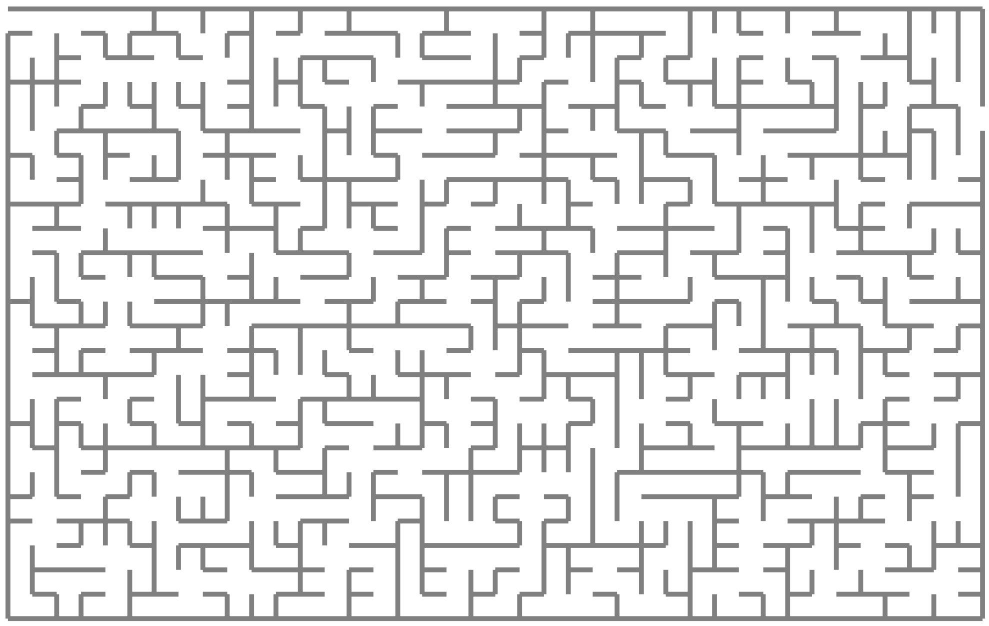
Master Thesis Template

Version of February 23, 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

TUDelft

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.

Preface

Preface here.

Jan Kees
Delft, the Netherlands
February 23, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

short introduction: what is PIE?

Using the PIE framework requires a lot of boilerplate. To mediate this, the PIE DSL is a DSL to specify PIE pipelines. It compiles to Java. The PIE DSL was created by Gabriël Konat in

reference

as part of his work on the PIE framework. This included a compiler from PIE DSL to Kotlin. For various reasons compiling to Kotlin was deemed undesirable, so I implemented a compiler from the DSL to Java

is there a reference? If not: add footnote to explain why

.

The PIE DSL was not [powerful | expressive | extensive] enough to be used in many real world use cases. The Java compiler implemented used string interpolation, which (as it turned out) is "suboptimal" [1].

Problems:

1. Existing compilers are not satisfactory
   - Compiler to Kotlin is outdated, compiling to Kotlin is undesirable.
   - Compiler to Java compiles via string interpolation, which is very limiting.
2. Single file only (does not scale to multi-language project setups)
3. Expressiveness
   a) No way to express core concepts of PIE, e.g. Suppliers and Results
   b) No way to declare injected values for tasks

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.

1. Implement a compiler that compiles to Java using Java ASTs

[1]it's severely limiting, don't do it

2. Implement a module system
3. Extend the old PIE DSL with new constructs
    a) Add generics
    b) (If adding generics fails) Add `supplier<T>` construct
    c) Add context parameters construct

Contributions:

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

Chapter 2

Problem analysis

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

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 Pipelines for Interactive Environments (PIE). During development of the PIE domain-specific language (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, Waar staat NaBL2 voor :'D (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.