spoofax 3 are just normal Strings. Since a String could be anything, it specifies a qualified value by naming it: `@Provides @Named ("packageId")String providePackageId()`. This can currently not be expressed in the PIE DSL. In the case study, this is used in three locations: for the package ID shown above, for a prototype `StrategoRuntime`, which is both provided and used by the project[1], and for a `HierarchicalResource`, which is qualified to be the root directory of the project.

The current design for injected values does not support this yet. Java uses annotations, so an obvious option is to use those in the DSL as well. While annotations have been considered as language feature for the DSL, it is a huge feature, since it interacts with the syntax, static semantics and code generation for every language element. Annotations are discussed in more detail in subsection 6.1.22. Additionally, we have not looked closely into the best design for qualifying injected values, so there may be better options.

[1] This prototype is copied when a new `StrategoRuntime` is required

6.1.5 Resource dependencies

Resource dependencies have been identified as a big problem in their current design. While some use cases can be expressed by using the Java API directly, certain others cannot because they use features which are not supported. In the case study, `walk` uses higher order functions, and declaring a dependency on a `HierarchicalResource` is not supported because it requires access to the `ExecContext`, which is not available in the DSL, and the built-in expression `requires` needs an `FSPath`, not a `HierarchicalResource`. Similarly, marking a file as generated by the current task also requires access to the `ExecContext` and renders tasks inexpressible.

All of these issues are caused by the fact that, yet again, the PIE framework has been updated while the DSL has lagged behind. In this case, the DSL uses `FSPath`, while the framework has updated to mostly use `HierarchicalResource`. This can likely be solved by using `HierarchicalResource` as the backing type for the built-in type `path`.

The `walk` and `list` expressions could also use an update. They currently do not support all matchers (in particular, there is no way to specify what to do with hidden files and directories) and no support for walkers, which define what directories will be recursively walked in a `walk` expression.

We could also look into reading and writing. Reading uses Unicode Transformation Format 8 (UTF8), and there is no way to use a different encoding. Writing files is not supported, and the Java API methods throw exceptions. We could extend reading to take an optional encoding and add a `write str to file as UTF8` expression.

The final part to consider is exception handling. File operations in the Java API throw `IOExceptions`. The DSL does not support exception handling, so an alternative design might be to return a `Result`. On the other hand, adding such an explicit dependency on `Result` might have negative effects. This will have to be investigated.

Both the encoding and using `Result` for exception handling run into the issue mentioned in *Suppliers*: how to refer to non-built-in types from within the DSL?

6.1.6 Nested classes

Nested classes are not specifically supported in the DSL. This has not been a problem for now, so no features have been planned or considered. It may be an idea to give the option to use type parameters in foreign Java qualified IDs, which would allow referring to inner classes.

6.1.7 Higher order functions

Higher order functions are not supported. Supporting them would solve a few issues, such as walking over a directory and workarounds for catching exceptions. It would also make it possible to express mapping ADTs like `Result` and `Option` more concisely, and gives such mapping operations a fluent API. On the other hand, it adds yet another feature to what was supposed to be a small language. It is also not entirely clear how to implement this, as Java closures are not Serializable. It may be possible to define our own Serializable function types, but it is not entirely clear that this works.

6.1.8 Exception handling

Exception handling is not supported, and it comes up quite a lot. It also seems like a good idea to add some support to the language, because every wrapper function has the same functionality: call the function that throws, catch any exceptions, wrap the return value or the caught exception in a `Result`. This is in fact so common that `Result` already has a method that does exactly this: `ofOkOrCatching`.

There are three ways to support exception handling in the PIE DSL. The first one is to simply add exception handling by copying Java, i.e. add a catch clause and a way to declare that functions throw exceptions. While that is the most straightforward solution, we do not want exception handling in the DSL, we want to use `Result` if possible. We only to be able to interoperate with exception handling in Java.

The second option is to add special syntax to foreign functions that they throw some specific exception, and to compile calls to these functions to `Result#ofOkOrCatching`.

Finally, if we add support for higher order functions, it is possible to call `ofOkOrCatching` yourself. This could be made a bit more concise by making that function available under the name `try`, so that it could be called as `val res: Result[Output, Exception] = try(()=> someFailingOpertation(args))`. While this has the advantage of only adding higher order functions, which have other uses as well, it also means that the type system does not enforce that throwing functions are always called with `ofOkOrCatching`. Since we would really like the type system to enforce this, the second option might be the best solution.

6.1.9 Field and enum access

Field and enum access has a workaround using generics and Java reflection. However, it is still fairly verbose, Java reflection definitely comes with a performance hit when compared to accessing fields or enum values directly, it may be disallowed with Java security settings, and the names of fields and values are not checked statically, because they are passed as strings. While these are compelling arguments to add some form of support, adding fields and enums to the PIE DSL adds a language feature that may not be strictly necessary. In both cases, it might be better to only implement support in the foreign Java declaration, and to model their access with functions or methods.

Fields

Figure 6.1 shows rough drafts of the two options for fields. Figure 6.1a shows support in the foreign implementation section only. Within the DSL, fields are modeled using getters and setters, which are compiled to direct field access. Figure 6.1b shows a draft of full integration with the DSL, where there is syntax for direct field access. Note that it also goes a step further and models the getter and setter for `Box` as a field as well, even though in Java those are actual methods. Full integration is likely not necessary (and therefore better avoided to avoid additional complexity in the DSL itself) if it is only for interoperating with Java. However, we may implement declaring data classes in the DSL itself, in which case full support for fields makes a lot more sense.

Enums

For enums, every enum has a built-in function `valueOf` to get a value from a string. As is shown in Figure 3.2, calling this function is about as short as it can get. The declaration is still a bit verbose, but that only happens once per enum, and can be alleviated by generating foreign Java declarations or accessing Java declarations directly (see subsection 6.1.20). That leaves the issue of static checks: there are no checks to verify that the specified string is an actual value of the enum. This could be fixed by defining a new datatype implementation for foreign Java enums, which defines the values that can be used. This is shown in Figure 6.2. Also shown in that Figure is the option to use the same syntax as Java for enum access, or to model it with functions. The functions can be generated in their own module, which makes them easy to refer to and does not even add boilerplate when importing the type: `import mb: pie:dsl:proposal:enums:Severity` conveniently imports both the datatype and the module.

```
module mb:pie:dsl:proposal:fields:model_as_methods

import somewhere:{ProjectResult, IStrategoTerm}

data SingleFileResult =
  foreign java mb.constraint.common.ConstraintAnalyzer.SingleFileResult {
  // These fields are final, so they only have a getter
  field ProjectResult? with getter          // field name is derived from type
  field analysis: IStrategoTerm with getter // getter name is derived from field name
  // alternative syntax, bit more verbose but more in line with PIE DSL declarations
  func getParsedAst() -> IStrategoTerm = getter // field name derived from getter name
}

data Box[T] = foreign java org.example.Box {
  // Getters and setters in backing Java class are just normal methods
  func get() -> T
  func set(t: T) -> unit
}

func exampleUse(result: singleFileResult, box: Box[IStrategoTerm]) -> IStrategoTerm = {
  // Using getters and setters will compile to direct field access for public fields
  val currentTerm = box.get();          // IStrategoTerm currentTerm = box.get();
  val analysis = result.getAnalysis();  // IStrategoTerm analysis = result.analysis;
  box.set(analysis);                    // box.set(analysis);
  currentTerm                           // return currentTerm;
}
```

(a) A rough draft of what modelling fields with methods could look like.

```
module mb:pie:dsl:proposal:fields:full_integration

import somewhere:{ProjectResult, IStrategoTerm}

data SingleFileResult =
  foreign java mb.constraint.common.ConstraintAnalyzer.SingleFileResult {
  // All fields are read-only by default
  field ProjectResult?           // field name is derived from type
  field analysis: IStrategoTerm
  // omitted: some unused fields
}

data Box[T] = foreign java org.example.Box {
  // cannot access value directly in Java, must use getter and setter.
  field value: T private with getter get, setter set
}

func exampleUse(result: singleFileResult, box: Box[IStrategoTerm]) -> IStrategoTerm = {
  // Direct field access compiles to getters and setters for private fields
  val currentTerm = box.value;     // IStrategoTerm currentTerm = box.get();
  val analysis = result.analysis;  // IStrategoTerm analysis = result.analysis;
  box.value = analysis;            // box.set(analysis);
  currentTerm                      // return currentTerm;
}
```

(b) A rough draft of what full integration of fields could look like.

Figure 6.1: Rough drafts of what support for fields could look like. Note that the compiled Java code is the same for both cases.

```
module mb:pie:dsl:proposal:enums

data Severity = foreign java enum Severity {
  ERROR, WARNING, NOTE, DEBUG, TRACE
}

func exampleUse() -> Severity = {
  // option 1: definition above generates functions in module
  Severity:ERROR()  // fully qualified: mb:pie:dsl:proposal:enums:Severity:ERROR()

  // option 2: full support, same syntax as Java
  Severity.ERROR
}
```

Figure 6.2: A rough draft of what support for enums in the DSL could look like.

6.1.10 Instanceof

The PIE DSL does not support the Java keyword **`instanceof`**, but a workaround exists that uses the method `Class#isInstance`. While this adds a lot of boilerplate, using **`instanceof`** is discouraged, as it is often better to use dynamic dispatch. Therefore, no additional support is planned for **`instanceof`**.

6.1.11 Arrays

Arrays cannot be expressed in the PIE DSL. In the case study, they are used in two cases. Writing files uses byte arrays to represent the encoded file contents to be written. This has a workaround that just takes the string and encoding. `StrategoRuntime#invoke` uses `String[]` internally to pass the Stratego stack trace. This prevents a workaround that avoids the exception handling of that method. However, if the DSL supported exception handling, there would have been no need to use arrays here.

One case has a workaround and the other one is not required with support for exception handling. Therefore, adding support for arrays does not have high priority.

Logically, an array is just a special type of list: an ordered, variable-length collection of heterogeneously typed elements. However, we would like to keep arrays distinct from lists so that an array is not accidentally used where a list is required. This means that arrays need to be a different type from lists. Because arrays are types, they cannot be expressed in just the implementation part of a foreign Java declaration: they have to be part of the function signature. That means they leak into the DSL itself, not just the compatibility layer of the DSL. We would like to avoid adding arrays to the DSL because as noted earlier, they are logically just lists, which already exist. Balancing these conflicting goals will be tricky, so we are lucky we do not plan on adding support.

6.1.12 Add methods to built-in types

Add methods to the PIE DSL built-in types. For example, paths could have methods to replace or remove their file extension and lists should really get methods to access individual elements. This likely runs in the aforementioned problem of referring to types which are not built into the DSL. See *Suppliers* for more information on that problem.

6.1.13 Create libraries

Now that the PIE DSL is multi-file and has a module system, reusable libraries can be defined and shared between different projects. The most important library is the PIE DSL standard

library. One of the main things it would contain is interoperability functions for the DSL with Java. Examples are the functions to access fields, identity functions to convert built-in types to their backing Java type, e.g. `path` to an `FSPath` (or `HierarchicalResource` when that is changed) and vice-versa. Another function could be an identity function with different input and output type. This is useful when you have two different data types in the PIE DSL that are backed by the same Java class and you need to convert one into the other. Other libraries could provide foreign Java declarations for Spoofax, resources, and ADTs such as `Result` and `Option`.

6.1.14 Type inference for type arguments

Right now, every type argument has to be provided explicitly, except for the type argument of the built-in `supplier` function. Inferring the type arguments from the context is very difficult in general, but should be easy for some common cases, such as a type argument that is used for a normal argument. Type inference can be added gradually because it is not required to be complete: if a type argument cannot be inferred, the pipeline developer can still provide it explicitly. As long as it is sound and does not slow down analysis too much, any type inference is a strict improvement to the language.

6.1.15 Preludes

The second source of boilerplate in the DSL is imports. In our file of ideal PIE code for the case study (see Appendix D) we use around 30 lines of imports. Instead of importing everything by hand, libraries could explicitly export their most commonly used elements so that other files can import these elements in one line. For example, `import mb:stratego :prelude` could import `mb:stratego:common:StrategoRuntime`, `invokeStrategoStrategy` (the helper method shown in Figure 4.13), `org:spoofax:interpreter:terms:IStrategoTerm` and `mb:aterm:common:termToString`. This would require support in the module system for transitive imports, but that should be possible in Statix.

6.1.16 Performance

We used to assume that the performance of the generated Java code would be an insignificant difference because the PIE code matched the reference Java code closely, and the generated code matches the PIE code closely. This is no longer the case: now that we can express generics, we use Java reflection to access fields and unwrap `Result` to modify its element instead of using a higher order function together with `map`. This likely has an impact is on the performance of the generated code. It would be excellent to set up an automated benchmark that we can run on newer versions of the DSL to prevent future regressions in performance and to quantify how much new features improve or degrade performance.

6.1.17 Void methods

Methods may operate entirely via side effects. In that case, they do not return a useful value. In the PIE DSL, this is signified with the return type `unit`. Java uses **`void`**. Unfortunately, the compiler currently assumes that every method has a return value, and generates code like this: `None result = call();`.[2] This results in an error in the generated Java code.

Adding void to the PIE DSL would bring a Java implementation detail into the DSL, so we want to avoid that. Instead, a solution would be to declare the foreign Java function as `unit`, and have the compiler generate code that does not use the return value of the method call, but just get an instance of unit itself: `call(); None result = None.instance;`. This does

[2] `None` is the backing Java class for the PIE type `unit`

```
comparable-with { _ <: List[_ <: T], _ <: List[_ :> T] }
data ArrayList[T] : List[T] = foreign java java.util.ArrayList {}
```

Figure 6.3: An example of how to use `comparable-with` to specify that `ArrayList[T]` is comparable with lists that contain subtypes or supertypes of `T`, not just `List[T]`.

mean that there is no way to distinguish between Java methods that have the return types **`void`** and `None`, but that likely never matters.

6.1.18 Precise semantics for comparisons

After implementing the new semantics for comparability, we realized that we assume that two expressions can only be equal if they yield the same value, but that assumption is incorrect. According to the Java semantics for equality, which the PIE DSL follows for interoperability, equals needs to be reflexive, symmetric, transitive, consistent, and return `false` when called with `null`. It does not require that different values are unequal. Java collections actually specify that ADTs are equal when their contents are equal, regardless of the implementation of the ADT, and regardless of the type arguments for the ADT. In general, two values can arbitrarily be equal, so it seems that to avoid rejecting perfectly valid and sensible comparisons, any two types should be comparable. On the other hand, we also would like to give warnings on comparisons which are known to be nonsensical. The original sentiment of the idea to check for intersection of types was good, but it used an invalid assumption. The core idea is still valid: a comparison is nonsensical when the result of the comparison is statically known, and such nonsensical comparisons should give warnings. Finding a semantics that finds all nonsensical comparisons without any false positives for sensible comparisons is future work.

An idea is to introduce a modifier `comparable-with`, which would declare that a data type `T_decl` can be compared to another data type. Figure 6.3 shows an example of a possible implementation of `comparable-with`. It takes a list of types `T_others`. When trying to compare a type `T` to `T_decl`, it checks if `T` matches any of `T_others`. If it does, the types are comparable. Otherwise, the normal checks with subtyping are used.

6.1.19 Dead code

In Java, certain forms of dead code are disallowed. For example, any code after a return statement or in the body of an **`if`** `(`**`false`**`){ ... }`. We do not want to use this in the PIE DSL because we do not want to make the semantics of the language depend on how well the analysis can detect dead code. While we do not want to make dead code analysis part of the semantics of the language, dead code analysis still needs to be implemented so that the compiler does not generate such dead code. If we are going to implement such analysis anyway, we might as well do it properly and issue warnings to the user that their code is dead.

6.1.20 Foreign Java declarations

As is shown in Table 4.2, the PIE DSL has significant boilerplate in the form of foreign Java declarations. There are two solutions for this. Ideally, foreign Java declarations are not required at all. The PIE DSL could just read the declarations from Java files directly. While this would be ideal, it is a lot of work and will likely take a very long time before this is available. An alternative solution is to generate foreign Java declarations from Java files. This is less user friendly than no foreign declarations at all, but still gives a lot of benefits over the

current state where users have to write foreign Java declarations manually. It is also much easier and quicker to implement than the full integration with other languages.

Generate foreign Java declarations As is shown in Table 4.2, the PIE DSL has significant boilerplate in the form of foreign Java declarations. These could be generated automatically from the Java files. Generating mostly correct declarations automatically would be fairly easy, as it can just generate based on the AST of the Java file, and have someone manually fix references and methods that do not map neatly to the DSL. However, such a setup means that there is always a human required to check and fix the generated boilerplate, and that regenerating requires reapplying the changes, or somehow merging the existing changes with the newly generated boilerplate.

In short, while creating a tool that generates mostly correct code would be easy, it would still leave a lot of manual work. It is also possible to make the tool such that it generates entirely correct foreign Java declarations. While this is a lot of upfront work, it would allow integrating it into the build of the PIE DSL code. This ultimately means that it will always be up to date, requires minimal maintenance, and since the generated boilerplate does not require any human intervention, it is generated code and does not need to be checked into the Version Control System (VCS).

Read declarations from JVM languages directly In the longer term, it would be great to integrate with the editor or the build system so that Java and other JVM declarations are available directly in the PIE DSL, without the need to write or generate foreign Java declarations. This will require significant work because it requires that either the PIE DSL analyzer gets a hold of the full Java Classpath and is able to analyze all the files on it (requires a parser and analyzer for any JVM dialect that is capable of communicating the results back to the PIE analyzer), or it requires passing the PIE DSL definitions to some other analyzer, which are unsupported by many (or even any) editors for cross-language analysis. To avoid an $N \times (N-1)$ situation where there are N different languages which need to implement front-ends for all other $N-1$ languages, a solution might be to define a common representation of declarations and references and a common protocol for publishing and searching such declarations and references. That way languages only need to implement publishing and searching the common representation of declarations and references, not searching declarations and references in every language representation. This has parallels to the Language Server Protocol (LSP), but the LSP does not have information on declarations besides the name and location. It could however be extended to keep that information, or a new, independent protocol could be implemented to do that. While arbitrary languages can have arbitrary language features, languages on the JVM will ultimately need to be compiled to bytecode. Bytecode has a known feature set, so the common representation for declarations and references could take the information in bytecode as the basics, with perhaps some optional extra information such as parametric polymorphism (generics), and information on nullability, termination, side effects and whether a function is deterministic.

6.1.21 Generate functions instead of tasks

While implementing the case study, we found that for certain longer tasks it can be useful to split the task up into smaller functions. While this could be done in the DSL right now, that will make the new functions tasks as well, which will incur a small amount of performance overhead from PIE. Additionally, all the inputs and outputs for these new tasks will now be cached, which may involve nontrivial amounts of intermediate representations of data. Furthermore, from the perspective of the pipeline these functions should not really be tasks, they are just implementation details of this one task. This leads to the conclusion that there

should be a way to declare functions in the PIE DSL that are not generated as tasks, but just as separate Java methods.

6.1.22 Annotations

Annotations would make it possible to update the static semantics and code generation of the PIE DSL. For example, Dagger is an injection framework in Java which makes heavy use of annotations. Spoofax 3 uses Dagger, and requires[3] annotations for Dagger scopes on tasks. However, always generating these annotations will break the generated code for anyone not using Dagger. Moreover, this is a feature that is specific to Spoofax 3, so it should not be built directly into the PIE DSL or its compiler at all, probably not even as a configurable option. Annotations would allow users to add this code generation themselves.

Annotations could also be used to do custom static analysis, such as keeping track of whether a function is deterministic and side-effect-free or not and using that to generate more efficient code, or to see if a function or task is unused, or to define that a certain task is a Spoofax 3 *command* (see *Declare Spoofax 3 commands within the DSL* in the next section).

6.1.23 Compiler options

The compiler currently only has one option: a path to a folder to generate the files. The current implementation parses this path manually in Stratego, but that format does not scale in the slightest to more compiler options. It is possible to implement a few more options, like what package structure to use for the generated tasks (all in the same package, follow the module structure, or one of the first two but with a prefix), to abort when the static analysis has errors, and to treat warnings as errors. There are also some other proposals that would add configuration options to the compiler, such as what to do if an annotation is not configured and where to generate static Java functions.

6.2 Improve the code base of the PIE DSL

Refactoring the code base of the PIE DSL could improve maintainability and performance.

For example, the compiler tests have a lot of boilerplate and take a long time to build, which slows down development as it is a bit daunting to work on. They also are end-to-end tests, which use a stable interface, but require a full program analysis every time. It might be possible to use an analyzed program fragment as input, which would test only the compiler.

It may also make sense to implement SPT tests for the compiler that only check the compilation from a PIE AST to a Java AST.

Another example is the type system. The current implementation has only two kinds of types: syntactic types and semantic types. However, near the end of implementing generics it became clear that there are multiple kinds of semantic types: parameterized types are types with type parameters, instantiated types are types where the type parameters have been substituted with type arguments, and erased types are types where type arguments and parameters have been erased. While all of these can currently be represented by semantic types, they represent different stages of types, and types from different stages should never be mixed. Splitting semantic types into these three type kinds would allow the Statix type system to catch mistakes where types of different stages are used together. It also makes the conceptual model of types clearer to developers.

The final example is the implementation of the module system. The current implementation for modules is both complicated and cannot make good use of the incremental Statix

[3] Not strictly required for a single language project, but they are required in multi-language project setups.

solver. A refactor could keep the semantics the same but would make the code a lot simpler and hopefully also improve the performance. This final example is explained in further detail in the next paragraphs

6.2.1 Optimize the module system implementation

The current implementation of the module system first gathers all modules and turns them into a tree by manipulating constructors. It then creates a scope graph from that module tree, which results in a tree with the project scope as root. An alternative implementation is to declare every module directly in the root scope by declaring the root module and then its submodules recursively, which results in a bunch of linked lists all starting at the project scope.

The current implementation was chosen because it uses 1 scope per module, while the alternative implementation has a scope for each file times the nesting level of each module (e.g. `a:b:c:d:e` has nesting level 5). For example, given two files `a:b:c1` and `a:b:c2`, the current implementation uses 4 scopes: one each for module `a`, `b`, `c1` and `c2`. The alternative implementation uses 6 scopes: `a`, `b`, `c1`, `a` and `b` again, and finally `c2`.

However, the current implementation has a few shortcomings as well. Statix has been improved since the initial implementation, and now has some optimizations based on modularity. In particular, it can pre-analyze libraries and perform concurrent analysis. However, these features only work when there are groups of independent files, which is not the case in the current implementation: the module tree depends on all files, the scope graph depends on the module tree, and every non-trivial file depends on the scope graph, so in the end, every file depends on every other file. In addition, the current implementation is also rather complex.

6.3 Improve the user experience of the DSL

Besides the inability to express certain tasks in the PIE DSL, the process of expressing the tasks that can be expressed is still a little painful at times. In particular, many useful editor features like suggestions, quick fixes and automatically adding imports are missing from the DSL. Additionally, the DSL could be integrated with its ecosystem to provide value beyond Java, such as automatically adding tasks to the PIE runtime, or declaring Spoofax 3 commands more succinctly and automatically adding them to Spoofax 3.

6.3.1 More editor features

One of the areas where the PIE DSL is clearly inferior to Java is its editor features. The defacto editor is the Spoofax 2 Eclipse IDE. All other editors just treat PIE files as generic text files. By default, the Spoofax 2 Eclipse IDE provides syntax highlighting, error markers, type information when hovering, reference following and a few features to show intermediate representations of DSL code, which mostly only help with debugging the specification of the DSL itself.

Missing and incomplete features Many editor features are still missing or incomplete. First of all, hovering over an expression, type or function shows type information, but at the moment that information shows the ID of the scope that implements a data type. This ID is almost completely useless to any user of the DSL, what they need is the name of the data type, possibly with its type arguments.

Spoofax also has support for outlines, but they are not generated by default. Outlines show an overview of the program, which can enable faster navigation and a different way of exploring an existing program.

Furthermore, the editor could check that foreign Java declarations are correct. This requires that the editor understands what Java definition a foreign Java declaration references, but would catch mistakes quicker.

Moreover, the editor could make suggestions while typing, such as suggestions for function names. For example, if the user types `file.ext` in a position where an expression is expected, the editor could suggest the method call `file.getLeafExtension()`. An improvement which we have not seen before but which has been suggested in Will (2016) is to also search independent functions when the syntax for a method is used. The previous example could also suggest `extractData(file)` if there was a function `extractData(file: path)-> _` which uses `file` as a parameter, even though this is a function call, not a method call.

Additionally, the editor could suggest quick fixes. These are like suggestions, but they show up on existing code, instead of as the user is typing. As the name suggests, they are suggestions to fix something, which means they often show up on errors or warnings. Examples are adding an import for an element which is defined in the PIE project but not imported in the module where it is used ("undefined function 'discombobulate'. Quick fix: import com:acme:stdLib:discombobulate"), adding a foreign Java declaration for an element which has not been defined in the PIE project but which does have a declaration in Java ("undefined function 'discombobulate'. Quick fix: add foreign Java declaration for com.acme.StdLib#discombobulate"), changing the declared types of variables, parameters and functions to match the actual type of the expression in case of a type mismatch ("Type mismatch: expected an int, but got a string. Quick fix: change type of 'x' to string"), and removing unused values and imports.

Next, the editor could have more refactorings. Refactorings improve the code without changing its function. There are several refactorings that could be implemented for the PIE DSL. Renaming elements such as modules, data types, functions, parameters and values allows giving better names after writing the code and better understanding the problem. Moving a function or datatype can be done by copy-pasting the code, but a refactoring can automatically update imports and references.

Finally, the editor could support meta-documentation, that is, documentation about the PIE DSL code itself, which is used by the developer of the code for maintenance and extension. Many editors can show documentation of a reference to a function or type in a palette window, without the need to follow the reference to the declaration. Additionally, there are tools to extract such meta-documentation and generate external documentation.

Spoofax ecosystem One of the advantages of the Spoofax ecosystem is that improvements to Spoofax will benefit the PIE DSL as well. This happens in two ways. First, when Spoofax targets more IDEs, these IDEs are automatically also available for the PIE DSL. For example, there is a plugin for IntelliJ that does not have feature parity with Eclipse yet, but when it does achieve feature parity the PIE DSL can be edited in IntelliJ with all features that it has in Eclipse, without any specific effort from the developers of the PIE DSL. Language servers from the LSP[4] are another example. Once Spoofax can generate a language server from a Spoofax language specification, the PIE DSL can be used with Visual Studio Code[5], Visual Studio[6], Vim8[7], NeoVim[8], Emacs[9], Sublime text[10], Atom[11], Eclipse IDE[12], Eclipse Che[13], and

[4] https://microsoft.github.io//language-server-protocol/overviews/lsp/overview/
[5] https://code.visualstudio.com/
[6] https://visualstudio.microsoft.com/
[7] https://www.vim.org/
[8] https://neovim.io/
[9] https://www.gnu.org/software/emacs/
[10] https://www.sublimetext.com/
[11] https://atom.io/
[12] https://www.eclipse.org/ide/
[13] https://www.eclipse.org/che/

```
let showToJavaCommand = command-def {
  task-def = task-def mb.calc.task.debug.
      CalcShowToJava
  display-name = "To Java"
  parameters = [
    file = parameter {
      type = java mb.resource.ResourceKey
      argument-providers = [Context(File)]
    }
  ]
}
```

(a) An example of a command declaration in a Spoofax 3 `spoofaxc.cfg` file.

```
command { display-name: "To Java" }
func calcShowToJava(
  providers { [Context(File)] } file:
      ResourceKey
) -> CommandFeedback = ... // body omitted
```

(b) A very preliminary example of the command declaration in the PIE DSL. It uses annotations, which are not yet implemented.

Figure 6.4: An example of a command declaration for Spoofax 3, the current way in `spoofaxc.cfg` and a proposed way using annotations in the PIE DSL.

MS Paint IDE[14], among others.[15]

The other way in which improvements to Spoofax improve the DSL is direct editor features. For example, Spoofax recently introduced language parametric renaming (Misteli 2021), which is a generic rename refactoring that works for any language implemented in Spoofax, as long as that language properly declares references. To implement this in a specific language, the language just needs to import the stratego module with rename refactoring (`statix/runtime/renaming`) and create a menu item for it. New language projects have this refactoring enabled by default. Language parametric features such as these also extend the features of the PIE DSL without much effort from the developers of the DSL.

6.3.2 Integration with the ecosystem

The previous section proposed improvements to the PIE DSL, but in ways that merely achieve feature-parity with Java. One of the ways in which the DSL can truly provide value beyond an implementation in Java is with higher integration with the ecosystem where it is used.

Automatically add tasks from the PIE DSL to the PIE runtime It would be nice if the tasks defined in the PIE DSL would be added automatically to the PIE runtime. This is possible by generating a `MapTaskDefs` as part of the DSL compilation process and adding this `MapTaskDefs` to the runtime. If a task is added in the DSL, the `MapTaskDefs` will be regenerated which will automatically add the task to the runtime. Adding tasks to the runtime has to be done manually in Java, which is easily forgotten and does not cause problems until runtime.

Generate PIE DSL declarations for Spoofax 3 auto-generated tasks Spoofax 3 automatically generates parsing, analyzing, styling, tokenizing and a few other PIE tasks in Java for language projects. Generating the tasks in Java allows users to use the Java PIE API, or the PIE DSL by declaring the tasks as `foreign`. Generating these tasks as PIE DSL code would lock users into the PIE DSL, so that is not desired. However, it would be nice to generate a PIE DSL file with declarations for these tasks, so that users do not need to write these declarations themselves.

Declare Spoofax 3 commands within the DSL Another opportunity for better integration is specific to the Spoofax 3 IDE. It has commands such as 'To Java', 'Rename variable' and

[14] `https://ms-paint-i.de/`

[15] The full list can be found at `https://microsoft.github.io/language-server-protocol/implementors/tools/`.

'View scope graph'. Commands are defined in `spoofaxc.cfg`, a configuration file format introduced in Spoofax 3. An example of the current command definitions can be found in Figure 6.4a. It defines the command, specifies the task that implements the command, a display name, and a parameter with its type and where to get the argument. Overall, this definition takes 10 lines. It has duplicate information in line 1 and 2: the task returns a `CommandFeedback`, so that already indicates that it is a command. Line 5 and 6 declare the name and type of the `file` parameter, but that is already defined on the task as well.

A proposal for how to specify this command within the PIE DSL can be found in Figure 6.4b. The first line has an annotation that the function is a command, and specifies the display name for the command. Line 3 uses another annotation to specify where to get the argument for the parameter `file`. The current definition only defines the task, it still requires a task specification somewhere else. So this proposal would replace 10 lines of command specification with 2 annotations. Not only does it remove boilerplate, it also merges the definition of the command and the task, which makes it easier to keep both up to date.

6.4 Roadmap

To prioritize the proposed improvements in this section, we consider how much the feature is used, how much it would improve the feature, the cost of extending the language, how urgent the improvement is, and how much work it would be to design and implement it. Most of these criteria should be self-explanatory. A feature is considered urgent if it is a breaking change, i.e. if it would render currently legal PIE programs invalid or change their semantics. We want to implement or reject these as soon as possible because right now there is no PIE code in the wild yet, but that may change now that the DSL is getting to a point where it is a decent alternative to Java. That said, the current priorities are as follows:

1. *Resource dependencies*
   These are breaking changes, resource dependencies are a core part of the domain, and they currently prevent expressing tasks, so these changes should be implemented soon.

2. *Exception handling*
   ... if it turns out that it is used often. This is likely to be a pretty small change, so if it has uses besides the two in the case study that is some nice low hanging fruit.

3. *Generate foreign Java declarations*
   We are unsure how easy it will be to implement this. It might be possible to do this without doing name resolution on the Java code, in which case it is likely not too much work to set up a small project that can generate a large part of the declarations. If we do need name resolution, this may or may not be pushed back, depending on how much time name resolution is expected to take.

4. *Generate project-wide boilerplate*
   After the previous three, this is one of the last obvious ways that the DSL can provide easy value over Java.

5. *The easy cases for type inference*
   Implementing type inference is difficult, but as mentioned in subsection 6.1.14, there are one or two cases where type inference should be easy.

6. *More...*
   This is everything that we can relatively confidently say is important and easy enough that it is likely to be included. After these five items it is likely a deeper understanding of the domain is reached and new pain points with the DSL are uncovered. We will

keep applying the criteria from the start of this paragraph to decide our priorities, but it is impossible to say what these priorities will be.

Chapter 7

Conclusion

In this thesis, we looked at the PIE DSL. We analyzed the problems with PIE DSL 1 and proposed solutions with PIE DSL 2. We compared PIE DSL 2 to Java in a case study with Tiger to evaluate whether the PIE DSL is a better alternative and what could be done to improve it.

Unfortunately, the case study used a version of the DSL without generics and injected values. These features are used by a lot of tasks, so we also looked into the features each task uses and used that to look at the improvements of PIE DSL 2 compared to PIE DSL 1.

We analyzed the problems with PIE DSL 1 in chapter 2. These could be split into three areas: the language itself, the code base of the language, and the user experience of the language. For each of the listed problems we also contributed a (partial) solution. The problems and contributions are summarized in the list below.

Language

1. *Lack of modularity*
   PIE DSL 1 does not support multi-file PIE projects, and a naive implementation would lead to name collisions. We enabled multi-file analysis and implemented a module system to prevent name collisions.
2. *Generic Java classes and methods*
   We extended the DSL with parametric polymorphism to provide full interoperability with Java. This also allows us to define generic functions for accessing fields and enum values.
3. *Suppliers*
   The PIE framework added suppliers, which improve the performance of incrementality checks in certain cases. PIE DSL 1 did not support suppliers. We added support in PIE DSL 2.
4. *Injected values*
   PIE DSL 1 cannot express injected values. There is also no workaround, so any task that uses these was inexpressible. We added support for injected values in PIE DSL 2.
5. *Overly restrictive subtyping constraint in list literals, if-else expressions and comparisons*
   We used the least upper bound in list literals, if-else expressions and comparisons, which improves on the previous constraint that one sub-expression must be a supertype of the other(s).
6. *Class inheritance*
   We implemented inheritance for methods, which includes overriding and method collisions. It does not include overloading yet, so a method will shadow methods with the same name in a super class, even if the method in the super class has an incompatible signature.

Code base

7. *The compiler used string interpolation*
   The compiler for PIE DSL 1 uses string interpolation, which is simple and easy to implement, but makes it inefficient to optimize the generated Java code. The compiler now compiles to ASTs, which makes it possible to optimize the generated Java code.
8. *Lack of tests*
   Tests prevent regressions, which enables developers of the PIE DSL to iterate quicker. We added tests for the grammar, static semantics and the compiler.
9. *NaBL2 limits static semantics*
   The specification for the static semantics of PIE DSL 1 was written in NaBL2. NaBL2 does not have the expressive power required to implement most of the module system and for generics, is deprecated by now, and worst of all, it did not have much in the way of static checks and was tedious to debug. We specified the static semantics in Statix, the successor of NaBL2.
10. *No differentiation between syntactic and semantic types*
    The NaBL2 specification did not differentiate between syntactic types and semantic types. This often lead to forgetting to transform a syntactic type to a semantic type, which works for most types, but will fail for others. We split these into different sorts and constructors, which means that the Statix type system will tell us already in the editor when we make such a mistake, which saves us from running the code at all and also saves us the time spent debugging the issue.

User experience

11. *Lack of user documentation*
    The lack of documentation made it much harder for users to use the PIE DSL. We added extensive user documentation for types and expressions, which is a good start for documentation but not satisfactory yet. The documentation is still missing an explanation of language features like the module system and generics, tutorials, how-tos, background information and a list of known bugs and limitations with workarounds.

To evaluate PIE DSL 2, we asked the question 'Is the PIE DSL better than Java for expressing PIE pipelines?' We split this question into three sub-questions and answered each of them separately. The answers to the three sub-questions are summarized below.

1. *Does the DSL have less boilerplate than Java?*
   With only a single task that could be expressed in the DSL, the DSL does not have less boilerplate than Java.

2. *Does the DSL provide less opportunities for mistakes than Java?*
   The PIE DSL catches less mistakes than Java, especially when we consider the Java ecosystem (IntelliJ, compiler plugins) as well. An exception is nullability, where the DSL catches the same mistakes but with fewer false positives.

3. *Is the DSL easier to understand than Java?*
   If a task can be fully expressed in the DSL it is easier to understand than it would be in Java. However, if the task uses features that are not supported by the DSL then workarounds are required to overcome these limitations and it quickly becomes harder than Java.

The conclusion from the evaluation is that the PIE DSL is better than Java for expressing pipelines if the tasks in the pipeline only use features supported by the DSL. While we

should in theory now be able to express 11 of the 19 tasks, during the case study we could only express 1 because generics and injected values were still missing. However, these tasks may be expressible, they still use features which are not fully supported by the DSL. This means they require workarounds and helper methods. It remains to be seen how much of a detriment that is in practice. We expect that this will make using the DSL or Java more of a tradeoff, where neither is clearly better than the other.

The conclusion from the evaluation leads to a new hypothesis: we can increase the number of pipelines for which the PIE DSL is the clearly better option by extending the PIE DSL to (directly) support more features. To that end, we have listed a number of possible improvements in chapter 6. In particular, resource dependencies are outdated and block many tasks. Once this language feature is updated to interoperate well with the current version of the PIE API, it seems like the majority of tasks should be expressible in the DSL.

Overall, while the PIE DSL is not the better option in the majority of cases yet, this seems to be only because of the starters advantage Java has by being 22 years older than PIE. There might not be a clear best between Java or the DSL now, but Java will not improve much for implementing PIE pipelines, while the DSL still has many fairly easy improvements with large impact. We see no fundamental problems that would prevent the PIE DSL from being better than Java, so all it takes now is putting in the work to make that happen.

Bibliography

Amin, Nada and Ross Tate (2016). "Java and scala's type systems are unsound: the existential crisis of null pointers". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. Ed. by Eelco Visser and Yannis Smaragdakis. ACM, pp. 838–848. isbn: 978-1-4503-4444-9. doi: 10.1145/2983990.2984004. url: http://doi.acm.org/10.1145/2983990.2984004.

Antwerpen, Hendrik van, Pierre Néron, et al. (2016). "A constraint language for static semantic analysis based on scope graphs". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, pp. 49–60. isbn: 978-1-4503-4097-7. doi: 10.1145/2847538.2847543. url: http://doi.acm.org/10.1145/2847538.2847543.

Antwerpen, Hendrik van, Casper Bach Poulsen, et al. (2018). "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA. doi: 10.1145/3276484. url: https://doi.org/10.1145/3276484.

Appel, Andrew W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press. isbn: 0-521-58388-8.

Bracha, Gilad, N. Cohen, et al. (Jan. 2004). "Adding generics to the Java programming language". In:

Bracha, Gilad, Martin Odersky, et al. (1998). "Making the Future Safe for the Past: Adding Genericity to the Java Programming Language". In: *OOPSLA*, pp. 183–200. doi: 10.1145/286936.286957. url: http://doi.acm.org/10.1145/286936.286957.

Kats, Lennart C. L., Rob Vermaas, and Eelco Visser (2011a). "Integrated language definition testing: enabling test-driven language development". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, pp. 139–154. isbn: 978-1-4503-0940-0. doi: 10.1145/2048066.2048080. url: http://doi.acm.org/10.1145/2048066.2048080.

— (2011b). "Testing domain-specific languages". In: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, pp. 25–26. isbn: 978-1-4503-0942-4. doi: 10.1145/2048147.2048160. url: http://doi.acm.org/10.1145/2048147.2048160.

Kats, Lennart C. L. and Eelco Visser (2010). "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, pp. 444–463. isbn: 978-1-4503-0203-6. doi: 10.1145/1869459.1869497. url: https://doi.org/10.1145/1869459.1869497.

Konat, Gabriël, Sebastian Erdweg, and Eelco Visser (2018). "Scalable incremental building with dynamic task dependencies". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, pp. 76–86. doi: 10.1145/3238147.3238196. url: https://doi.org/10.1145/3238147.3238196.

Konat, Gabriël, Roelof Sol, et al. (2019). "Precise, Efficient, and Expressive Incremental Build Scripts with PIE". In: *Second Workshop on Incremental Computing (IC 2019)*.

Konat, Gabriël, Michael J. Steindorfer, et al. (2018). "PIE: A Domain-Specific Language for Interactive Software Development Pipelines". In: *Programming Journal* 2.3, p. 9. doi: 10.22152/programming-journal.org/2018/2/9. url: https://doi.org/10.22152/programming-journal.org/2018/2/9.

Misteli, Phil (2021). "Renaming for Everyone". MA thesis. Delft University of Technology. url: http://resolver.tudelft.nl/uuid:60f5710d-445d-4583-957c-79d6afa45be5.

Néron, Pierre et al. (2015). "A Theory of Name Resolution". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, pp. 205–231. isbn: 978-3-662-46668-1. doi: 10.1007/978-3-662-46669-8_9. url: http://dx.doi.org/10.1007/978-3-662-46669-8_9.

Rouvoet, Arjen et al. (2020). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA. doi: 10.1145/3428248. url: https://doi.org/10.1145/3428248.

Smits, Jeff and Eelco Visser (2017). "FlowSpec: declarative dataflow analysis specification". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, pp. 221–231. isbn: 978-1-4503-5525-4. doi: 10.1145/3136014.3136029. url: http://doi.acm.org/10.1145/3136014.3136029.

Smits, Jeff, Guido Wachsmuth, and Eelco Visser (2020). "FlowSpec: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis". In: *Journal of Computer Languages* 57, p. 100924. doi: 10.1016/j.cola.2019.100924. url: https://doi.org/10.1016/j.cola.2019.100924.

Sol, Roelof (2019). "Task Observability in change driven incremental build systems with dynamic dependencies". MA thesis. Delft University of Technology. url: http://resolver.tudelft.nl/uuid:3bd052ee-b8a0-4687-85d0-ca6df0b07d0d.

Souza Amorim, Luis Eduardo de and Eelco Visser (2020). "Multi-purpose Syntax Definition with SDF3". In: *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*. Ed. by Frank S. de Boer and Antonio Cerone. Vol. 12310. Lecture Notes in Computer Science. Springer, pp. 1–23. isbn: 978-3-030-58768-0. doi: 10.1007/978-3-030-58768-0_1. url: https://doi.org/10.1007/978-3-030-58768-0_1.

Visser, Eelco, Zine-El-Abidine Benaissa, and Andrew P. Tolmach (1998). "Building Program Optimizers with Rewriting Strategies". In: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. Ed. by Matthias Felleisen, Paul Hudak, and Christian Queinnec. Baltimore, Maryland, United States: ACM, pp. 13–26. doi: 10.1145/289423.289425. url: http://doi.acm.org/10.1145/289423.289425.

Will, Brian (2016). *Object-Oriented Programming is Bad*. YouTube video.

Acronyms

ADT abstract data type

API application programming interface

AST abstract syntax tree

DSL domain-specific language

ESV Editor Service

glb greatest lower bound

ID identifier

IDE Integrated Development Environment

JVM Java Virtual Machine

LSP Language Server Protocol

lub least upper bound

NaBL2 Name Binding Language 2

PIE Pipelines for Interactive Environments

SDF3 Syntax Definition Formalism 3

SPT SPoofax Testing language

URL Uniform Resource Locator

UTF8 Unicode Transformation Format 8

UX User Experience

VCS Version Control System

Appendix A

Case study data

This appendix shows the full aggregated data for the Tiger case study. It compares the version where all tasks are implemented in Java with the version where tasks are implemented in the PIE DSL when practical, and still in Java otherwise. Selected parts of this data were also presented in Table 4.2.

Value	Java	PIE DSL	Diff	Diff (%)
java lines including layout	2327	2289	-38	-1.63 %
java lines excluding layout	1919	1888	-31	-1.62 %
PIE DSL lines including libraries and layout	-	46	+46	+inf %
PIE DSL lines excluding libraries, including layout	-	19	+19	+inf %
PIE DSL lines including libraries, excluding layout	-	32	+32	+inf %
PIE DSL lines excluding libraries and layout	-	15	+15	+inf %
total lines including libraries and layout	2327	2335	+8	+0.34 %
total lines excluding libraries, including layout	2327	2308	-19	-0.82 %
total lines including libraries, excluding layout	1919	1920	+1	+0.05 %
total lines excluding libraries and layout	1919	1903	-16	-0.83 %
java characters including layout	91040	89620	-1420	-1.56 %
java characters excluding layout	80413	79095	-1318	-1.64 %
PIE DSL characters including libraries and layout	-	1990	+1990	+inf %
PIE DSL characters excluding libraries, including layout	-	934	+934	+inf %
PIE DSL characters including libraries, excluding layout	-	1957	+1957	+inf %
PIE DSL characters excluding libraries and layout	-	911	+911	+inf %
total characters including libraries and layout	91040	91610	+570	+0.63 %
total characters excluding libraries, including layout	91040	90554	-486	-0.53 %
total characters including libraries, excluding layout	80413	81052	+639	+0.79 %
total characters excluding libraries and layout	80413	80006	-407	-0.51 %
tasks implemented in java	19	18	-1	-5.26 %
tasks fully implemented in PIE DSL	-	1	+1	+inf %
tasks implemented in PIE DSL with helper function	-	-	0	0.00 %
total tasks implemented in PIE DSL	-	1	+1	+inf %
total tasks	19	19	0	0.00 %

Table A.1: Aggregated data for the Tiger case study. This description uses 'the Java version' to refer to the version of the case study that is fully implemented in Java, and 'the (PIE) DSL version' to refer to the version that also uses the PIE DSL when practical.
The fourth column ('diff') and the fifth column ('diff (%)') show the absolute and relative difference between the Java version and the DSL version, with Java as the baseline.
The row names are divided into four parts. Lines and characters refers to whether we count the lines or the characters.
The language can be Java, PIE DSL or total. Java counts only lines/characters in the Java parts of the case study version, PIE DSL counts the lines/characters in the PIE DSL parts of the version, and total counts both Java and PIE DSL for that version.
Libraries are code which is defined in a separate project. This code is never included for the Java version, since it is the exact same for both versions. For the PIE DSL version, libraries refers to foreign Java declarations for types and functions from libraries.
The final part of the row names is layout. For both Java and the PIE DSL, layout is whitespace and comments. It includes JavaDoc, since it is unfair to count that in Java when the benefits such documentation bring are not included in the counts for the PIE DSL. Including layout simply counts all layout/characters in a file. Excluding layout means that consecutive layout is counted as one character, and lines with only layout are excluded for the line count. For a detailed explanation of how layout is counted, see section 4.1.
Finally, the last section of the table shows task counts. 'Tasks implemented in Java' refers to tasks which are implemented in Java for each version. The tasks implemented in the PIE DSL are subdivided into two catagories: fully implemented in the DSL and implemented with helper functions. Tasks are considered to be implemented with helper functions if they need a function that is implemented in Java, which is specific for that task.

Appendix B

End-to-end test full files

The full files for the end-to-end test in Figure 3.15. This also includes `SimpleChecker.java`, which has shared functions for all end-to-end tests to easily set up the PIE runtime, pass a task, some input and expected output, and check if running the task on the provided input results in the expected output.

genericTestGen.pie

```
module mb:pie:lang:test:call:foreignFunc:generic

func func[C, D](c: C, d: D) -> D =
  foreign java mb.pie.lang.test.call.Bar#func

func main_generic() -> (string, bool) = {
  func[int, (string, bool)](217, ("generic", true))
}
```

Bar.java

```
package mb.pie.lang.test.call;

import java.io.Serializable;

public class Bar<T> implements Serializable {
    private T t;
    private String arg;

    public <E>Bar(T t, E e, String arg) {
        this.t = t;
        this.arg = arg;
    }

    public static <C, D> D func(C c, D d) {
        return d;
    }

    public <C, D> T method(C c, D d) {
        return t;
    }
```

```
21
22      @Override
23      public int hashCode() {
24          return t.hashCode() * 31 + arg.hashCode();
25      }
26
27      @Override
28      public boolean equals(Object obj) {
29          if (!getClass().equals(obj.getClass())) {
30              return false;
31          };
32          Bar that = (Bar) obj;
33          return this.t.equals(that.t) && this.arg.equals(that.arg);
34      }
35  }
```

GenericTest.java

```
1   package mb.pie.lang.test.call.foreignFunc.generic;
2
3   import mb.pie.api.ExecException;
4   import mb.pie.util.Tuple2;
5   import org.junit.jupiter.api.Test;
6
7   import static mb.pie.lang.test.util.SimpleChecker.assertTaskOutputEquals;
8
9   class GenericTest {
10      @Test void test() throws ExecException {
11          assertTaskOutputEquals(DaggergenericComponent.class, new Tuple2<String,
                Boolean>("generic", true));
12      }
13  }
```

genericComponent.java

```
1   package mb.pie.lang.test.call.foreignFunc.generic;
2
3   import dagger.Component;
4   import mb.pie.dagger.PieComponent;
5   import mb.pie.dagger.PieModule;
6
7   import javax.inject.Singleton;
8
9   @mb.pie.dagger.PieScope
10  @Component(modules = {PieModule.class, PieTestModule.class}, dependencies = {mb.log
        .dagger.LoggerComponent.class, mb.resource.dagger.ResourceServiceComponent.class
        })
11  public interface genericComponent extends PieComponent {
12      main_generic get();
13  }
```

PieTestModule.java

```
package mb.pie.lang.test.call.foreignFunc.generic;

import dagger.Module;
import dagger.Provides;
import dagger.multibindings.ElementsIntoSet;
import mb.pie.api.TaskDef;

import javax.inject.Singleton;
import java.util.HashSet;
import java.util.Set;

@Module
abstract class PieTestModule {
    @Provides @mb.pie.dagger.PieScope @ElementsIntoSet
    public static Set<TaskDef<?, ?>> provideTaskDefs(
        main_generic generic
    ) {
        final HashSet<TaskDef<?, ?>> taskDefs = new HashSet<>(1, 1);
        taskDefs.add(generic);
        return taskDefs;
    }
}
```

SimpleChecker.java

```
package mb.pie.lang.test.util;

import mb.log.dagger.DaggerLoggerComponent;
import mb.log.dagger.LoggerComponent;
import mb.log.dagger.LoggerModule;
import mb.pie.api.ExecException;
import mb.pie.api.MixedSession;
import mb.pie.api.None;
import mb.pie.api.TaskDef;
import mb.pie.dagger.PieComponent;
import mb.pie.dagger.PieModule;
import mb.pie.runtime.PieBuilderImpl;
import mb.resource.dagger.DaggerRootResourceServiceComponent;
import mb.resource.dagger.ResourceServiceComponent;
import mb.resource.dagger.RootResourceServiceComponent;

import java.io.Serializable;
import java.lang.reflect.InvocationTargetException;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class SimpleChecker {
    public static <O extends Serializable> O assertTaskOutputEquals(
```

```
24          Class<? extends PieComponent> componentClass, O expectedOutput) throws
                ExecException {
25          return assertTaskOutputEquals(componentClass, None.instance, expectedOutput
                );
26      }
27
28      public static <I extends Serializable, O extends Serializable> O
            assertTaskOutputEquals(
29          Class<? extends PieComponent> componentClass,
30          I input,
31          O expectedOutput
32      ) throws ExecException {
33          final O output = requireTask(componentClass, input);
34          assertEquals(expectedOutput, output);
35          return output;
36      }
37
38
39      public static <I extends Serializable, O extends Serializable> O requireTask(
40          Class<? extends PieComponent> componentClass
41      ) throws ExecException {
42          return requireTask(componentClass, None.instance);
43      }
44
45      public static <I extends Serializable, O extends Serializable> O requireTask(
46          Class<? extends PieComponent> componentClass,
47          I input
48      ) throws ExecException {
49          try {
50              Object builder = componentClass.getMethod("builder").invoke(null);
51              builder.getClass().getMethod("pieModule", PieModule.class).invoke(
                    builder, new PieModule(PieBuilderImpl::new));
52              final LoggerComponent loggerComponent = DaggerLoggerComponent.builder()
                    .loggerModule(LoggerModule.noop()).build();
53              builder.getClass().getMethod("loggerComponent", LoggerComponent.class).
                    invoke(builder, loggerComponent);
54              final RootResourceServiceComponent resourceServiceComponent =
                    DaggerRootResourceServiceComponent.builder().loggerComponent(
                    loggerComponent).build();
55              builder.getClass().getMethod("resourceServiceComponent",
                    ResourceServiceComponent.class).invoke(builder,
                    resourceServiceComponent);
56              PieComponent component = (PieComponent)builder.getClass().getMethod("
                    build").invoke(builder);
57              try(MixedSession session = component.getPie().newSession()) {
58                  @SuppressWarnings("unchecked") final TaskDef<I, O> main = (TaskDef<
                        I, O>)component.getClass().getMethod("get").invoke(component);
59                  return session.require(main.createTask(input));
60              }
61          } catch(NoSuchMethodException e) {
62              throw new RuntimeException("Expected method to exist", e);
63          } catch(IllegalAccessException e) {
```

```
64              throw new RuntimeException("Expected method to be accessible", e);
65          } catch(InvocationTargetException e) {
66              throw new RuntimeException("Unexpected exception", e);
67          } catch(InterruptedException e) {
68              throw new RuntimeException(e);
69          }
70      }
71  }
```

Appendix C

A PIE DSL implementation of Unsound.Java

Amin and Tate (2016) show that generics in Java are unsound. They provide a Java class that demonstrates this unsoundness. We were interested whether that same unsoundness exists in the PIE DSL, so we translated the file to the DSL.

Unfortunately, the file has some type-checking errors due to some bugs. These should be solvable though. Additionally, generic functions give an error because they cannot be compiled, but that does not matter for type-checking. Finally, `b` in `upcast` cannot be assigned to `A` because the declared subtyping `B <: A` is not used in the body of the task. It probably should though, so that might get implemented in the future.

It seems nevertheless unlikely that this same unsoundness exists in the DSL, because it does not use the bound implied by `Constrain[U, _ :> T]`, so the call to `upcast` will not type-check.

Unsound.java

```
1  class Unsound {
2      static class Constrain<A, B extends A> {}
3      static class Bind<A> {
4          <B extends A>
5          A upcast(Constrain<A,B> constrain, B b) {
6              return b;
7          }
8      }
9      static <T,U> U coerce(T t) {
10         Constrain<U,? super T> constrain = null;
11         Bind<U> bind = new Bind<U>();
12         return bind.upcast(constrain, t);
13     }
14     public static void main(String[] args) {
15         String zero = Unsound.<Integer,String>coerce(0);
16     }
17 }
```

unsound.pie

```
module org:example:unsound

data Constrain[A, B <: A] = foreign java Constrain {}

data Bind[A] = foreign java Bind {}
func newBind[A]() -> Bind[A] = foreign java constructor Bind
func upcast[A, B <: A](this: Bind[A], constrain: Constrain[A, B]?, b: B) -> A = b

func coerce[T, U](t: T) -> U = {
  val constrain: Constrain[U, _ :> T]? = null;
  val bind: Bind[U] = newBind[U]();
  upcast[U, T](bind, constrain, t)
}

func main() -> unit = {
  val zero: string = coerce[int, string](0);
  unit
}
```

Appendix D

Ideal DSL code for case study

The following file is an attempt to see what the tasks in the case study could look like if the PIE DSL supported a few more features. This has many TODOs, particularly for resource dependencies. We used this file during development of the DSL to see what language features were still missing and how these features would be used in actual code.

The file parses, type-checks and compiles in the latest version of the DSL, but the generated Java files have not been run or type-checked with Java. Nevertheless, it can be used as a preview of what the DSL could look like and can give some indication of how much boilerplate the case study would have in the latest version of the DSL.

tiger.pie

```
1  module spoofax3:example:tiger
2
3  import java:lang:Exception
4  import java:nio:charset:{CharSet, StandardCharsets}
5  import java:util:optionalOfNullable
6  import javax:inject:Provider
7  import mb:aterm:common:{termToString, termToStringWithMaxWidth}
8  import mb:constraint:common:{SingleFileResult, MultiFileResult}
9  import mb:common:editing:TextEdit
10 import mb:common:message:{Severity, Messages, KeyedMessages,
       createKeyedMessagesBuilder}
11 import mb:common:option:{Option, some, none}
12 import mb:common:region:Region
13 import mb:common:result:{Result, ok, err}
14 import mb:common:style:{styleNameFromString, Styling}
15 import mb:common:token:Token
16 import mb:common:util:{createEmptyListView, listViewOf}
17 import mb:completions:common:{CompletionResult, createCompletionResult,
       createCompletionProposal, CompletionProposal}
18 import mb:jsglr:common:{JSGLRTokens, getSmallestTermEncompassingRegion}
19 import mb:resource:{ResourceKey, ResourceService}
20 import mb:resource:hierarchical:{
21   ResourcePath, HierarchicalResource, pathResourceWalker, noHiddenPathMatcher,
       extensionPathMatcher, fileResourceMatcher, pathResourceMatcher
22 }
23 import mb:stratego:common:{StrategoRuntime, invokeStrategoStrategy}
24 import mb:tiger:TigerStyler
```

```
import mb:tiger:spoofax:task:reusable:{TigerParse, analyze, analyzeMulti}
import mb:spoofax:core:language:command:{
  CommandFeedback, commandFeedbackOf, commandFeedbackOfFile, commandFeedbackOfText,
      commandFeedbackOfTryExtractMessagesFrom, showFileFeedback, showTextFeedback
}
import org:spoofax:interpreter:terms:IStrategoTerm
import std:adapters:java:{getField, getStaticField, enumValue}
import std:types:int:getIntMaxValue as maxInt
import std:types:path:{getHierarchicalResource, write}
import std:types:string:joinStrings

func completeTaskDef(astProvider: supplier[IStrategoTerm?]) -> CompletionResult? =
    {
  val ast = astProvider.get();
  if (ast == null)
    return null;

  createCompletionResult(listViewOf[CompletionProposal]([
    createCompletionProposal("mypackage", "description", "", "", "mypackage",
        styleNameFromString("meta.package")!, createEmptyListView[TextEdit](), false
        ),
    createCompletionProposal("myclass", "description", "", "T", "mypackage",
        styleNameFromString("meta.class")!, createEmptyListView[TextEdit](), false)
  ]), true)
}


func listLiteralVals(
  astSupplier: supplier[Result[IStrategoTerm, _ <: Exception]]
) -> Result[string, _ <: Exception] = inject runtime: StrategoRuntime in {
  val termResult = astSupplier.get();
  if (termResult.isErr())
    return err[string, Exception](termResult.unwrapErr());

  ok[string, _ <: Exception](termToStringWithMaxWidth(
    runtime.invoke("list-of-def-names", termResult.unwrap()),
    maxInt()
  ))
}

// todo: equal to listLiteralVals?
func listDefNames(
  astSupplier: supplier[Result[IStrategoTerm, _ <: Exception]]
) -> Result[string, _ <: Exception] = inject runtime: StrategoRuntime in {
  val termResult = astSupplier.get();
  if (termResult.isErr())
    return err[string, Exception](termResult.unwrapErr());

  ok[string, _ <: Exception](termToStringWithMaxWidth(
    runtime.invoke("list-of-def-names", termResult.unwrap()),
    maxInt()
  ))
```

```
}


func style(tokensSupplier: supplier[Option[JSGLRTokens]]) -> Option[Styling] =
  inject styler: TigerStyler in {
    val tokens = tokensSupplier.get();
    if(tokens.isEmpty())
      none[Styling]()
    else
      some[Styling](styler.style(tokens.unwrap().getTokens()))
  }


func check(
  file: ResourceKey, rootDirectoryHint: ResourcePath?
) -> KeyedMessages = inject parse: TigerParse in {
  val messagesBuilder = createKeyedMessagesBuilder();
  val parseInputBuilder = parse.inputBuilder().withFile(file).rootDirectoryHint(
      optionalOfNullable[ResourcePath](rootDirectoryHint));
  val parseMessages = parseInputBuilder.buildMessagesSupplier().get();
  messagesBuilder.addMessages(file, parseMessages);
  val analysisResult = analyze(file, parseInputBuilder.buildRecoverableAstSupplier
      ());
  if (analysisResult.isOk()) {
    val (context, result) = analysisResult.unwrap();

    messagesBuilder.addMessages(
      getField[SingleFileResult, ResourceKey](result, "resource"),
      getField[SingleFileResult, Messages](result, "messages")
    )
  } else {
    messagesBuilder.addMessage("Analysis failed", analysisResult.unwrapErr(),
      enumValue[Severity]("mb.common.message.Severity", "Error"))
  };
  messagesBuilder.build()
}


func checkAggregator(input: ResourcePath) -> KeyedMessages = {
  val messagesBuilder = createKeyedMessagesBuilder();
  // requires input; // todo: ResourcePath is a path type?
  val rootDirectory = getHierarchicalResource(input);

  // todo: multiple filters; filters isDirectory, isFile, hidden
  [requires dir | dir <- walk rootDirectory /*with isDirectory(true) and hidden(
      false)*/];
  [{
    val key = (val tmp: ResourceKey? = null)!; // todo: file.getKey()
    val messages = check(key, input);
    messagesBuilder.addKeyedMessages(messages)
  } | file <- walk rootDirectory with /*isFile(true) and*/ extension "tig"];

```

```
121   messagesBuilder.build()
122 }
123
124
125 func checkMulti(input: ResourcePath) -> KeyedMessages =
126   inject parse: TigerParse in {
127     val messagesBuilder = createKeyedMessagesBuilder();
128     // requires input; // todo: ResourcePath is a path type?
129     val rootDirectory = getHierarchicalResource(input);
130
131     val builder = parse.inputBuilder().rootDirectoryHint(optionalOfNullable[
          ResourcePath](input));
132     // todo: multiple filters; filters isDirectory, isFile, hidden
133     [requires dir | dir <- walk rootDirectory /*with isDirectory(true) and hidden(
          false)*/];
134     [{
135       val filePath = {val tmp: ResourcePath? = null}!; // todo: file.getPath();
136       messagesBuilder.addMessages(filePath, builder.withFile(filePath).
            buildMessagesSupplier().get())
137     } | file <- walk rootDirectory with /*isFile(true) and*/ extension "tig"];
138
139     val walker = pathResourceWalker(noHiddenPathMatcher());
140     val matcher = fileResourceMatcher().and(pathResourceMatcher(
          extensionPathMatcher("tig")));
141     val analysisResult = analyzeMulti(input, parse.
          createRecoverableMultiAstSupplierFunction(walker, matcher));
142     if (analysisResult.isOk()) {
143       val (messagesFromAstProviders, context, result) = analysisResult.unwrap();
144       messagesBuilder.addKeyedMessages(getField[MultiFileResult, KeyedMessages](
            result, "messages"));
145       messagesBuilder.addKeyedMessages(messagesFromAstProviders)
146     } else {
147       messagesBuilder.addMessageWithKey("Project-wide analysis failed",
            analysisResult.unwrapErr(),
148         enumValue[Severity]("mb.common.message.Severity", "Error"), input)
149     };
150
151     messagesBuilder.build()
152   }
153
154
155 func compileDirectory(dir: ResourcePath) -> CommandFeedback =
156   inject parse: TigerParse, resourceService: ResourceService in {
157     // val directory = requires dir with isFile and extension "tig"; // todo
158     val directory = ./dir;
159     val messagesBuilder = createKeyedMessagesBuilder();
160
161     val defNameStrings = [{
162       val filePath = {val tmp: ResourceKey? = null}!; // todo: file.getPath();
163       val defNames = listDefNames(parse.inputBuilder().withFile(filePath).
            buildAstSupplier());
164       if (defNames.isOk()) {
```

```
        defNames.unwrap() + "\n"
      } else {
        messagesBuilder.addMessageWithKey("Listing definition names for '$file' failed", defNames.unwrapErr(),
          enumValue[Severity]("mb.common.message.Severity", "Error"), filePath);
        "[]\n"
      }
    } | file <- list directory with /*isFile and*/ extension "tig"];

    val str = "[\n  ${joinStrings(defNameStrings, ", ")}]";

    // todo: use dir + "_defnames.aterm" when path <: ResourcePath
    val generatedPath = dir.appendSegment("_defnames.aterm");
    val generatedResource = resourceService.getHierarchicalResource(generatedPath);
    // todo: requires helper method (can't getBytes, returns byte[])
    // generatedResource.writeBytes(str.getBytes(StandardCharsets.UTF_8()));
    write(generatedResource, getStaticField[CharSet]("java.nio.charset.StandardCharsets", "UTF_8"), str);
    // generates generatedResource by hash; // todo when path <: ResourcePath
    commandFeedbackOf(messagesBuilder.build(), showFileFeedback(generatedPath))
  }


func compileFile(file: ResourcePath) -> CommandFeedback =
  inject parse: TigerParse, resourceService: ResourceService in {
    val astSupplier = parse.inputBuilder().withFile(file).buildAstSupplier();
    val listedLiteralVals = listLiteralVals(astSupplier);
    if (listedLiteralVals.isOk()) {
      val generatedPath = file.replaceLeafExtension("literals.aterm");
      val generatedResource = resourceService.getHierarchicalResource(generatedPath);
      write(generatedResource, getStaticField[CharSet]("java.nio.charset.StandardCharsets", "UTF_8"), listedLiteralVals.unwrap());
      // generates generatedResource by hash; // todo when path <: ResourcePath
      commandFeedbackOfFile(showFileFeedback(generatedPath))
    } else {
      commandFeedbackOfTryExtractMessagesFrom(listedLiteralVals.unwrapErr(), file)
    }
  }

func encodeBase64(string) -> string = foreign java mb.tiger.piedsl.Helpers#encodeUTF8ToBase64
func compileFileAlt(
  file: ResourcePath,
  listDefNames: bool,
  base64Encode: bool,
  compiledFileNameSuffix: string
) -> CommandFeedback =
  inject parse: TigerParse, resourceService: ResourceService in {
    val astSupplier = parse.inputBuilder().withFile(file).buildAstSupplier();
    val strResult = if(listDefNames) listDefNames(astSupplier) else listLiteralVals(astSupplier);
```

```
    if (strResult.isOk()) {
    // note: Java code uses mapCatching, which catches any exceptions thrown in
        this block and uses the else block instead
      val str = if (base64Encode) encodeBase64(strResult.unwrap()) else strResult.
          unwrap();
      val generatedPath = file.replaceLeafExtension(compiledFileNameSuffix);
      val generatedResource = resourceService.getHierarchicalResource(generatedPath
          );
      write(generatedResource, getStaticField[CharSet]("java.nio.charset.
          StandardCharsets", "UTF_8"), str);
      // generates generatedResource by hash;  // todo when path <: ResourcePath
      commandFeedbackOfFile(showFileFeedback(generatedPath))
    } else {
      commandFeedbackOfTryExtractMessagesFrom(strResult.unwrapErr(), file)
    }
  } @ file


func ideTokenize(key: ResourceKey) -> Option[JSGLRTokens] =
  inject parse: TigerParse in
    parse.inputBuilder().withFile(key).buildTokensSupplier().get().ok()


func showAnalyzedAst(key: ResourceKey, region: Region?) -> CommandFeedback =
  inject parse: TigerParse in {
    val analysis = analyze(key, parse.inputBuilder().withFile(key).buildAstSupplier
        ());
    if (analysis.isOk()) {
      val (_, singleFileResult) = analysis.unwrap();
      val ast = getField[SingleFileResult, IStrategoTerm](singleFileResult, "ast");
      val astPart = if (region != null)
        getSmallestTermEncompassingRegion(ast, region!) else ast;
      commandFeedbackOfText(showTextFeedback(termToString(astPart), "Analyzed AST
          for '$key'"))
    } else {
      commandFeedbackOfTryExtractMessagesFrom(analysis.unwrapErr(), key)
    }
  }


func showDesugaredAst(key: ResourceKey, region: Region?) -> CommandFeedback =
  inject
    parse: TigerParse,
    strategoRuntimeProvider: Provider[StrategoRuntime]
  in {
    val parseResult = parse.inputBuilder().withFile(key).buildAstSupplier().get();
    if (parseResult.isOk()) {
      val ast = parseResult.unwrap();
      val astPart = if (region != null)
        getSmallestTermEncompassingRegion(ast, region!) else ast;
      val desugarResult = invokeStrategoStrategy(strategoRuntimeProvider.get(), "
          desugar-all", ast);
```

```
256       if (desugarResult.isOk())
257         commandFeedbackOfText(showTextFeedback(termToString(desugarResult.unwrap())
                , "Desugared AST for '$key'"))
258       else
259         commandFeedbackOfTryExtractMessagesFrom(desugarResult.unwrapErr(), key)
260     } else {
261       commandFeedbackOfTryExtractMessagesFrom(parseResult.unwrapErr(), key)
262     }
263   }
264
265
266 func showParsedAst(key: ResourceKey, region: Region?) -> CommandFeedback =
267   inject parse: TigerParse in {
268     val parseResult = parse.inputBuilder().withFile(key).buildAstSupplier().get();
269     if (parseResult.isOk()) {
270       val ast = parseResult.unwrap();
271       val astPart = if (region != null)
272         getSmallestTermEncompassingRegion(ast, region!) else ast;
273       commandFeedbackOfText(showTextFeedback(termToString(ast), "Parsed AST for '
              $key'"))
274     } else {
275       commandFeedbackOfTryExtractMessagesFrom(parseResult.unwrapErr(), key)
276     }
277   }
278
279
280 func showPrettyPrintedText(key: ResourceKey, region: Region?) -> CommandFeedback =
281   inject
282     parse: TigerParse,
283     strategoRuntimeProvider: Provider[StrategoRuntime]
284   in {
285     val parseResult = parse.inputBuilder().withFile(key).buildAstSupplier().get();
286     if (parseResult.isOk()) {
287       val ast = parseResult.unwrap();
288       val astPart = if (region != null)
289         getSmallestTermEncompassingRegion(ast, region!) else ast;
290       val prettyprintResult = invokeStrategoStrategy(strategoRuntimeProvider.get(),
              "pp-Tiger-string", ast);
291       if (prettyprintResult.isOk())
292         commandFeedbackOfText(showTextFeedback(termToString(prettyprintResult.
              unwrap()), "Pretty-printed text for '$key'"))
293       else
294         commandFeedbackOfTryExtractMessagesFrom(prettyprintResult.unwrapErr(), key)
295     } else {
296       commandFeedbackOfTryExtractMessagesFrom(parseResult.unwrapErr(), key)
297     }
298   }
299
300
301 func showScopeGraph(key: ResourceKey, region: Region?) -> CommandFeedback =
302   inject
303     parse: TigerParse,
```

```
304     strategoRuntimeProvider: Provider[StrategoRuntime]
305   in {
306     val analysis = analyze(key, parse.inputBuilder().withFile(key).buildAstSupplier
          ());
307     if (analysis.isOk()) {
308       val (context, result) = analysis.unwrap();
309       val runtime = strategoRuntimeProvider.get().addContextObject(context);
310       val termFactory = runtime.getTermFactory();
311       val inputTerm = termFactory.makeTuple(getField[SingleFileResult,
            IStrategoTerm](result, "ast"), termFactory.makeString(key.asString()));
312       val analysisTerm = invokeStrategoStrategy(runtime, "spoofax3-editor-show-
            analysis-term", inputTerm);
313       if (analysisTerm.isOk())
314         commandFeedbackOfText(showTextFeedback(termToString(analysisTerm.unwrap()),
              "Scope graph for '$key'"))
315       else
316         commandFeedbackOfTryExtractMessagesFrom(analysisTerm.unwrapErr(), key)
317     } else {
318       commandFeedbackOfTryExtractMessagesFrom(analysis.unwrapErr(), key)
319     }
320   }
```