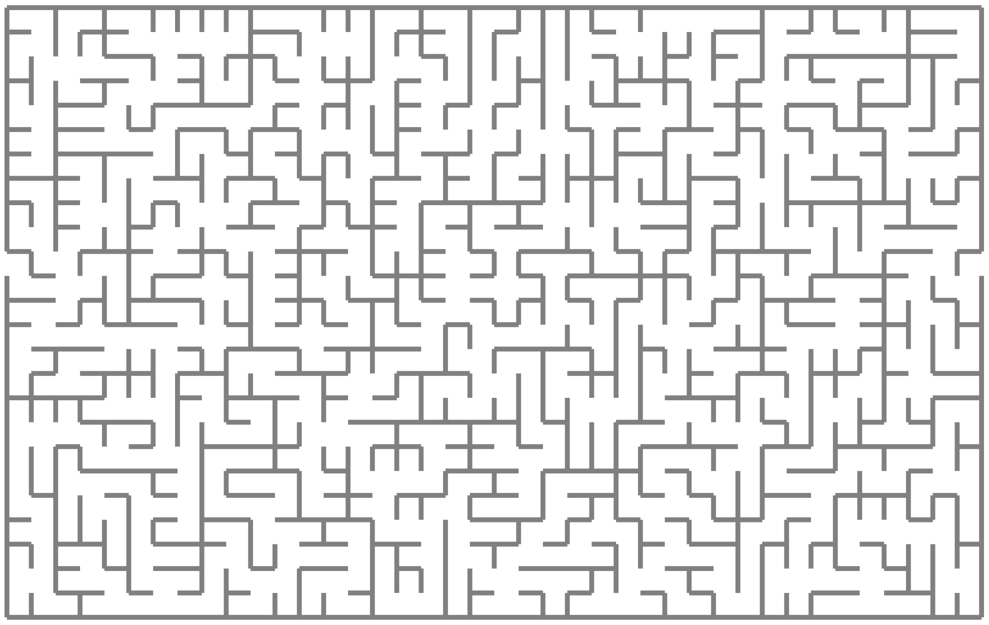
Extending the Domain Specific Language for the Pipelines for Interactive Environments build system

Version of June 18, 2021



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
Email:	`irgwilms@gmail.com`

Abstract

Abstract here.

Preface

Preface here.

Ivo Wilms
Delft, the Netherlands
June 18, 2021

Contents

List of Figures

List of Tables

Todo list

Chapter 1

Introduction

Transforming some inputs into some outputs is often not done with a single program, but with multiple programs that have to be executed in sequence, passing the result between them. This is called a pipeline, and for the purposes of this thesis, the steps in such a pipeline are called tasks. For smaller projects, such a pipeline can be expressed in a shell script like a bash script or a bat script, but as the project grows such build scripts often become unmaintainable.(Hatch and Flatt 2018; "Introducing Ant" 2006) Additionally, the full pipeline will be executed each build, even if nothing changed. There is a multitude of build systems that aim to resolve these problems. question: In your last feedback, you asked "What is a build system?". I expect my audience to know what a build system is, so what is this question asking? Well known examples are Make, Maven, Ant, Gradle and MSBuild.[citation needed] One of the issues with shell scripts is that dependencies are implicitly encoded by the order in which steps are executed. This does not scale well because steps that depend on each other can get arbitrarily far away from each other, which makes it hard to see the dependencies. Build systems aim to keep build scripts maintainable by providing a concise way to express dependencies of and between tasks. question: Which of these examples is better?

```
// runBenchmark depends on the files generated by generateTestData.
// This is declared within the function, so this dependency is visible
// regardless of how many functions are between these two.
func generateTestData(outputDir: path) -> unit = {
  exec("generateData", [("dir", outputDir)]);
  provides test_data_A.txt; provides test_data_B.txt;
  unit
}
// ... potentially many other functions here ...
func runBenchmark(testDir: path) -> benchMarkResults = {
  requires test_data_A.txt; requires test_data_B.txt;
  exec("benchmark", testDir)
}

// before parsing the program we first
// need to read the file into a string
func parseFile(file: path) -> AST = {
  val program: string = read file;
  parse(program)
}
```

They also keep track of which inputs changed in order to build incrementally, which saves a lot of time in the case of small changes.

```
func main() -> unit = {
  val inputFiles: path* = walk ./lib/pieLib/ with extension "pie"
    + ./src/main/main.pie;
  val programs = [(file.getSimpleName(), parseFile(file)) | file <- inputFiles];
  val analysisResult = analyzeProject(programs);
  val compiled = [(name, compile(name, program, analysisResult)) | (name, program) <-
      programs];
  [writeFile(./out/java/$name.java, program) | (name, program) <- compiled];
  unit
}
```

In the example above, the main function uses the files in the directory `lib/pieLib/` and the single file `src/main/main.pie`. It parses each file individually, analyzes all files together, compiles files using the analysis results and finally writes the compiled programs to the file system. If we now update `main.pie` and rerun the build, a simple build system would parse, re-analyze and recompile everything. A more advanced build system could see that only `main.pie` was updated, and could therefor parse only that file, and then run everything else for everything again, as the analysis depends on all files and therefor all compilation results depend on all files. This at least saves us having to parse all the library files. An even more advanced build system might check if the update to `main.pie` actually affected the program. Maybe the update to main was just editing a comment, in which case the program is unchanged, and after parsing `main.pie` everything is up-to-date.

todo: explain what precise dynamic dependencies are (and why we care?) A build system is precise if it keeps track of the exact dependencies of a task. A dynamic dependency is a dependency that can only be resolved at runtime. Almost no build system supports precise dynamic dependencies in a concise way.[citation needed] verify this claim, back it up with some references, or make it more specific so that it is true To fill this gap, PIE was developed by Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg and Eelco Visser in (Konat et al. 2018). PIE is a build system which expresses pipeline steps as functions. A function with its arguments forms a task. Each task can have dependencies on other tasks and on files. These dependencies are discovered dynamically during runtime. Because the dependencies are kept track of individually for each set of arguments, PIE can keep precise dynamic dependencies. PIE also provides incremental execution, both from the goal of making a certain task up-to-date and from a set of changed files.

PIE is implemented as a framework in Java. As such, it has a lot of boilerplate. Java is also poor for expressing concepts from the domain of pipelines. To solve these issues, the PIE DSL was developed. The PIE DSL allows a pipeline developer to express the relevant parts of a task definition in a language that allows easily expressing common concepts in pipelines. Because the DSL has significantly less boilerplate, it saves the developer time, is easier to read and has less duplication which improves maintainability and prevents bugs. Does this follow from the evaluation? The concise PIE DSL task definition is compiled to Java, which generates all the Java boilerplate.

While the PIE DSL and the compiler had implementations, they had several shortcomings:

1. Existing compilers were not maintainable.
2. The DSL does not scale to larger projects.
3. The implementation of the static semantic DSL is written in an outdated language.
4. The DSL has limitations in expressiveness.

These problems will be explained in more detail in the next chapter, Problem analysis.

We set out to solve these issues in the PIE DSL. These improvements were subject to the following non-functional requirements:

- Keep the PIE DSL general.

  - It could also compile to another language if another compiler and PIE framework were written in that language.
  - It should work for pipelines in general, not just pipelines in our specific use case.

- Since PIE is meant to execute pipelines that give realtime feedback, the code generated by compiling the DSL should be reasonably performant.

- Keep the PIE DSL extendable. Do not add features that are incompatible with future extensions of the language where this is reasonably possible.

We solved the above issues while meeting the provided requirements. For our solution, we:

later: did we do all of these?

- Implemented a new compiler for the DSL

- Implemented static semantics of the DSL in a more modern language

- Add modules, context parameters (and generics?) to the DSL

- Evaluation of performance for generated code (?)

In the end, this thesis makes the following contributions: Section 2 gives an explanation of our use cases and explains the problems in more detail. Section 3 lists the improvements that were made to the PIE DSL. Section 4 evaluates the new DSL in three case studies:

- simple transformation from Tiger (a toy language)

- database pipelines at Oracle

- testing pipelines at Oracle

It also compares expressing a pipeline in the DSL or directly in the Java framework. question: Alternatively: ”we make ... contributions”, and ”we provide an explanation ...” (follows the scalable incremental building paper literally). Which is better?

Chapter 2

Problem analysis

2.1 Use case

Language engineering is the discipline of designing and implementing programming languages. The state of the art right now is to implement languages ad-hoc, i.e. most major languages implement most of their parsing, syntax highlighting, static analysis, etc. from scratch.[citation needed] A language workbench is an Integrated Development Environment (IDE) for developing programming languages. Integrating relevant tools into the editor makes it easier to implement a language, which in turn makes it easier to iterate and experiment with language design.

Spoofax is a language workbench with the goal of abstracting over the implementation details of a language, and allowing language developers to focus solely on the design of their language, instead of how it should be implemented. It does this using multiple declarative meta-DSLs. The language developer can write a specification of their language in these DSLs and Spoofax automatically builds an implementation from these specifications.

Spoofax allows language developers to open example files of their language. Spoofax uses the language specification to generate syntax highlighting, error reporting and menu actions for things such as compiling the example file to another language and debugging the example file or language specification. Having an example file open allows language developers to quickly test their specification, which achieves the ”easy to iterate and experiment” goal of language workbenches. Although running code immediately on example programs is great for quick feedback, testing by running and checking on examples only gets you so far. Testing frameworks and unit tests address this in regular software development. Spoofax includes SPT, a meta-DSL and framework for testing a language specification. SPT is mostly useful to perform regression testing, but it also serves as documentation on how the language is supposed to work and it encourages one to think about edge cases.

Keeping Spoofax editors up-to-date follows a pipeline.

1. The file is parsed, using the parse table. In the case of meta-DSLs, the parse table is included with Spoofax. For user-defined languages (e.g. the language currently under development), the parse table is a build artifact. This step produces an ATerm of the filequestion: not explained yet, refer to future section or use AST for now? and notes, warnings and errors.

2. Syntax highlighting is added to the file using the parse results in the ATerm, generated colors from the grammar and custom colorings defined in configuration files.

3. Depending on configuration, static analysis is performed on either only the ATerm for this file or the ATerms from all files in this language within the project. The specific semantics to apply are defined in artifacts from the language implementation. This

produces notes, warnings and errors markers which are added to the editor. It also produces analysis results. These can be used by menu actions, for example when compiling to another language.

Because Spoofax is an IDE, language developers expect it to update as they are typing. This means that Spoofax needs to run editor services as fast as possible, preferably within a second. This is not always possible. Building a language specification to update example programs when a specification file is updated takes longer than a second for larger languages. Nevertheless, if the specification is unchanged and edits were only made to an example file, parsing, syntax highlighting and static analysis need to run. In this case, the language specification is unchanged and so does not need to be rebuild. Instead, the build artifacts can be cached and reused. Keeping track of which dependencies changed and only rerunning tasks that need updating is the task of the build system.

The current build system, Pluto, uses files as inputs and outputs for tasks. This implicitly means that files can be used as caches, Pluto only needs to keep track of which file is up to date (which I suppose it could do by checking the modification timestamp, but I don't know enough about the internals of Pluto to give a definitive answer question: Do you know this?). Reading and writing was left up to the pipeline developers, which meant that every task needs to implement its own file writing and reading. Additionally, files systems are different across operating systems (OSs). As an example, Windows requires that file paths are at most 260 characters long. If the project is already inside a deeply nested folder, sometimes files simply cannot be generated without intentionally shortening the name, which introduces complexity. So every task needs to write robust code or needs some out of framework way to share such robust code. This lead to duplicate effort and a higher probability of bugs.

The next major version of Spoofax, Spoofax 3, will use PIE as its build system. PIE includes caching out of the box. Tasks are Java classes that implement an interface which basically specifies that they have a method `exec`. The inputs and outputs of this method are automatically cached by the PIE framework. It saves the inputs and corresponding outputs as a database in a single file, which avoids many of the issues with files. It also means that the pipeline developer need not concern themselves with caching.

2.2 Background information

2.2.1 ATerms

Programs are expressed in human readable syntax, but this is hard and inefficient to manipulate directly. Instead, code is parsed to an abstract syntax tree (AST), which throws away the syntax and keeps the parts of a program that can change. In Spoofax, these ASTs are represented using ATerms. An ATerm consists of a constructor, arguments and possibly some annotations. As an example, the expression `x * 9 - 3` is represented as `Minus(Times(Var("x"), Int("9")), Int("3"))`. Spoofax meta-languages can pattern-match on these constructors to define rules, such as rules in Stratego to transform a constructor or rules in Statix to define the static semantics.

2.2.2 Ways to compile

There are two ways to compile from a source language to a target language.

The simplest way to compile a source language to a target language is to compile fragments of the source language into fragments of the target language, write a template in the target language and insert the compiled fragments into that template. This method is called string interpolation. It has the advantage of being fairly easy to read and write, but it has several disadvantages. A major disadvantage of string compilation is that it does not have static checks. This means that the Stratego compiler todo: general term for "compiler compiler" does not check that the generated code will form a correct program, or even that it is syntactically correct. A simple typo in the generated code is only caught at runtime instead of compile time. This leads to longer development times and transformation rules that are tested on a small subset of possible inputs, instead of verified for all possible inputs. String interpolation is also inefficient in case one wants to do some post-processing on the generated code, such as performing optimizations. To perform such optimizations, the code needs to be parsed to an AST, so concatenating strings only to then parse the strings is inefficient.

To avoid bugs due to typos in the generated code fragments, one can transform to an AST of the target language instead. The AST is then transformed to code by a generated prettyprinter. Additionally, it is easier to check correctness of an AST. Stratego currently verifies the arity of constructors. Verifying the arguments to a constructor are of the right type (e.g. checking that `x` and `y` in `Add(x, y)` are expressions instead of, say, import statements) is currently a work in progress. Transforming to an AST also means that the transformed program is immediately ready for post-processing, no parsing required.

2.2.3 Language projects in Spoofax

question: Move to "Use case" above?

A language in Spoofax is represented as a language project. Language projects contain several types of files. Syntax Definition Formalism 3 (SDF3) files specify the grammar of the language. The productions in SDF3 files are also used to generate constructors used in the ATerms of the program ASTs. Static semantics[1] can be implemented using either Name Binding Language 2 (NaBL2) or Statix. Both are meta-DSLs for specifying the static semantics, Statix is the successor of NaBL2. Languages can also have Stratego files. Stratego is a language for executing transformations on programs. Examples of transformations are optimizing a program, reducing it to a normal form and compiling to another language. Language projects include Editor Service (ESV) files that specify editor configuration settings like syntax highlighting (which is also autogenerated, but this allows a language developer to customize it) and editor actions.

Finally, a `metaborg.yaml` file denotes a directory as a language project. This file specifies project configurations like what SDF3 compiler to use. It also allows exporting parts of this language project and importing exported parts of other language projects. This makes language composition possible. For example, the PIE language project imports the Java grammar to compile the PIE DSL to Java.

2.2.4 Compilation with Stratego

Stratego is a meta-language to express program transformations. Examples are optimizations (`5 + x + 10 * x + 8` to `18 + 11 * x`) and compiling one language to another: `for (int i = 0; i<10; i++) {...}` to `for i in 0..9: ...`. Stratego uses rules to define transformations from one constructor to another. For example, the following rule combines two integer literals into a single value: `add-literal-ints: Plus(Int(x), Int(y)) -> Int(x + y)`. There can be multiple rules with the same name. Stratego will try them until one of them succeeds or all failed. This is like an implicit pattern match against constructors:

```
fold-constants: And(True(), e2) -> e2
fold-constants: And(False(), _) -> False()
fold-constants: And(e1, True()) -> e1
```

[1]Static semantics of a language refer to basic stuff like `"Hello"` is a string, adding two strings together yields another string, if you call a function that takes a single string then you should give it a single argument and that argument has to be a string, etc.

```
// No And(e1, False()) -> False() because that changes semantics, it won't execute e1
    anymore
fold-constants: If(True(), body_true, _) -> body_true
fold-constants: If(False(), _, body_false) -> body_false
```

The `fold-constants` rule above will perform constant folding on booleans and if statements where the condition is a constant. However, if we apply this rule to the term `If(And(True(), True()), Print("It's true"), Print("It's false"))`, it fails. The `And` constructors don't match because the top level constructor is `If`. And the `If` constructors don't match because the condition is neither `True()` nor `False()`, it is `And(...)`.

What we want instead, is to apply this rule to any matching subterm of the AST. To do this, Stratego has strategies. A strategy defines how to apply a rule to an AST. In this case, we can try to apply it to all subterms: `bottomup(try(fold-constants))`. This first applies the rule to the subterms to produce `If(True(), Print("It's true"), Print("It's false"))`, and then applies the rule to the term itself to produce the final result `Print("It's true")`.

2.2.5 Foreign functions in PIE DSL

The PIE DSL can interoperate with Java by calling Java methods. To call a Java method, it needs to be declared in the PIE program. A declaration looks like this:

```
func sign(x: int) -> int = foreign java java.lang.Math.signum
```

The part before the `=` is a normal PIE function declaration. It defines a function `sign` that takes an integer `x` and returns an integer. The part after the `=` specifies the function definition. `foreign java` designates this as a function that is declared in Java, and `java.lang.Math.signum` is the fully qualified name to the class and the function name. The definition does not need the Java parameters because the PIE parameters are mapped to Java parameters automatically.

2.3 Problems

todo: Should I talk about these problems in the present tense or past tense? ("The PIE DSL cannot express ..." vs. "The PIE DSL could not express ...") The problems with the PIE DSL fall into two categories. There are problems with the language design and problems with the language implementation.

2.3.1 Lack of scalability

A PIE program used to be written entirely in a single file, referring to other files was not possible. In Spoofax, we want one PIE file per language project, PIE is limited to a single language project at a time. In practice, 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. Finally, a project could just consist of multiple projects. An example is the Green-Marl compiler, which is one of the case studies (see todo: add reference to Green-Marl pipelines case study). In conclusion, we want a principled way to refer to other PIE files in order to use PIE in bigger projects with multiple languages.

2.3.2 Limitations in expressiveness

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

At the start of the thesis, the PIE framework used to throw exceptions to signal pipeline failure. Checked exceptions in Java hurt composability of tasks, because checked exceptions from dependencies either need to be declared or caught. Throwing exceptions to signal pipeline failure also has some semantic issues: exceptions are meant to signify an exceptional condition. Malformed input is not unexpected, so this should not lead to an exception. The PIE framework solved this 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 presents a problem for the PIE DSL: it now needs to support Result, and other generic library classes like Supplier<T>, Option<T>, <TODO: more examples?>. The PIE DSL does not support generics. Because foreign Java methods are declared in a PIE program with PIE DSL syntax, it is barely possible to interface with generic classes. The only thing that can be declared is an instance of the generic parameter. That is not enough for classes that are used with different generic arguments, such as these library classes. Additionally, it would be good if the PIE DSL could interoperate with arbitrary generic classes, not just generic classes from the PIE library.

The other new requirement from the DSL stems from a property of tasks: not everything that is required to execute a task is an argument. Sometimes values are hardcoded into the task[2]. 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. The PIE DSL did not have any way to specify such an instance variable.

We would like to extend the language with new syntax and semantics in order to handle these uses.

2.3.3 NaBL2 limits static semantics

As mentioned in the previous problem (2.3.2), the PIE DSL could not interoperate with generic classes in Java. The simplest solution is to add support for generic classes to PIE DSL as well. However, PIE DSL is implemented using NaBL2. NaBL2 does not have the power/expressiveness to allow implementing generics. Furthermore, NaBL2 has limited static error reporting and limited ways to provide debugging information in case of runtime errors, which makes development in NaBL2 tedious.

2.3.4 Compiler

The PIE DSL had two compilers. The first one compiled to Kotlin. Kotlin is a language that gets compiled to bytecode for the JVM, which means that it can interoperate with Java and thus with the PIE framework. Oracle didn't want to introduce Kotlin to their environment and asked us to compile directly to Java.

The second compiler compiled to Java using string interpolation. Using 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 don't get caught until Java compile time. The second issue is that it is inefficient when you want to do post-processing on the generated Java code. The generated Java string needs to be parsed before it can be used. Instead of compiling to a string and then parsing that string to a Java AST, it is more efficient to compile directly to a Java 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.

[2]In the actual code these values are not hardcoded but added during `TaskDef` construction using dependency injection

All this leads to two requirements for the compiler: it needs to compile to something that is already used in the Spoofax ecosystem (i.e. Java or bytecode) and it needs to compile to an AST.

Chapter 3

PIE Improvements

3.1 Compiler

todo: describe compiler to Java

3.2 Context paramaters

todo: describe context paramaters

3.3 Statix

Generics are a significant part of the Java implementation of PIE. During development of the PIE DSL, it was determined that interoperating between the PIE DSL with Java would be easier if the DSL also implemented generics. Originally, the static semantics for PIE were implemented in NaBL2, a DSL for static semantics from the Spoofax ecosystem. Unfortunately, NaBL2 is either not expressive enough to implement generics, or at the very least it is complicated enough that efforts to achieve that were abandoned. todo: describe attempt in NaBL2 - create new declaration for each instance of the generic class Instead, we decided to re-implement todo: better word than "re-implement" the static semantics in Statix, the successor of NaBL2. Statix is described in detail in <todo>.todo: add reference

Statix and NaBL2 operate mostly on the same high-level model: scope graphs and constraints. One of the major changes compared to NaBL2 is that scope graph construction and constraint solving are no longer two separate steps. Instead, constraints are solved when enough information is known that their result cannot change anymore.

3.3.1 Module system

Because Statix and NaBL2 use the same model, most of the changeover to Statix was fairly mechanical. One part that was not trivial is the module system in the PIE DSL. The implementation in NaBL2 did not implement the full specification, but simply used an import edge to import modules. todo: add an example? The implementation in Statix does implement the full specification. Partial imports todo: add code in combination with renaming make the implementation of the module system non-trivial. `renamed:sub:someFunc` should resolve, even if it is defined in a module `someModule:original_name:sub:someFunc`. This leads to the decision to create scopes for each submodule, and to declare submodules of the current module in the `mod` relation: todo: create figure

To declare the modules, the approach that matches the semantic model the most is a tree of submodules. The most straightforward way to create such a tree is adding modules one at a time by checking if the submodule already exists and creating it if it does not, in Statix code:

```
declareModule : scope * list(MODID)
declareModule(s, []).
declareModule(s, [name|names]) :-
  getOrCreateModuleScope(s, name) == s_mod,
  declareModule(s_mod, names). // not allowed: declareModule may try to extend s_mod, but we
       do not have permission to extend it.

getOrCreateModuleScope : scope * MODID -> scope
getOrCreateModuleScope(s, name) = s_mod:-
  query [...] |-> occs,
  getOrCreateModuleScope_1(s, occs, name) == s_mod.

  getOrCreateModuleScope_1 : scope * (path * (MODID * scope)) * MODID
  getOrCreateModuleScope_1(s, [(_, (_, s_mod))], _) = s_mod. // submodule was already
      declared.
  getOrCreateModuleScope_1(s, [], name) = s_mod :- // submodule not declared yet, declare a
      new one
    new s_mod,
    !mod[name, s_mod] in s.
```

This is not allowed due to Statix semantics: only scopes that were passed down as an argument to the current function or that were created in the current function can be extended.

Since the most straightforward approach does not work, there are a few alternatives to consider:

1. Create a representation of the full module tree before converting it to its representation in the scope graph.
2. Declare each module as its own linked list from the root scope.

The second solution means that each submodule can have multiple scopes associated with it. To resolve a qualified function, the query would need to run for each of the scopes, and the results would need to be merged manually. The number of queries grow linearly with the number of module scopes. Alternatively, we could define a temporary scope that has `I` edges to all module scopes so that we can run a single query. In this case, there is only one query, but the number of temporary scopes to point to all the module scopes grows linearly with the number of references to this module. Ideally, there is only constant growth: a single reference does not create scopes and runs a single query to resolve the reference. This means that I went for the first option: build a data structure that represents the module tree and instantiate that.

First of all, we want to create the representation of the module tree in the `projectOk` function. To do this, we first need a list of the modules to declare. This is not entirely trivial, because `projectOk` does not get a list of files. To pass the modules to `projectOk`, we declare each module in a dedicated relation `mod_wip` in `programOk`. This relation is then queried by `projectOk` to get the list of modules.

The data structure representing the module tree uses two constructors: `ModuleTreeRoot : list(scope) * list(ModuleTree) -> ModuleTree` and `ModuleTreeNode : MODID * list(scope) * list(scope) * list(ModuleTree) -> ModuleTree`. The `ModuleTreeRoot` contains a list of file scopes and a list of submodules. A `ModuleTreeNode` has the name of the submodule, a list of file scopes, a list of tree scopes representing the same submodule, and a list of submodules. The function `addToModuleTree` adds a `MODULE` to a moduleTree. A `MODULE` is a list of names and a scope. The sort `MODULE` has two constructors: `MODULE`, where the scope represents a file, and `SUBMODULE`, where it represents a module tree scope.

`AddToModuleTree` does to a `ModuleTree` what `declareModule` tried to do: check if the submodule already exists, adding it to that submodule if it did, or creating the submodule if it didn't. We do not run into the scope extension issue because it does not extend a scope, it just returns a new `ModuleTree`.

Finally, the moduleTree is instantiated. This is rather straightforward, the only thing to note is that each submodule in the tree creates a new scope, even if it already has a file scope. The reason for this is that a file scope cannot be extended, so it is not possible to add any submodules to the file scope. Instead, the newly created tree scope has a `FILE` edge to the file scope.

3.3.2 Generics

[I will start writing this section when I actually start on implementing generics] Reasons to implement generics: - Could not interop with generic classes - Workaround by making common library classes is ugly (also from an easthetic sense: we could not give methods on generic types the same syntax, so `Supplier.get()` did not work, we had to use `Supplier.get<>()`) todo: describe generics

Chapter 4

Evaluation

The goal of the PIE DSL is to reduce boilerplate and bugs. It should do this without sacrificing runtime performance or build time performance. From a design perspective, the PIE DSL should cover future use cases as much as possible, or be extendable so that new language features could be added to cover unforeseen use cases. This chapter evaluates whether the PIE DSL met these goals by applying it to three case studies and possibly some performance testing.

The first case study uses Tiger, which is a small functional language. This gives a clean example of how the PIE DSL can be used to parse, analyze and compile a Spoofax language.

The second case study applies the PIE DSL to database pipelines. Explaining what that means would just be a copy of the introduction for that section, so I'll just shut up now.

The last case study uses the PIE DSL to test language frontends at Oracle. These language frontends only define the syntax for a language, and are meant as compilation targets for PGX. As a sanity check, we would like to do reparse tests: parse an example program, pretty-print it, reparse the output of pretty-printing. This should lead to the same AST.

The last section of this chapter provides an analysis of the results. It looks at the main goals of reducing boilerplate and performance. It also discusses the objectives of the generality and extensibility of the language itself.

4.1 Case study: Tiger

Tiger is a small functional language introduced in Appel (1998).[1] It is used here as a clean example to apply PIE.

The goal in this use case is to run some sort of transformation.todo: what kind of transformation? (optimization: merge integers: 1 + 2 ==> 3) todo: Add example code showing input and output of transformation This requires parsing and analyzing the tiger program, running the transformation, pretty-printing the transformed program to a string, and finally writing the string to a file.

4.1.1 Pipeline implementation

Figure 4.1 shows the PIE DSL code for this case study. PIE tasks for parsing and analyzing are generated by Spoofax when building the language project. Writing to a file is a task in the PIE standard library. All that is left is writing a task that invokes the transformation and wiring everything up.

todo: Add code that is used to call the thing

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

Figure 4.1: PIE DSL code for the tiger case study

todo: Replace with actual code

```
module tiger:optimize

import std:writeToFile
import mb:metaborg:spoofax3:{IStrategoTerm, invokeStrategoStrategy as invoke}

func parse(program: supplier<string>) -> IStrategoTerm = foreign mb:metaborg:example:
    tiger:Parse
func analyze(ast: IStrategoTerm) -> (AnalysisResult) = foreign mb:metaborg:example:tiger:
    Analyze
func prettyprint(ast: IStrategoTerm) -> string = foreign mb:metaborg:example:tiger:
    PrettyPrint

// reads a tiger file, optimizes it, and writes the result back to the same file.
func main(file: path) -> path = {
  val ast = read file;
  val analysisResult = analyze(ast);
  val optimized = optimize(ast, analysisResult);
  writeToFile(file, prettyprint(optimized));
}

func optimize(ast: IStrategoTerm, analysis: AnalysisResult) -> IStrategoTerm = invoke("
    tiger-optimize-all")
```

Figure 4.2: Java code equivalent to the pie code in 4.1.

question: Put this here or add it as appendix? todo: Add Java code

4.1.2 Analysis

The equivalent Java code is given in figure 4.2. The PIE DSL code uses X lines of code (loc), while the equivalent Java code uses Y loc. todo: Fill in actual numbers

4.2 Case study: PGX Algorithms pipelines

Parallel Graph AnalytiX (PGX) is a toolkit for graph analysis, both for graph algorithms and SQL-like queries. It provides two DSLs, Green-Marl and PGX-A. Both can express graph algorithms. Green-Marl is a standalone DSL for expressing graph algorithms. PGX-A is an Embedded Domain Specific Language (EDSL) in Java, this makes integration with Java easier. Examples of graph algorithms are Dijkstra's algorithm for finding the shortest path[citation needed] and Kruskal's algorithm for finding the minimum spanning tree[citation needed] . Another example is the Adamic-Adar index of edges. Its implementation in Green-Marl and PGX-A can be seen in figure 4.3. The Adamic-Adar index can be used to predict edges between nodes in a social network. Depth-first search (DFS) and Breadth-first search (BFS) are considered such fundamental algorithms that they are built-in functions. The algorithms expressed in PGX-A are compiled to an implementation for either a single machine or shared memory PGX runtime.

The compilation goes as follows:

1. The Java code is parsed and analyzed with a Spoofax definition of Java
2. The Java code is transformed to Green-Marl.

Figure 4.3: An implementation of the Adamic-Adar index in Green-Marl and PGX-A

```java
/*
 * Copyright (C) 2013 - 2021 Oracle and/or its affiliates. All rights reserved.
 */
package oracle.pgx.algorithms;

import oracle.pgx.algorithm.EdgeProperty;
import oracle.pgx.algorithm.annotations.GraphAlgorithm;
import oracle.pgx.algorithm.PgxGraph;
import oracle.pgx.algorithm.PgxVertex;
import oracle.pgx.algorithm.annotations.Out;

import static java.lang.Math.log;

@GraphAlgorithm
public class AdamicAdar {
  public void adamicAdar(PgxGraph g, @Out EdgeProperty<Double> aa) {
    g.getEdges().forEach(e -> {
      PgxVertex src = e.sourceVertex();
      PgxVertex dst = e.destinationVertex();

      double value = src.getNeighbors()
         .filter(n -> n.hasEdgeFrom(dst))
         .sum(n -> 1 / log(n.getDegree()));

      aa.set(e, value);
    });
  }
}
```

```
/*
 * Copyright (C) 2013 - 2021 Oracle and/or its affiliates. All rights reserved.
 */

procedure adamic_adar(graph G; edgeProp<double> aa) {

  foreach (e: G.edges) {
    node src = e.fromNode();
    node dst = e.toNode();

    // In C++ backend, the compiler optimizes below
    e.aa = sum (n: src.nbrs) (n.hasEdgeFrom(dst)) {1 / log(n.numNbrs())};

    // into
    // e.aa = sum(n: src.commonNbrs(dst)) {1 / log(n.numNbrs())};
  }
}
```

3. The Green-Marl is analyzed.
4. The Green-Marl is compiled to C, C++, Java or Procedural Language for SQL (PL/SQL) depending on what the runtime requires.

This section - [done] Explain Green-Marl and PGX-A - [done] Add example code for both - Explain goal of this section: pipeline for PGX-A, executable as command line tool - Give results: PIE code, equivalent Java code, supporting code (Gradle, Java classes that can't be expressed in PIE, etc)

4.2.1 Introduction

todo: What are Green-Marl, PGX? What are we trying to do in this case study? -> Transform programs in some language to other languages.
Interesting feature: multiple language projects and backends in separate projects.

todo: write section

4.2.2 Pipeline implementation

todo: describe implementation

4.2.3 Analysis

todo: analysis todo: should include part about multiple language projects

4.3 Case study: testing pipelines

4.3.1 Introduction

What are we trying to do in this case study? -> Execute Parse and reparse tests (check that parsing and reparsing example programs works)

todo: write section

4.3.2 Pipeline implementation

todo: describe implementation

4.3.3 Analysis

todo: analysis

4.4 Analysis

Note: some/most/all of this probably needs to go to the introduction or problem analysis. It is currently here because I had to write it somewhere and I can't really make a good structure before I know what my goals and methodologies are.

4.4.1 User goal and tasks

Note: the user refers to the pipeline developer.
User use case: user needs to build some system and would like it to be precise, sound, incremental, etc. The user has decided to use PIE to implement this pipeline. We distinguish five categories of tasks related to writing pipeline code:

1. Implementing the initial pipeline. The user either has no pipeline at all or has an existing pipeline in some other language or framework but decided to change to PIE.

2. Changing or extending an existing pipeline. This refers to semantic changes to the pipeline: some pipeline steps need to be executed earlier, later, added or removed.

3. Maintaining an existing pipeline. This refers to changes to the code that do not change the conceptual pipeline. For example, there was a change in the api of a dependency, so now the pipeline code needs to be updated to the new api.

4. Reading the pipeline code to understand what it does, with no intention of changing it. This will happen a lot, definitely the most if we include reading of code during the other user tasks. If you want to change the pipeline in some way, you will likely need to read parts of it that do not need to be changed.

5. Debugging the pipeline. This is not a goal by itself, but arises as part of the other tasks. It is mentioned explicitly because it is important: being unable to properly debug the pipeline (even with just print statements) severely harms the user friendliness.

4.4.2 Ultimate goals of the PIE DSL and this thesis

Goal of the PIE DSL: make developing in PIE more user friendly by making it easier to write tasks. Goals of this thesis: extend / improve the PIE DSL system and verify that the changes did indeed make it easier to write tasks. We divide this into the following subgoals:

1. Increase readability. Readability is a nebulous concept, but it more or less means that we want to convey the necessary information/understanding in an as short a time as possible. "Necessary information" refers to the information that the user requires for their particular task. Note that this does not mean that we optimize for shorter code. Shorter code is only useful if it allows the user to read the code faster. Reading the code happens in all user tasks, and is central to task 4: "reading the pipeline code to understand it".

2. Increase the ease of writing. It should be easy and quick to write PIE DSL code. This can be achieved by making it specific to the domain of pipelines, by including types that are relevant to pipelines and by making common patterns easy to express. This is related to readability: if it is easy to read, it is likely not too long or complicated, so it would be easy to write as well.

3. Make it easy to reason about. This is the step between reading and writing when making a change: once the user has read the code and understands what it does, it should be easy to reason about so that it is easy to figure out what to do to make the desired changes. The model of the framework is quite elegant, so we use it for the DSL as well. This makes it easier for users if they need to implement something in Java directly if the task they need cannot be expressed in the DSL. It also makes it easier to keep feature parity between the DSL and framework. Because we closely follow the PIE model and that is completely (mostly?) done, this thesis does not provide major contributions to the model.

Afterthought: I think of it like this:

- readability: how easy is it to read some fragment (e.g. a single task) of code and understand what it does?

- reasoning about it: once you understand what each fragment of code does, how easy is it to figure out what the pipeline as a whole does? How easy is it to figure out what changes should be made to bring about some desired change in behavior? How easy is it to break up an understanding of what the pipeline should do into fragments?

- Writing: once you understand what fragments there should be and how the fragments should compose to do what you want, how easy is it to write down these fragments as code or to modify the existing fragments?

Reasoning about the pipeline is mostly influenced by the model defined by PIE. The reading and writing are where the DSL should bring improvements.

4.4.3 Measurable goals for this thesis

Our direct goals for the DSL are impossible to measure directly because they depend on the user. There are however instrumental goals that improve usability:

1. Increase the average amount of information that is visible on the screen (while keeping the readability of that information the same). We optimize for the amount of code visible on the screen, instead of the code that is contained within a line, because ultimately, the user reads the code from the screen. Many languages have boilerplate in the form of imports. Users often don't need to look at the imports to understand the code, so we don't care how much information is in those lines. We wouldn't care about the amount of lines imports take at all if it wouldn't take longer for users to scroll down to the code. (idea: put imports at the bottom of the file?)

2. Reduce the amount of duplicate expressions of information. A duplicate expression of information is two or more places that express the same information. Duplicate expressions of information are bad because they need to be kept in sync: if the information changes, all expressions of it need to change as well. Duplicate expressions of information include code duplication, which expresses the same logic multiple times, but there other ways information can be expressed multiple times as well. A good example are PIE TaskDef identifiers. These are used to identify a task to the PIE framework. In regular `TaskDefs`, these are always `getClass().getName()`, which returns the fully qualified name of the class. The obvious duplicate expression of information here is the duplicate code, which expresses the logic that all these classes use their qualified class name as their ID. However, even when class IDs are written out by hand, like `mb.tiger.spoofax.task.TigerShowParsedAst`, it still is a duplicate expression of the information "this class is named `TigerShowParsedAst` and is located in package `mb.tiger.spoofax.task`", which is also already defined by the folder structure and file name *and* the package statement and class name. Writing out the fully qualified ID explicitly duplicates the qualified name, but it also implicitly duplicated the logic that classes use their qualified name implicitly. Using the expression `getClass().getName()` expresses that logic explicitly, and avoids duplicating the qualified class name. Java unfortunately does not allow us to de-duplicate the logic, but this logic is expressed only once in the DSL and the user does not even need to know that task definitions have IDs.

3. Increase editor services like static error reporting, showing documentation, automated imports and reference following. PIE DSL compared to pure Java actually does the opposite: no documentation, no static checks and no following references from Java to PIE or vice versa. There is however more static error reporting when comparing the old DSL to the new DSL.

4. Increase expressiveness to increase applicability. Writing several tasks or helper functions and wrappers in Java does not improve user friendliness. Our main goal is to make it possible to express tasks at all, not necessarily to express things in multiple ways.

5. Increase applicability to various domains. PIE can't be user friendly if it doesn't work for your domain.

To evaluate whether the new PIE DSL meets these goals, we use the PIE DSL to express pipelines in various domains:

- Spoofax 3 language compiler pipeline. Compiles a language specification to a language implementation.
- PGX-A program pipeline. Compiles a program from Green-Marl to C, C++, PL/SQL.
- Code editor/IDE for Tiger. Various tasks for editor actions.
- Testing pipeline. A pipeline for testing the generated parser and prettyprinter of a language.
- Benchmarking. Benchmarking pipelines for various things. We only want to re-execute the benchmarks for elements that were changed.

These case studies will be evaluated on the following points:

1. Lines of code. We express each pipeline in the old DSL, the new DSL and Java. Because we express the exact same pipeline in all these languages, they each contain the same total amount of information. This means that we can use the lines of code to compare the amount of information per line in each language. This covers the amount of information that is visible on screen (sort of).

2. The amount of information that is duplicated. todo: How to test this?

3. The percentage of tasks that can be expressed in the DSL. This covers the expressiveness of the DSL. We will also compare this percentage between case studies to gauge how applicable the PIE DSL is across domains.

4. Performance of the PIE tasks. While we don't require that the generated code is as fast as handwritten code, it should not be more than half a second / a factor 3 slower (whichever is higher).

5. Build time of the pipeline. This is less important than the runtime, but having to wait a long time to test a pipeline is really annoying. Will be tested for both a clean build and with changes in a single task (incremental build).

We will also perform some evaluations outside the case studies.

1. Editor services will be analyzed from a theoretical perspective. This is not done as part of the case studies because this concerns dynamic errors during development, and I really do not feel like writing down every time I get a static error or follow a reference.

2. The time to write tasks is an important factor for the user friendliness, but is again something that cannot be tested with a completed pipeline. It is also hard to test objectively because experience from the other languages will likely improve the time of the last language. Instead, we will take some tasks that have already been implemented, read them, remove them (and the dependencies which are not used by other tasks), and then write them again to see how long that takes.

Table 4.4: Lines of code, number of characters and their ratios between PIE DSL code and the equivalent Java code.

	Lines of code			Number of characters		
Case study	Java	PIE DSL	ratio (%)	Java	PIE DSL	ratio (%)
Tiger	A	B	$C = B/A * 100\%$	U	V	$V/U * 100\%$
Database	D	E	$F = E/D * 100\%$	W	X	$X/W * 100\%$
Testing	G	H	$I = H/G * 100\%$	Y	Z	$Z/Y * 100\%$
Average	-	-	$(C + F + I)\%$	-	-	...

4.4.4 Boilerplate reduction

The conciseness of a language depends on two things: the information redundancy and code density. The information redundancy is a measure of how much information is repeated within a program. For example, in the Java statement `int x = 5;`, the type `int` is redundant, as it can already be derived from the expression. Redundancy should not be minimized entirely, because redundant information can serve to make programs more understandable and to make error checking easier. An example are function signatures. Often, the signature of a function can be derived from its body, but that takes time and may require very complex analysis. Having the type signature as part of the function signature allows developers and editors to get the type of a function without analyzing or even reading the function body. In conclusion: information redundancy should be balanced with understandability and analyzability of the code.

loc only captures the vertical length of a program. There is also the horizontal length, i.e. line length. Long lines take longer to read than short lines [citation needed].

Goal: compare both information density (how much information is put into a given amount of code) and code density (physical amount of code on screen).

While loc is not a metric that should be optimized when writing a program, it is useful to compare boilerplate of languages. For the number of characters, sequences of layout are counted as a single character, i.e. in the following example, the whitespace from '{' to 'p' is counted as a single character.

```
Class Foo {
  private final int someNumber;
```

The lines of code and number of characters for each case study are summarized in table 4.4. In each case study, the ratio of lines of code is below SomeNumber (where SomeNumber should be at most 50% or I have to check this text again), but I expect it to be around 25%), and the ratio of characters is around SomeOtherNumber (at most 40%, expected 25%). This discrepancy is due to overhead from having multiple files and actual Java boilerplate. Because each task needs to be a separate class and classes in Java are typically written one per file, we use two files to express the tasks that were not generated by Spoofax. This duplicates the package statement and some of the imports.

The `exec` method contains almost all necessary information about the task. The exception is the parameters in case there are more than one, these are defined in the `Input` class. todo: There also injected values that are not inputs for the task, but probably shouldn't mention them here The boilerplate comes from elements that repeat information. These elements are:

1. The class declaration repeats the class input and output type, and specifies that this is a PIE task.

2. The `Input` class specifies the parameters and their types, but then repeats them in the constructor (twice), the `equals`, `hashcode` and `toString` methods.

3. The fields and the constructor repeat the dependencies on other tasks (three times)

4. `getId` gives a unique identifier for the task. This can be derived from the task name.

4.4.5 Non-functional requirements

todo: discuss non-functional requirements

Chapter 5

Related work

Related work here.

Chapter 6

Conclusion

Conclusion here. (this will include future work)

Bibliography

Appel, Andrew W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press. isbn: 0-521-58388-8.

Hatch, William Gallard and Matthew Flatt (Nov. 2018). "Rash: From Reckless Interactions to Reliable Programs". In: *SIGPLAN Not.* 53.9, pp. 28–39. issn: 0362-1340. doi: `10.1145/3393934.3278129`. url: `https://doi.org/10.1145/3393934.3278129`.

"Introducing Ant" (2006). In: *Pro Apache Ant*. Berkeley, CA: Apress, pp. 1–9. isbn: 978-1-4302-0092-5. doi: `10.1007/978-1-4302-0092-5_1`. url: `https://doi.org/10.1007/978-1-4302-0092-5_1`.

Konat, Gabriël et al. (2018). "PIE: A Domain-Specific Language for Interactive Software Development Pipelines". In: *Programming* 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`.

Acronyms

AST	abstract syntax tree
BFS	Breadth-first search
DFS	Depth-first search
DSL	domain-specific language
EDSL	Embedded Domain Specific Language
ESV	Editor Service
IDE	Integrated Development Environment
loc	lines of code
NaBL2	Name Binding Language 2
OS	operating system
PGX	Parallel Graph AnalytiX
PGX-A	PGX Algorithm
PIE	Pipelines for Interactive Environments
PL/SQL	Procedural Language for SQL
SDF3	Syntax Definition Formalism 3

Appendix A

A

Appendix here.