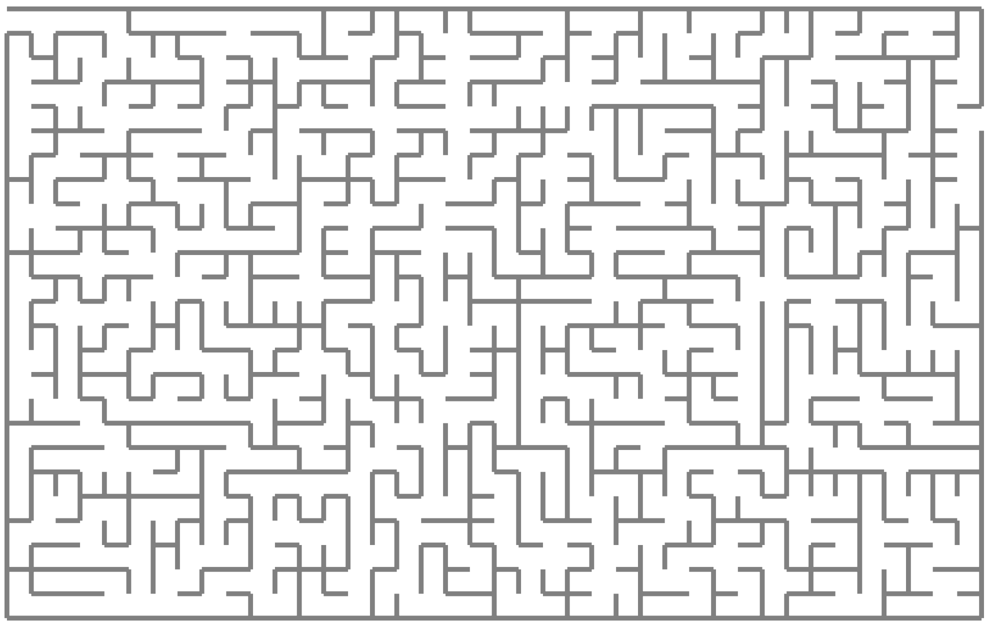
Master Thesis Template

Version of March 15, 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`

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
March 15, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Set editor to US English

Change Chapters to Sections

Transforming some inputs into some outputs is often not done with a single program, but with multiple programs that have to 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 command line script like a bash script or a bat script, but as the project grows such build scripts often become unmaintainable. Additionally, executing the full pipeline will take a long time, even if nothing changed. Build systems aim to keep build scripts maintainable by providing a concise way to express dependencies of and between tasks. 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.

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.

verify this claim, back it up with some references, or make it more specific so that it is true

PIE does support precise dynamic dependencies. 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. 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, e.g. projects that consist of multiple language projects.
3. The DSL is implemented in NaBL2, which has a few limitations in expressiveness.
4. The DSL has limitations in expressiveness
   a) No way to express core concepts of PIE, e.g. Suppliers and Results.
   b) No way to declare injected values for tasks.

We set out to improve the PIE DSL with the following objectives:

- Solve these problems
- 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 Spoofax pipelines.
- Since PIE is meant to give realtime feedback in Spoofax, it 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.

In the end, this thesis [has] the following contributions:

- a new compiler for the DSL
- implemented static semantics of the DSL in Statix
- Add modules, context parameters (and generics?) to the DSL
- Evaluate the new DSL in three case studies:
  - simple transformation from Tiger
  - database pipelines at Oracle
  - testing pipelines at Oracle
- Evaluation of performance for generated code (?)

The rest of this thesis is set up as follows. 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 with three case studies. It also compares expressing a pipeline in the DSL or directly in the Java framework. Section 5 lists related work.

slightly more descriptive: "puts this work into context by comparing it to solutions for similar problems"

Section 6 concludes this thesis.

Chapter 2

Problem analysis

TODO: In this chapter:

- Use case: Spoofax pipelines
  - What is language development?
  - What is Spoofax? (also explains the Spoofax Ecosystem, e.g. SDF3, NaBL2, Statix, Stratego)
  - What are pipelines that we want to express with the PIE DSL?
- More detailed explanations of the problems

PIE framework exists and is in active development. It is implemented in Java so there is lots of boilerplate.

To resolve this, the PIE DSL avoids the boilerplate and adds language support for domain specific elements, e.g. paths, list comprehensions. It still has room for improvement:

1. cannot compile to Java
2. cannot generate tasks with a dependency on something other than a task
3. cannot compose PIE files
4. cannot interface with Java generic classes

write chapter

use case. Summary: we want to run a pipeline in Spoofax, but we don't want to specify persistence and incrementality ad-hoc. This is what build systems do, but none of them are expressive and support precise dynamic dependencies

Pipelines for Interactive Environments (PIE) is a build system that aims to allow expressing pipelines with dynamic dependencies with automatic persistence and incremental execution. It achieves this by being imperative rather than declarative: tasks can call other tasks. PIE consists of the PIE framework and the PIE domain-specific language (DSL). The framework is an implementation of PIE in Java. The DSL is a language developed to create a concise specification of pipelines. The user can specify pipelines as functions that call each other. This PIE DSL code is then compiled to definitions in the Java framework by the PIE compiler. The Java framework saves the inputs and corresponding outputs of tasks. If a task is called with inputs that are cached, the corresponding output is returned without executing the task.

The Java framework keeps track of the task and file dependencies of tasks. When a file changes, it can use the dependency graph to keep track of things.

Set editor to US English

Language engineering is the discipline of designing and implementing programming languages. Spoofax is a language workbench, i.e. an IDE for designing and implementing programming languages. It brings tools that exist in regular software development to language engineering, like syntax highlighting, static analysis and a testing framework. It allows a language developer to work on the specification of a language and have example programs of that language open at the same time. This allows the language developer to get immediate feedback on their changes. Spoofax will compile the language specification and parse and analyse the example programs, which allows the language developer to experiment with language features. The goal of Spoofax is to abstract over the implementation details of implementing a language. A language developer only has to declaratively specify the language, Spoofax will generate an implementation for a parser, static analyser, syntax highlighting and other editor tools.

Generating these implementations uses a combination of external binary programs and Java functions. These programs and functions have dependencies on files and each other's outputs. This means that they form pipelines. Because Spoofax is an IDE, we would like it to give quick feedback when a language developer makes small changes to the code. However, executing the entire pipeline is really slow (in the order of minutes). This means that executing the entire pipeline is not an option. Instead, results are cached and recomputed when the output changes; this is called incrementality. In the beginning, this was done ad-hoc: every program and Java function would have its own implementation for incrementality and persistence

explain persistence before here

. This turned out to be hard to maintain: caching is already colloquially known to be one of the "fundamental problems of computer science", so reinventing the wheel each time leads to a lot of bugs.

The problem of incrementally executing programs and functions in the right order already has a solution: a build system. Unfortunately, existing build systems did not have all features required for Spoofax pipelines. In particular, many build systems do not support dynamic dependencies. Dynamic dependencies are dependencies that are not known until runtime. A well known example are C header files: it is impossible to know what header files a particular C file depends on until you parse it. The common solution is to overapproximate the dependency: make each C file depend on *every* header file. While this makes sure that every change is picked up on by the build system, it also means that every time a header file is changed, *all* C files are considered out of date, even if few or even none of those C files depend on that particular header.

To solve this, dependencies should be precise: by keeping precise track of dependencies for every task

The Spoofax ecosystem includes several DSLs, like SDF3 to specify the grammar of a language, NaBL2 and Statix to specify the static semantics, and Stratego to specify program transformations.

Can I assume that readers are familiar with language engineering in general? (i.e. they don't know Spoofax, but they do know what grammar, static semantics etc. are)

The specifications in these DSLs are combined into a language specification. This language specification is used by Spoofax to update open editors of the user language by parsing, analysing, syntax highlighting and showing errors. It can also be packaged into a plugin that does these things so that the user language can be used independently from Spoofax.

All of these tasks have dependencies on the specification files, other files, each other and other tasks. Executing all tasks each time anything changes takes too long and is often not necessary: if an example program is updated, the language specification stays the same, so the language does not need to be rebuild. Only the editor for that example program (and possibly other editors if they depend on this one) need to be reparsed and re-analysed. The solution is incrementality: only execute the tasks with changed inputs. If the input stays the same, return the output from cache.

Additionally, we don't want to rebuild the language each time we restart the IDE.

PIE is a build system that aims to allow expressing pipelines with dynamic dependencies with automatic persistence and incremental execution. It achieves this by being imperative rather than declarative: tasks can call other tasks. PIE consists of the PIE framework and the PIE DSL. The framework is an implementation of PIE in Java. The DSL is a language developed to create a concise specification of pipelines. The user can specify pipelines as functions that call each other. This PIE DSL code is then compiled to definitions in the Java framework by the PIE compiler. The Java framework saves the inputs and corresponding outputs of tasks. If a task is called with inputs that are cached, the corresponding output is returned without executing the task.

The Java framework keeps track of the task and file dependencies of tasks. When a file changes, it can use the dependency graph to keep track of things.

The PIE DSL is a language for expressing pipelines. Pipelines are expressed as functions that call other functions. While the DSL and the compiler did work, they also had several shortcomings:

1. Existing compilers were not maintainable.
2. Does not scale to larger projects, e.g. projects that consist of multiple language projects.
3. Implemented in NaBL2, which has a few limitations in expressiveness.
4. Limitations in expressiveness of the PIE DSL.
    a) No way to express core concepts of PIE, e.g. Suppliers and Results.
    b) No way to declare injected values for tasks.

Chapter 3

PIE Improvements

3.1 Compiler

describe compiler to Java

3.2 Context paramaters

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, Name Binding Language 2 (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.

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.

add an example?

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

describe generics

Chapter 4

Case study: Tiger

4.1 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

4.2 Pipeline implementation

describe implementation

4.3 Analysis

analysis

Chapter 5

Case study: database pipelines

5.1 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

5.2 Pipeline implementation

describe implementation

5.3 Analysis

analysis

should include part about multiple language projects

Chapter 6

Case study: testing pipelines

6.1 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

6.2 Pipeline implementation

describe implementation

6.3 Analysis

analysis

Chapter 7

Related work

Related work here.

Chapter 8

Conclusion

Conclusion here. (this will include future work)

Acronyms

AST	abstract syntax tree
DSL	domain-specific language
NaBL2	Name Binding Language 2
PIE	Pipelines for Interactive Environments

Appendix A

A

Appendix here.