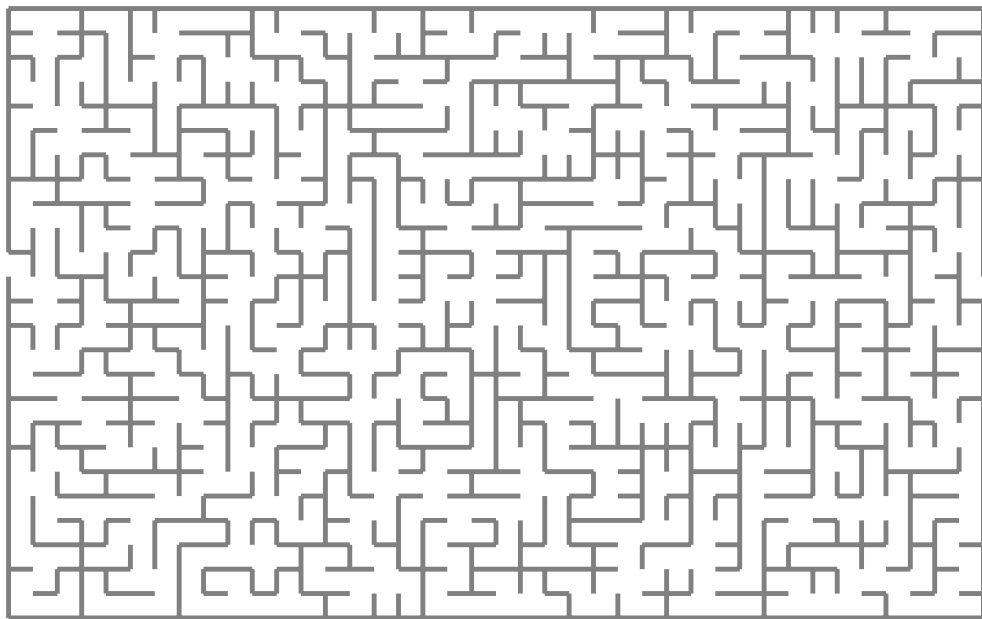


Master Thesis Template

Version of April 30, 2021



Jan Kees

Master Thesis Template

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jan Kees
born in Amsterdam, the Netherlands



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

© 2021 Jan Kees.

Cover picture: Random maze.

Master Thesis Template

Author: Jan Kees
Student id: 31415
Email: jan.kees@student.tudelft.nl

Abstract

Abstract here.

Thesis Committee:

Chair:	Prof. dr. C. Hair, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bee, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Dee, Faculty EEMCS, TU Delft
University Supervisor:	Ir. E. Ef, Faculty EEMCS, TU Delft

Preface

Preface here.

Jan Kees
Delft, the Netherlands
April 30, 2021

Contents





Preface	iii
Contents	v
List of Figures	vii
List of Tables	ix
0.1 Introduction	3
0.2 Problem analysis	6
1 PIE Improvements	11
1.1 Compiler	11
1.2 Context paramaters	11
1.3 Statix	11
2 Case study: Tiger	15
2.1 Introduction	15
2.2 Pipeline implementation	15
2.3 Analysis	15
3 Case study: database pipelines	17
3.1 Introduction	17
3.2 Pipeline implementation	17
3.3 Analysis	17
4 Case study: testing pipelines	19
4.1 Introduction	19
4.2 Pipeline implementation	19
4.3 Analysis	19
5 Related work	21
6 Conclusion	23
Bibliography	25
Acronyms	27
A A	29

List of Figures

List of Tables

Todo list

<input checked="" type="checkbox"/>	Using sections does not work well with this thesis template, it expects chapters as top level elements. Other students also use chapters. Do I really need to use sections?	3
<input checked="" type="checkbox"/>	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?	3
<input type="checkbox"/>	add reference	3
<input checked="" type="checkbox"/>	Which of these examples is better?	3
<input type="checkbox"/>	explain what precise dynamic dependencies are (and why we care?)	4
<input type="checkbox"/>	add reference	4
<input type="checkbox"/>	Does this follow from the evaluation?	4
<input type="checkbox"/>	did we do all of these?	5
<input checked="" type="checkbox"/>	Alternatively: "we make ... contributions", and "we provide an explanation ..." (follows the scalable incremental building paper literally). Which is better? . . .	5
<input type="checkbox"/>	add reference	6
<input checked="" type="checkbox"/>	not explained yet, refer to future section or use AST for now?	6
<input checked="" type="checkbox"/>	Do you know this?	7
<input type="checkbox"/>	general term for "compiler compiler"	7
<input checked="" type="checkbox"/>	Move to "Use case" above?	8
<input type="checkbox"/>	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 ...")	9
<input type="checkbox"/>	add reference to Green-Marl pipelines case study	9
<input type="checkbox"/>	describe compiler to Java	11
<input type="checkbox"/>	describe context paramaters	11
<input type="checkbox"/>	describe attempt in NaBL2 - create new declaration for each instance of the generic class	11
<input type="checkbox"/>	better word than "re-implement"	11
<input type="checkbox"/>	add reference	11
<input type="checkbox"/>	add an example?	11
<input type="checkbox"/>	add code	11
<input type="checkbox"/>	create figure	12
<input type="checkbox"/>	describe generics	13
<input type="checkbox"/>	write section	15
<input type="checkbox"/>	describe implementation	15
<input type="checkbox"/>	analysis	15
<input type="checkbox"/>	What are Green-Marl, PGX?	17
<input type="checkbox"/>	write section	17
<input type="checkbox"/>	describe implementation	17
<input type="checkbox"/>	analysis	17

 should include part about multiple language projects	17
 write section	19
 describe implementation	19
 analysis	19

0.1 Introduction

Using sections does not work well with this thesis template, it expects chapters as top level elements. Other students also use chapters. Do I really need to use sections?

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.

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.

add reference

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.

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 therefore parse only that file, and then run everything else for everything again, as the analysis depends on all files and therefore 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.

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.

add reference

verify this claim, back it up with some references, or make it more specific so that it is true To fill this gap, Pipelines for Interactive Environments (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 domain-specific language (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:

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.

Alternatively: "we make ... contributions", and "we provide an explanation ..." (follows the scalable incremental building paper literally). Which is better?

0.2 Problem analysis

0.2.1 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.

[add reference](#)

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 file

[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 rebuilt. 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

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 Spoofox, Spoofox 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.

0.2.2 Background information

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 Spoofox, 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"))`. Spoofox 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.

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

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.

Language projects in Spoofox

Move to “Use case” above?

A language in Spoofox 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 semantics¹ 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.

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
// 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

¹Static 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.

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")`.

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.

0.2.3 Problems

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.

Lack of scalability

A PIE program used to be written entirely in a single file, referring to other files was not possible. In Spoofox, 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

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.

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 task². 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.

NaBL2 limits static semantics

As mentioned in the previous problem (0.2.3), 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.

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.

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

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

Chapter 1

PIE Improvements

1.1 Compiler

describe compiler to Java

1.2 Context paramaters

describe context paramaters

1.3 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 Spoofox 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.

describe attempt in NaBL2 - create new declaration for each instance of the generic class

Instead, we decided to re-implement

better word than "re-implement"

the static semantics in Statix, the successor of NaBL2. Statix is described in detail in <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.

1.3.1 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.

add an example?

The implementation in Statix does implement the full specification. Partial imports

1. PIE Improvements

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:

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 1 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.

1.3.2 Generics

[I will start writing this section when I actually start on implementing generics]

describe generics

Chapter 2

Case study: Tiger

2.1 Introduction

What is Tiger? -> Toy language to test stuff like this

What are we trying to do in this case study? -> transform a Tiger program using Stratego

write section

2.2 Pipeline implementation

describe implementation

2.3 Analysis

analysis

Chapter 3

Case study: database pipelines

3.1 Introduction

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.

write section

3.2 Pipeline implementation

describe implementation

3.3 Analysis

analysis

should include part about multiple language projects

Chapter 4

Case study: testing pipelines

4.1 Introduction

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

write section

4.2 Pipeline implementation

describe implementation

4.3 Analysis

analysis

Chapter 5

Related work

Related work here.

Chapter 6

Conclusion

Conclusion here. (this will include future work)

Bibliography

- 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

DSL domain-specific language

ESV Editor Service

IDE Integrated Development Environment

NaBL2 Name Binding Language 2

OS operating system

PIE Pipelines for Interactive Environments

SDF3 Syntax Definition Formalism 3

Appendix A

A

Appendix here.