CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Using semantic analysis to assist schedule optimization

Master's thesis in Computer Science – Algorithms, Languages & Logic

JACOB RIPPE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Using semantic analysis to assist schedule optimization

JACOB RIPPE

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Using semantic analysis to assist schedule optimization
JACOB RIPPE

© JACOB RIPPE, 2018.

Supervisor: Krasimir Angelov, Department of Computer Science & Engineering
Advisors: Fredrik Altenstedt and Thomas Johnsson, Jeppesen Systems AB
Examiner: K. V. S. Prasad, Department of Computer Science & Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Using semantic analysis to assist schedule optimization
JACOB RIPPE
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Airline Crew Planning is a complex task often divided into many steps to reduce complexity. One approach for finding good solutions is to generate a large set of solutions and determine which combination of solutions satisfy all constraints at the lowest cost.
We have analyzed a software system for generating solutions that uses rules stated in a DSL to determine solution legality. The legality is checked during solution generation, and the legality of the solution in progress affect the generation strategy. The question this project sought to answer is whether it is possible to determine certain properties of the rules at compile-time. The approach has been to analyze the structure of the rules and assign semantic information in a method inspired by type inference rules.
In an iterative fashion, theories about the system were formed, and tested by implementing checks inside the DSL compiler. After manual verification of the theories by looking at the output of the implementation, the theories were expanded upon or revised.
The implementation has been able to automatically identify a few rules with the desired properties, and more may be identified with continued development. Inference rules were a general approach to deciding properties, and more interesting findings might be found using more advanced techniques. However, there is a limit to what can be decided by static analysis.
Aside from analyzing existing rules, some semantic information may be of assistance when writing new rules, and might even be of assistance at runtime in future projects.

Keywords: computer science, semantic analysis, static analysis, compilers, domain specific languages

Acknowledgements

I am very grateful to have been given this opportunity to do my thesis work at Jeppesen at their Gothenburg office. Jeppesen has kindly provided me with everything I've needed to make this thesis a possibility.
I want to thank Fredrik Altenstedt and Thomas Johnsson for agreeing to be my technical advisors and provide insight, technical assistance, feedback and many interesting discussions about the project.
Thanks to everyone at Jeppesen who have shown interest and cheered me on.
Thanks to Andreas Westerlund, who made this project a reality by introducing me to my advisors.
Thanks to Kristoffer Ptasinski for taking care of the administrative details of my work at Jeppesen.
Thanks to Tomas Gustafsson for assisting with the thesis writing.
Thanks to Michael Spicar for letting me stay in his office and allowing me to claim all whiteboard space for the past few months.

I want to thank my supervisor Krasimir Angelov and my examiner K. V. S. Prasad for helping me shaping this project into a reality, and for valuable feedback on my thesis writing.
I want to say thank you to my family for all the support, and a big thank you to Anna Norén for always being there for me.

Jacob Rippe, Gothenburg, February 2018

Contents

List of Figures

Glossary

AST Abstract Syntax Tree, tree representation of code
DSL Domain Specific Language, a specialized language in a restricted domain
Expression Instruction in a programming language
Leg A flight between two airports
Duty Part of a trip, a sequence of flights in the same working day
Trip A sequence of flights, typically starting and ending in the same airport
Crew Pairing Part of Crew Planning, A procedure with the goal of finding a legal set of trips for an anonymous crew member
Crew Rostering Part of Crew Planning, A procedure where crew members are assigned to trips so that all flights are staffed
Deadhead Leg where a crew member is travelling as a passenger instead of working

1

Introduction

Programming languages and their usage evolve over time. One goal when developing a language further is to make it easier for a programmer to express what they want to achieve.

Domain Specific Languages (DSLs) are a sort of programming languages that are designed to be especially expressive in their own domain but with limited use outside of that domain, Mernik et al. [24]. This enables a programmer to model problems without needing to think about the low-level implementation details. Keshishzadeh and Mooij have discussed semantic analysis of DSLs [19], and DSL design and implementation is described in detail by Voelter et al. [29].

The company Jeppesen supplies Airline Optimization Solutions, one being crew schedule optimization. To model this domain, they have developed their own DSL, **Rave** (Rule And Value Evaluator) [2, 3, 21]. Kasirzadeh et al. describes some challenges of crew scheduling in general [18], and Goumopoulos & Housos [15] discusses another DSL and system for trip generation in Crew Pairing.

Rave is used to formally express rules and costs associated with scheduling that can be used in their optimizing system. To deal with the massive amount of regulations in Airline - such as government rules, union agreements, company policies, and so on - these regulations are implemented as Rave code and compiled into rulesets that can be evaluated when scheduling.

This project will study Rave in an effort to automate the detection of patterns in the code that can lead to inefficiencies.

1.1 Project Aim

This project aims to develop a method for automatically classifying different types of rules expressed in the DSL Rave, specifically detecting if a rule is a so called Illegal Subchain rule or not. Currently, it is up to the programmer to manually classify these rules, which puts an extra burden on them to have a grasp of the scheduling process. Improper classifications of rules are not always easy to find, and can lead to performance issues or worse solutions.

The main theory that this project aims to verify is that it is possible to classify these rules automatically at compile time.

1.2 Problem Description

Here is a description of the system in use and the problem that this project intended to solve. A more thorough description of the system is described by Karisch & Kohl [21]. Other related sources: Hjorring & Hansen [16], Andersson et al. [3].

1.2.1 Crew Planning

The Crew Planning process takes place after a timetable for flights have been constructed. Its goal is to assign crews to all flights with a total trip cost as low as possible. Crew Planning is typically split up into two major phases in order to reduce the complexity of the task, Crew Pairing and Crew Rostering [2, 3, 18, 21]:

- Crew Pairing deals with finding a set of legal trips that together cover the crew needs for all flights.
- Crew Rostering assigns actual crew members to the trips using the trips that were generated in Crew Pairing.

For this project we focus on the Pairing Generator, and the rulesets that are used to check trip legality. A trip is a sequence of flights that start and end in the same base.
The Jeppesen Crew Pairing optimizer has two major components, the generator and the optimizer. First, the generator generates a large number of potential trips that are legal for some unspecified crew member. Then, the optimizer finds a subset of these trips that cover all crew needs for all flights.

1.2.2 The Pairing Generator

Somewhat simplified, the generator has a trip in progress that it tries to add flights to. When adding a flight, the generator checks with the rule system if the trip along with the new flight is legal according to the rules.

- If the trip is legal, the trip so far is saved as a potential complete trip, and the generator tries to add another flight.
- If the trip is illegal, the generator removes the last flight and tries adding another flight.

This is how the process works for most of the rules, but there exist some types of rules that complicate the matter. These types of rules are called Illegal Subchain rules, and they need to be handled differently by the generator.

1.2.3 The Rule System and Illegal Subchain Rules

The rule system consist of regulations, which are made available to the system by the Rave language. The regulations are typically defined on complete trips; when checking a trip to see if it is legal, the rule system returns a true or false, if it is legal or illegal respectively. However, when generating solutions, many incomplete trips will be checked against the rule system, which adds some complexity to the problem.

In order to check legality of incomplete trips by the generator, the concept of Illegal Subchains was created.
An Illegal Subchain Rule is a rule that when given a complete trip may be evaluated to true, but it may evaluate some, or all, sub-trips to false. Example:
A rule stating that a trip must start and end in the same airport.
When building a trip satisfying this rule, the generator starts with one flight, and checks if it is legal. Assuming that there are no scheduled flights that start and end in the same airport, there is no flight that alone can satisfy the rule.
Thus, without some changes in strategy, the generator would find no legal trips when using this rule. To circumvent this, it is possible to state that a rule should only be evaluated when certain conditions are met.
For a complete trip to be legal it still has to satisfy all rules in the ruleset, but conditional rule checking can make sure that Illegal Subchain Rules do not interfere with the generation of otherwise legal trips.

1.2.4 Rule Classification

A distinction is made between *Illegal Subchain* Rules and *Final* Rules, and all rules can be classified as one of the two. An informal way of describing them is this:
If a trip is tested against a rule when generating and the rule is violated, there are two scenarios:

- The rule is an Illegal Subchain Rule: Then the trip may become legal by continuing adding flights to the trip.
- The rule is a Final Rule: Then this sub-trip is illegal, and there can exist no legal trip that can be constructed by only adding flights to the current trip.

Basically, Illegal Subchain Rules need to be satisfied in order for a complete trip to be legal, but a violated Illegal Subchain Rule should not exclude any legal solution during the generation process.
As mentioned earlier, it is up to the programmer to manually classify rules. The goal of this project is to analyze the rules that the programmer writes and determine whether they are Illegal Subchain Rules or not.
Given some domain knowledge and insight on how the generator works together with the rule system, the circumstances under which Illegal Subchain rules emerge will be examined in order to develop a method to classify them automatically.

1.3 Limitations

This project is intended as a proof of concept for possible future development of optimization of processes using the Rave language. As such it is somewhat experimental, and its main focus is studying the possibility to improve the existing system with semantic properties to reason about how the evaluation of expressions changes during Pairing Generation.
Rave is used to model many things in the system, but this project is only interested in how it is used during the generation phase of Crew Pairing. Therefore, only a subset of the language is studied, and some of the theories will only apply in this con-

text. The assumptions and simplifications of the system are described in Section 2.3.

The project only implements some static analysis of some existing code, but some other approaches are considered as well.

2
Theory

In the following section, more about the theory is discussed.

2.1 Background and Related work

Compilers and Programming languages are highly relevant to this project. Some material used as reference include Compilers: Principles, Techniques, and Tools [1], Implementing Programming Languages [26], and Compiler Design: Syntactic and Semantic Analysis [30]. Compilation and compiler development is a broad subject that covers many subproblems, but this project mainly relates to type checking and semantic analysis.
Type systems and inference rules are covered in depth by Pierce [25].
An important concept is Abstract Syntax Trees (ASTs), which are a way to represent the expressions in a programming language. These are the target of analysis.
The work done in this project is a form of static analysis, i.e. the analysis is done without running the program. Abstract Interpretation is a method of reasoning about the possible outcomes of code, researched in depth by Cousot & Cousot [11–13]. Bruynooghe [6] has applied some Abstract Interpretation on logic programming, which is a kind of declarative programming language.
To heighten the confidence in code correctness and to make it easier to write good code, methods for software verification and validation is an active area of research. Hovemeyer & Pugh [17] have written an influential paper on finding patterns in code that may be erroneous. Their experiences of implementing the detection of what they refer to as "bug patterns" was very positive. Basically, with the knowledge of the syntactical structure of a bad pattern in program code, it is possible to find similar errors with static analysis. They also show some examples how trivial mistakes can be common in large projects, and how effective rule based error detection can be. Another important thing mentioned is how mis-interpretation of semantics is a common source of faulty code.
Flanagan et al. [14] describe their experiences of "extended static checking", where they have implemented an additional layer of formal reasoning on top of Java to ensure correctness of the code. Among other things they extend the syntax to enable the programmer to formally specify properties that are checked by interpreting the code.
Chin, Markstrum and Millstein [8, 9] have presented some experiences with adding "semantic type qualifiers" to an existing language, with subtyping of variables and enforced additional checking of these new types. This has been an inspiration for

the *Attributes* that were developed during this project.
Solodkyy et al. [27] have also written about extending an existing language with additional type checking, and describe how these additional checks can be done without modifying the compiler.
Reasoning about semantics in a specific domain can be used to ensure correctness and may make certain kinds of optimizations possible.
Some related papers on this subject are written by Mooij & Keshishznadeh [19] who reason about DSL correctness through formal semantics, and Cire et al. [10] who describe optimization of operations in a domain model.

2.2 Rave Language Overview

This section contains a short overview of the Rave language parts that are relevant to this project and how they behave during Pairing Generation.

2.2.1 About the language

Rave is a declarative DSL used to model cost and legality in Jeppesen's Crew & Ops management products. The language is designed so that the compiled code can be executed by external applications. It is the external application that supplies the input data that can be referenced in the Rave code. For example, the start time of a flight can be referenced in Rave code, but it has no concrete value until it is evaluated in a context where flight data exists.
This modular separation of concerns means that a user may make adjustments to the rules, the external applications, and the input data separately [2, 16]. The external application in the context of this project is the Pairing Generator, which supplies the trips that the rules are checked against.
The lack of data during compile-time poses some interesting challenges to this project, since it is not safe to make assumptions about variables in the Rave code. However, from how the Pairing Generator uses the Rave code and the data it supplies, some properties and relationships between data are known.
As a side note, the declarative nature of Rave is convenient from an analysis-standpoint because it is intuitive to write and read the code, and the structure of the computations are stated rather clearly.

2.2.2 The components of Rave code

Here are a few of the more important components in Rave code referenced in this project. Two central kinds of definitions are **Rule**- and **Variable** definitions. They both have a name and are defined in terms of **Expressions** - which are described in more detail in Section 2.2.3. Expressions are the subject of analysis in this project. Two other important aspects are **Keywords** and **Levels**.
Variable definitions are expressions that have a name, which can be referenced by other expressions. The name is surrounded by %-characters and the defining expression is stated after the =-character. Example variable definitions:

```
%one% = 1 ;

%five% = 4 + %one% ;
```

Keywords are like a special kind of variable that is used to refer to external data, such as flight times. This data is supplied by the application running the Rave code; in this case it is the Pairing Generator. The keywords are not assigned values in the Rave code, but they can be referenced in Rave expressions.
Rules are defined in terms of boolean expressions that describe the legality of input data. Rule definitions start with the reserved word `rule` followed by the rule's name and the =-character. There exist more constructs to express more complex rules but, for the scope of this project, what's interesting is that rules have a defining expression. Optionally a **valid case** can be added before the defining expression, to decide when a rule should be evaluated. The end of a rule definition is marked by the reserved word `end`. Rules and legality are described in more detail in Section 2.2.5.
Example rules:

```
rule rulename =
  %rule_expression% ;
end

rule cond_rule =
  valid %rule_condition% ;
  %rule_expression% ;
end
```

Levels are a way to partition the trip in progress into smaller parts.
Using levels it is possible to write expressions that refer to "all legs within a working day", "all legs in the current trip", or "all working days in the current trip". New levels can be defined in Rave code, but this project will focus on three standard types: **Trip**, **Duty**, **Leg**. Leg refers to a single flight, duty refers to the flights of a working day and trip refers to all legs. Legs are automatically grouped into level instances according to the level's definition. More on this in Section 2.2.4
Figure 2.1 shows an example of a trip with 6 legs, partitioned into 2 duties.
An example statement: The total flight time of a duty can be described as the sum of flight times of all legs in that duty.

2.2.3 Expressions in Rave

An expression is a construct for describing how to compute a value. Expressions can be built up from multiple subexpressions. The following section will describe some of the expression types.

Variable and **Keyword references** are ways to reference the expressions of other named entities. Their value is the evaluation of the expression that they refer to. Examples:

- `%variable%`
- `leg.%start_utc% ;`

Mathematical operations are operations such as addition, subtraction, multiplication, division and modulo. The operands are expressions as well, and are con-

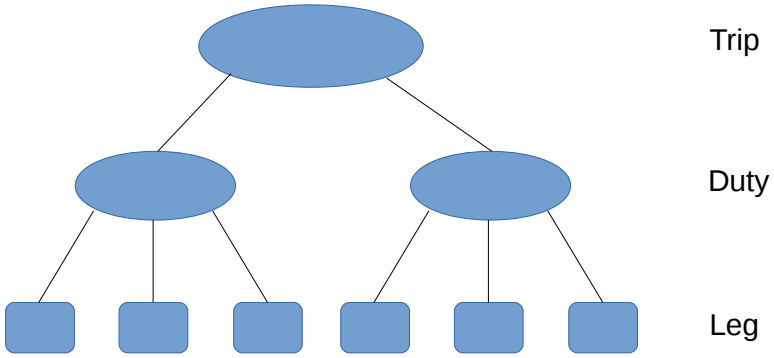


Figure 2.1: Illustration of some common levels in Rave. A trip consists of duties, which consist of legs. The underlying datastructure can be regarded as a list of legs in chronological order (a trip) with indices that denote where the different level instances start and end.

sidered subexpressions to the operation. The value of the operations follow the algebraic laws and is usually an integer. Examples:

- `5 + 5 ;`
- `-1 ;`
- `100 * 100 ;`

Comparisons are related to the mathematical operations, but they state the relationship of two expressions in relation to an operator. The operators are Equality(=), Nonequality(<>), Greater than (>), Less than (<), Greater or equal(>=), and Less or equal(<=). The value of the operations state whether the comparison is true of false. Examples:

- `1 > 2 ;`
- `%sum% <= %limit% ;`
- `True <> False ;`

Value constants are expressions that are simply defined to hold some value, which doesn't change during runtime. **Parameters** are related to constants, as they are defined in terms of a constant value. The difference is that the value of a parameter may be changed at runtime, before generation. Optionally, parameters can have a minimum and/or a maximum value limit(s) that limit the possible value range of the parameter. Examples:

- `50 ;`
- `True ;`
- `"Hello" ;`
- `%bounded_param% = parameter 7 minvalue 5 maxvalue 12 ;`

The last example defines the parameter `%bounded_param%`, with a default value of 7. This may be changed at runtime, but it can only be set to a value within the range [5,12].

Boolean operations such as **and**, **or**, and **not** are used to combine boolean expressions into more complex boolean expressions. Examples:

- `True or False ;`
- `not True ;`
- `(5 > 3) and %boolean_variable% ;`

Conditionals are the standard `if` *a* `then` *b* `else` *c* constructs as seen in most programming languages used when the resulting value depends on some condition. The value of the expression is the evaluation of *b* or *c*, if *a* evaluates to true or false respectively. Example:

```
if %condition%
  then 20
  else 25 ;
```

This expression evaluates to 20 if `%condition%` is true, and 25 if it is false.

Traversers are used to evaluate expressions on one or more level instances. They can be used to refer to multiple level instances at once or to relate to a specific instance, and there are many kinds that compute different things. Some central ones are **sum**, **count**, **next**, **prev**, **first**, **last**, **any**, and **all**.
The evaluation of a traverser expression may vary depending on what the trip in progress consist of, so they can be used to express properties of specific level instances.
Traversers make up the functionality that is made possible by loops and recursion in many other programming languages and as such, Rave has no other concepts of loops.

- `sum(leg(duty), leg.%time%)` evaluates the sum of the leg-expression `%time%` for all legs in a duty. This is one way to define the total flight time of a duty.
- `count(leg(duty))` counts the number of instances of legs in a duty. Equivalent to `sum(leg(duty), 1)`.
- `prev(duty(trip), duty.%time%)` evaluates the duty-expression `%time%` of the duty before the current duty.
- `last(leg(duty), leg.%end_utc%)` evaluates the leg-expression `%end_utc%` for the last leg in the current duty. This is one way to define the time of the last landing in a duty.
- `any(leg(duty), leg.%is_international%) ;` evaluates the leg expression `%is_international%` for each leg in a duty. The expression is true if any of the evaluations is true.

2.2.4 Instances of Levels and Expressions

Expressions belong to a specific level, which is decided depending on how the expression is defined. The expression level dictates in which context it gets evaluated as well as how often it may change in value. When a new instance of a level object is created, then each expression which belongs to that level also has a new instance created.
For example, each leg has an instance of the `leg.%time%` expression which tells the flight duration time because the expression is defined on a per-leg basis. Each duty has a `duty.%time%` expression which tells how long the working day for that duty is. The duty time is defined in terms of the legs in that duty instance, so the `duty.%time%` instance changes when new legs are added.

Which instance of a level that a new leg belongs to is handled by checking the new leg against each level's defining expression.
An important distinction should be made to avoid ambiguity when generating trips. When regarding level instances, such as *duty*, generation may either result in a continued instance or a new instance of that level.
Definition 2.2.1. New Instance: When the generator adds a leg that, according to a level's definition, can not be a part of an existing level instance, then a **new instance** of that level is created. All expressions belonging to that level have a new expression instance created and evaluated.
Definition 2.2.2. Continued Instance: When the generator adds a leg that can be part of an existing level instance, the leg is added to the level instance. The level instance containing the new leg is then considered a **continued instance**, or **continuation** of the previous instance without the leg. All expression instances which are affected by the added leg get evaluated to a new value.
Example: The generator tries to add a new leg. The starting time of the new leg is 12 hours later than the last leg of the last duty. According to duty's level definition, the new leg should be considered part of a new duty, so a new duty instance is created containing the new leg.
As elaborated on in Section 2.2.5, rules are expressions - they too have levels and instances. Furthermore, for a rule to be considered valid, then all existing instances of that rule must be valid. Because of this, any rule describing legality on, for example, "a duty" will be checked for all duty instances, relieving the rule writer from thinking about *when* to apply rules.
Observation 2.2.1. The fact that expressions have multiple instances means that any property that is proven about the expression is true for all instances of the expressions. Conversely, in order to prove something about an expression, it has to be true for all contexts.
Note: Many expressions only affect one instance of a level object, but it is also possible to depend on other instances of the same level. For example, the rest time before the beginning of a new duty is a value specific to one duty instance, but it depends on two duties in direct succession.
This dynamic makes it possible for later instances to trigger a violation of an earlier rule instance. However, no matter which instance the rule is violated for, it is the last added leg that is of interest when deciding whether there may exist legal continuations. This is true because of the order that solutions are found in, which is elaborated on in Section 2.3.

2.2.5 Rules and Legality

On a structural level, the legality of a rule in Rave is determined by it's defining expression, e_r. e_r is a boolean expression which is true when the rule is satisfied by a trip and false when the rule is violated.
The central concept to classifying rules in this project is to detect properties of e_r which can describe how legality changes in relation to generating trips.
Since e_r is a boolean expression, some of the most important properties for classification are those that describe how trip generation affect boolean expressions. Central

to this report are the comparison-expressions.

2.2.5.1 Valid case

The Valid case is an optional additional boolean rule expression, e_v, that describe whether e_r should be evaluated or not during generation. If e_v is evaluated to true, the rule legality is decided by e_r. If e_v is evaluated to false, the Generator keeps generating regardless if e_r is true or false. Omitting a e_v expression is equivalent to having e_v defined as true.
There are a few cases where one would expect a Valid case to be used:

- To tell that "This rule is only applicable in certain cases, namely under precondition e_v". In this case the rule's legality is decided by the logical implication $e_v \rightarrow e_r$, i.e. the rule is valid unless e_v evaluates to true and e_r evaluates to false.
- To tell that "This node may be illegal according to e_r, but there may exist legal continuations".

Combinations of these cases may also exist. The difference between them relate to how they are handled in the Pairing Generator. This is elaborated on in Section 2.3.4.

2.3 Model and Domain

Here is a description of the problem domain and how it is modelled in this project in order to reason about it. Note that some assumptions and some simplifications are made in order to reason about the system more easily, but the findings of this project are still applicable to the full system.

2.3.1 Domain to model

Flight regulations are given to the **rule writer**, a programmer who translate the regulations into **Rave code** that describe them in a formal context. The rave code is compiled by the **compiler** into the **rule system** that the **Pairing Generator** can use to check the legality of **trips**. Figure 2.2 illustrates how the different parts in the system work together.

The Pairing Generator wants to find a set of trips that are legal for an anonymous crew member. As input data it has the **flight schedule** for aircrafts, which can be regarded as a list of flights (referred to as "legs"), sorted by departure time. The legs have many properties, some which are data that get supplied at runtime, and some that are expressions computed from this data. This project and its implementation will put special emphasis on starting time (*departure*) and ending time (*arrival*) of flights, properties which affect the legality of many rules. Other important properties of a flight include starting and ending airport.
To illustrate the set of possible solutions (trips), the search space is introduced.
Definition 2.3.1. Search Space: The search space of the Generator is a tree structure whose nodes represent all possible trips of a flight schedule.

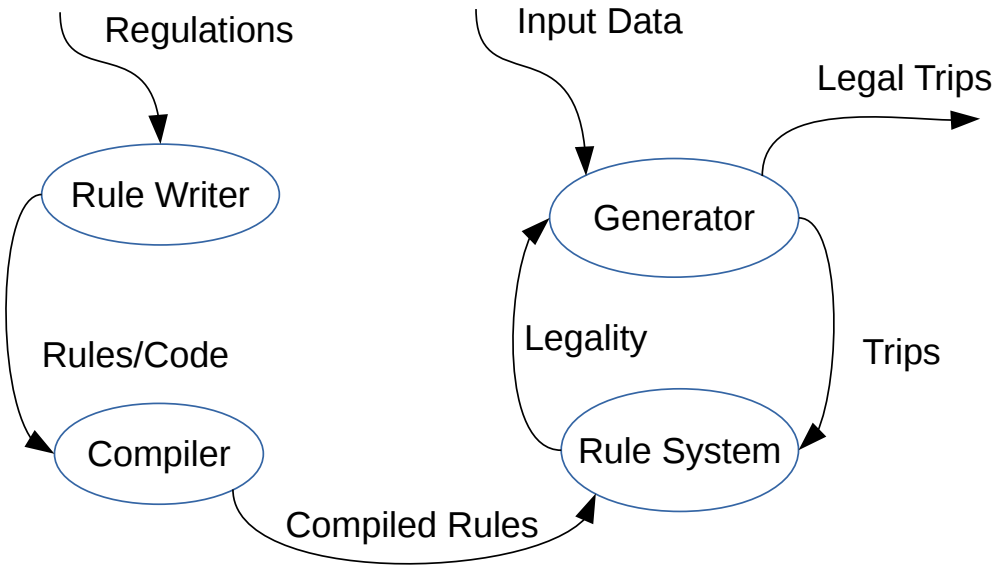


Figure 2.2: Illustration of the model of the system.

The root is an empty node which represents an empty trip starting in some airport. For each leg in the flight schedule starting in that airport, a child node is created. In this step, these child nodes each represent a potential trip of length 1. Each of these nodes has a child node created for every future compatible flight in the flight schedule. Two flights are compatible if the upcoming flight starts in the same airport that the previous flight ended, and the start of the upcoming flight is later than the end of the previous flight.
Following this procedure for each node, the entire search space of a flight schedule can be constructed. Figure 2.3 is an illustration of a small search space example.
Traversing down the search tree from the root, each child node represents adding a leg to the trip in progress, so each node represents a unique trip in progress.
Note that in practice, there is a limit on how many child nodes are explored for each node, as the number of nodes grows exponentially with each level in the tree structure.
Proposition 2.3.1. All possible legal trips are represented by a node in the search space.

Proof. Consider a trip t. Every trip is a sequence of legs which are compatible with each other. Starting at the root of the search space, visit the child nodes representing adding all the legs of t. The search space contain all legal leg-sequences, so either the node representing t exists, or t is not legal. □

Somewhat simplified, the Generator traverses the search space to find legal trips in what can be described as depth first search with some heuristics. Interfacing with

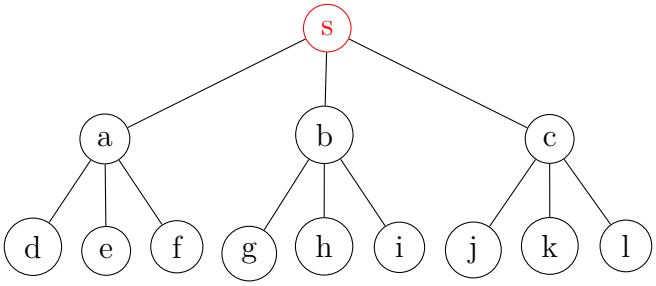


Figure 2.3: An illustration of a search space. This example shows three potential legs in each node, and has a search depth of 2.
The trip represented by node **i** consist of the two legs added in node **b** and **i**.

the rule system, the generator uses the Final Rules to prune the search tree, and the Illegal Subchain Rules to decide if a trip is complete.
For a trip to be **complete** and **legal**, it has to evaluate all rules to true.

2.3.2 Some formalisms

Informally, in the context of pairing, one may say that the difference between Final and Illegal Subchain rules is how the generator should behave when a rule is violated, i.e. what happens when the current trip is illegal according to some rule.
In order to reason formally about classification, the classifications also has to be formally defined. Some notation and definitions:

Definition 2.3.2. Node legality in relation to a rule: A node n in the search space is legal, or *satisfied*, according to a rule r if the trip represented by node n is legal according to r. This is denoted as $r(n)$.
$\neg r(n)$ denotes that n is illegal, or *violated*, according to r.

Definition 2.3.3. Legal Node: A node is considered legal if is is legal according to all rules.
$\forall r \in Rules.r(n)$

Definition 2.3.4. Subnodes: The subnodes of a node n is the set of all nodes that are reachable by traversing downwards from n. This represents the set of all possible continuations of the trip represented by node n. This is denoted as $s(n)$.

Here are the formal definitions for the rule classes used in this project:
Definition 2.3.5. Final Rule: A rule r is a Final Rule if, when the rule is violated for a node n, there exist no legal continuation of n. The following is true for a Final Rule r:
$\forall n.\neg r(n) \rightarrow \neg\exists n_s \in s(n).r(n_s)$

A rule being a Final Rule can be denoted $r \in$ **Final**
$\forall n.\neg r(n) \rightarrow \neg\exists n_s \in s(n).r(n_s) \iff r \in Final$
Definition 2.3.6. Illegal Subchain Rule: A rule r is an Illegal Subchain Rule

if, when the rule is violated for a node n, there may exist a legal continuation of n. The following is true for an Illegal Subchain Rule r.
$\exists n.\neg r(n) \wedge \exists n_s \in s(n).r(n_s)$

A rule being a Illegal Subchain Rule can be denoted $r \in$ **Illegal Subchain**.
$\exists n.\neg r(n) \wedge \exists n_s \in s(n).r(n_s) \iff r \in Illegal\ Subchain$
Note on $\exists n_s \in s(n)$: That there may exist a subnode, or that not all subnodes have some property depends on the search space. In practice, the possible search space depends on the input data, but in this model description it's interpreted as that it's *possible* to for a leg *to exist* that represents a node with this property. This reflects the fact that if there is no proof that there is no legal continuation, then generation should continue.
Here is an proposition that is useful for classification:
Proposition 2.3.2. Final or Illegal Subchain: A rule is either Final or Illegal Subchain.

Proof. For a rule r, assume $r \notin Final$. The Final Rule Property,
$\forall n.\neg r(n) \rightarrow \neg\exists n_s \in s(n).r(n_s)$, does not hold, but its negation holds.

$\neg[\forall n.\neg r(n) \rightarrow \neg\exists n_s \in s(n).r(n_s)]$	Negation of Final Rule Property
$\exists n.\neg[\neg r(n) \rightarrow \neg\exists n_s \in s(n).r(n_s)]$	$\neg\forall\varphi \iff \exists\neg\varphi$
$\exists n.\neg[\neg\neg r(n) \vee \neg\exists n_s \in s(n).r(n_s)]$	$\neg\neg - elimination$
$\exists n.\neg[r(n) \vee \neg\exists n_s \in s(n).r(n_s)]$	De Morgan, $\neg(\varphi \vee \psi) \iff (\neg\varphi \wedge \neg\psi)$
$\exists n.\neg r(n) \wedge \neg\neg\exists n_s \in s(n).r(n_s)$	$\neg\neg - elimination$
$\exists n.\neg r(n) \wedge \exists n_s \in s(n).r(n_s)$	Illegal Subchain Property

$Illegal\ Subchain\ Property = \neg Final\ Property$, so
$r \notin Final \rightarrow r \in Illegal\ Subchain$ and
$r \notin Illegal\ Subchain \rightarrow r \in Final$

□

This proposition means that if it is proven that a rule is *not* a Final Rule then it must be an Illegal Subchain Rule, and vice versa. Note however, that failing to find proof of a property is not the same as proving that it is false.
In order to classify rules in this project, rules are categorized after the structure of their defining expression. The intuition is that some patterns can be used to categorize rules into subsets of Final Rules and Illegal Subchain Rules.

2.3.3 Consequences of Misclassification

Rules that are never violated during Pairing Generation can never prune the search space. As such, when a Final Rule is misclassified as an Illegal Subchain Rule, then every violation of that rule implies a subtree in search space that is explored for which no subnode can represent a legal trip.
Consider an extreme example case where no rules affect the Pairing Generator, for example if all rules are interpreted as Illegal Subchain Rules. The only way to find which trips are legal for all rules is to examine all rules for all possible trips, i.e.

exhaustive search. Since the amount of possible trips grows exponentially with trip length, this is extremely inefficient.
On the other hand, if an Illegal Subchain rule is erroneously classified as a Final Rule and gets to prune the search space when there may exist legal continuations, then many possible solutions are never found. Omitting solutions carries the risk of omitting optimal solutions, leading to overall worse results.

2.3.4 Conditional rule checking

The way Rave enables incomplete trip checking in generation is through adding additional conditions to Illegal Subchain Rules. This is done using the Valid case described in Section 2.2.5.1. A way to make sure that a rule never prunes the search space unwantedly is to make sure that it is never violated during generation.
When generating trips, regarding a rule as trivially true implies that that rule will never prune search space. If a rule's e_v depends on an expression that is always false during generation, the legality of a rules expression e_r affects the trip's legality, but not the generator.
This functionality is possible through the `is_closed` keyword. It is false when checking rules to decide whether to keep generating continuations, and false when checking rules to decide if a rule is satisfied or not.
An example rule to illustrate how it works:
A rule that makes sure that a trip starts and ends in the same airport can look like this:

```
rule same_airport =
  valid is_closed ;
  last(leg(trip), %arrival_airport%) = first(leg(trip), %departure_airport%) ;
end
```

Here, `is_closed` makes sure that only trips starting and ending in the same airport are considered legal trips, and that even if a trip doesn't start and end in the same airport, there may exist a continuation that satisfies the rule.

2.3.4.1 Manual rule classification

The rules in Rave can be said to have two requirements in order to be useful. They should be validated by all intended legal trips, and they must be written so that the generator can find the legal trips by traversing the search space.
When translating regulations into Rave code, the first requirement is probably the most straightforward. The rule should be able to decide whether a trip is legal.
The second requirement demands that the rule writer understands how the generator works in order to write efficient rules, which is an additional mental overload.
The `is_closed` is sometimes misused or misunderstood, which can lead to faulty classification. The consequences of misclassification may be severe, and furthermore there is no obvious way to actually know that an error has been made.
The valid condition e_v can be formulated in many ways, but if it is "dominated" by `is_closed` then the rule will never prune search space.

```
%valid_condition% =
 any(leg(duty), %is_deadhead%) ;
```

	1 leg	2 legs	3 legs
e_r	F	F	_
e_v^g	F	T	_
e_v^c	F	T	_
gen	T	**F**	_
complete	T	F	_

```
%valid_condition% =
 any(leg(duty), %is_deadhead%) and
 duty.is_closed ;
```

	1 leg	2 legs	3 legs
e_r	F	F	T
e_v^g	F	F	F
e_v^c	F	T	T
gen	T	T	T
complete	T	F	T

Figure 2.4: Truth tables visualizing the different implications on rule completeness and generation on rule violation.

2.3.4.2 Conditional rule checking example with *is_closed*

Here is an example to show how `is_closed` affects generation in practice:
A deadhead leg is a leg where a crew member is traveling as a passenger without working, to be able to work on a leg starting in a different flight base.
Experience has shown that short duties containing so called deadhead legs are not generally part of good solutions, so a rule is to be written to exclude such rules from possible solutions.
The sentence "Any duty containing a deadhead leg must be at least 3 legs long" can be represented in Rave code as:

```
rule heuristic_deadhead =
  valid %valid_condition% ;
  count(leg(duty)) >= 3 ;
end
```

Here, $e_r :=$ `count(leg(duty)) >= 3` and $e_v :=$ `%valid_condition%`.

This rule should be able to allow the following example duty, d: Three legs, where the second leg is deadhead. This can be modeled by the leg expression `%is_deadhead%`, which is true for the second leg and false for the other two.
This rule needs a valid case in order to avoid being violated by every duty with fewer than 3 legs.
Figure 2.4 illustrates the relation between rule expression legality, generation and rule completeness. The rows show the boolean values of the evaluation of e_r, e_v, whether generation should continue, and whether the trip is considered complete according to the rule. For clarity, e_v is split up into e_v^g and e_v^c to illustrate the different values that e_v holds during generation and completeness checking. Notice that in the left case e_v^g and e_v^c are equal, since e_v does not depend on `is_closed`.
The columns represent the sub-trips of duty d.
The difference between the two cases in column 2 effectively show the importance of using `is_closed` correctly. In the left case, e_v becomes valid when the second leg is added, but e_r is violated, which means that the trip in progress violates this rule, and the Generator will not look for continuations of this trip. As such, the solution acquired by adding the third leg is never found.
In the right case, the trip containing the first two legs is not a legal (complete) trip,

but since the valid case depends on `is_closed`, the generator continues looking for legal continuations anyway and can find the solution with three legs.

3
Methods

In this chapter, the approach and methods used to fulfill the project goals are described. The Rave language with its compiler has been studied as well as parts of the Pairing Generation system in order to formulate the model and theory discussed in earlier chapters. This chapter will describe the formalia and implementation details that were developed during this project, along with some of the reasoning behind the decisions.

3.1 Classification and Attributes

The goal of this project is to develop and implement a method to automatically decide if a rule should be classified as Final or Illegal Subchain. In order to classify rules, a method to identify patterns in code was needed.
Typically in compiler optimization and code analysis, this is done on the ASTs of the code. In order to classify rules, some method was needed to convey additional semantic information about the expressions. Earlier experiences with type checking and some literature suggested that semantic analysis could be done by annotating the ASTs with desired properties [1, 26, 30].
This inspired the introduction of *Attributes*, which can describe properties that expressions will have when evaluated. Ultimately, the desired properties to detect were those that could describe patterns in legality similar to Final or Illegal Subchain Rules. Such properties were typically observed in traversers or expressions operating on traversers.
Much like type systems, the properties typically depend on the whole expression-tree and its subexpressions. For this reason, attributes are inferred in a fashion similar to typing rules, which is a common way of expressing typing relations [25, 26].
When inferring attributes, every expression is checked against the *attribute rules*, and if an expression has subexpressions they get checked first. This way, attributes are propagated bottom up, and the rules need only refer to an expression and the attributes of its subexpressions, if they exist.

3.1.1 Approach and Goals

The process of defining and redefining attributes and attribute rules have been done in an iterative fashion, where some commonly occurring rules and patterns in existing code has been studied. In parallel with this, the implementation of rule classification inside the Rave compiler has been developed. When new attribute rules

were implemented they were tested against some code examples, and the output was examined and compared against the desired outcome. This helped identify where rules were not strict enough, or when rules contained logic faults.

In the case that the current attributes were lacking to describe a pattern, the introduction of newer attributes and attribute rules were considered.

One goal that was set early was to find rules that compared growing numerical expressions against some limit, with defining expressions such as:

```
%time_sum% <= %limit% ;
```

```
%end_time% - %start_time% <= %time_limit% ;
```

This goal was decided upon because an important part of Crew Pairing is deciding for how long a crew member may work depending on different conditions. Therefore, these kinds of expressions were considered a good target pattern to detect. As such, examining the different kinds of rules that had this "shape" inspired many of the concepts that were developed.

Furthermore, one of the more easily identifiable Final Rules - comparisons with monotonic sums, described in Section 3.2.4.1 - has this shape, so this seemed like an appropriate early goal. However, small and subtle deviations from the patterns turned out to have consequences on legality patterns, at least what can be guaranteed.

This lead to further examinations of what could be assumed about these patterns and their legality.

3.1.2 Classification using attributes

During this project, different subcategories of rules have been identified, and knowing that some categories can be considered subsets of Final Rules or Illegal Subchain Rules, the categories can be used to classify rules accordingly.

The approach to classification was to use the attribute rules to identify legality patterns of the defining expression of rules, e_r.

Note that when looking at rule legality, the valid condition e_v can also be an important factor. However, to analyze the interplay between e_v and e_r and their effect on legality patterns was considered interesting but too advanced for the scope of this project. Especially considering that it would build upon the theories developed *during* this project. Therefore it was not prioritized at this time even though it may be interesting to study in more detail in the future.

Even though e_r was prioritized over e_v, legality patterns in e_r alone can show what patterns in legality the base of the rule has, which is crucial when deciding how to define e_v. For this reason, the legality patterns of e_r has been considered the most interesting component for rule classification. This pattern could be an indicator that a valid case is "missing" or lacking in order for that rule to be useful. Also, if a rule's legality pattern looks like a Final Rule, then that may be an indicator that a valid case should *not* depend on `is_closed`.

3.1.3 Attribute rules and propagation

Here is a short explanation on how attributes and their rules are written.
$e :_a A$ denotes that expression e has attribute A. This is similar to typing rules, where $e : T$ means that expression e has type T. The attribute rules are written to match the abstract expressions in order to be general and to apply to the ASTs of Rave.
Example attribute rule:

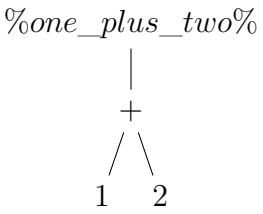
$$\frac{e_1 :_a A_2 \qquad e_2 :_a A_3}{e_1\ Op\ e_2\ :_a\ A_1}$$

This is read as: The expression $e_1\ Op\ e_2$, where Op is some operator, has attribute A_1 if e_1 has attribute A_2 and e_2 has attribute A_3. Informally, if the premises above the line are true, then the conclusion below the line is true. There may exist rules without premises, which are a way to say that an expression has a certain attribute regardless of the properties of its subexpressions.

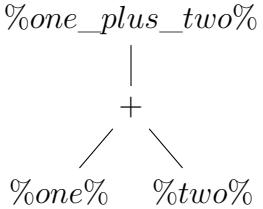
A convention that makes it easier to get an overview of expressions is that reference-expressions such as variable references are "followed" in the expression tree. For example, take the following code:

```
%one_plus_two% = %one% + %two% ;

%one% = 1 ;

%two% = 2 ;
```

When checking `%one_plus_two%` against the attribute rules, the structure of the analyzed expression is considered to have the following shape:



instead of the three separate trees:



This makes expressions much easier to check against the attribute rules, and to reason about.

Another simplification done in order to not write as many rules has been to consider some attributes subsets of other attributes, and the proofs of subsets to also act as proof of the superset. This can be compared to how polymorphism and inheritance is handled in many programming languages. For example: Positive numbers is a subset of Nonnegative numbers, so any proof that a number is positive also implies that the number is nonnegative. The same reasoning is done for "value direction" and "direction in legality" attributes which are described in Sections 3.2.3 and 3.2.4: If something is proven to be increasing, that also implies that it is nondecreasing.
In general, attributes that describe what is not true have been more versatile than those that guarantee something to be true, such as **NonNegative** in contrast to **Positive** and **NonDecreasing** in contrast to **Increasing**. This is in part related to the definition of Final rules, as what's interesting to prove is that an expression can *not* become legal again after being violated. Then it is enough to prove what direction in legality is *not* possible, as explored in Section 3.2.4.

3.2 Attributes by category

This section describes attributes introduced in this project along with some description on what they convey and how they are motivated. Some of the central attribute rules are described as well.

3.2.1 Range

These attributes describe the value range of expressions in very general terms. A basic but important expression property is knowing whether its evaluation is positive or negative. These properties are referred to as range attributes. Two central ones are **NonPositive** and **NonNegative**.
Two other related attributes considered are **IsZero** and **NonZero**, but there were no interesting findings relating to rule classification using these attributes in this project.
Expression-types of interest
Some of the central expression-types to this attribute category are constant values, arithmetic operations and sums.

- Constant values are simple. The value is clearly stated in code, so the range of the expression is the range of the value.
- Sums are nonnegative if their subexpression is positive or nonnegative, and they are nonpositive if their subexpression is negative or nonpositive.
- The range of *arithmetic operations* depend on both the operator and the operands. This is discussed in the following section.

3.2.1.1 Range of arithmetic operations

With the range known of the operands some resulting ranges can be deduced without access to the operands values, as can be seen in Figure 3.1. The implementation of this project uses these relations as basis for some simpler attribute rules. Not shown

	pp	np	pn	nn
*	p	n	n	p
+	p	?	?	n
-	?	n	p	?
<	?	True	False	?

Figure 3.1: The attributes of some arithmetic expressions knowing only the range of the operands. The rows tell which operator is used and the columns show the range of the operands, pp meaning two positive numbers, np meaning a negative value as left operand and positive value as right operand, and so on. The cells tell the range of the result of the operation. The bottom row also includes less than-comparison with cells telling whether the comparison holds.

in the figure is the simple case of unary negation, which simply changes sign of the value. Negation of a nonnegative value is nonpositive and vice versa.
This kind of reasoning about the value range of operations in code by looking at the range of the operands has been discussed by Cousot and Cousot [11,12], and Chin et al. describe their experiences with extending existing programming languages with features such as this [8,9].
Some results are uncertain as they may differ depending on if one of the operands is larger than the other, such as subtraction of two positive numbers. This is unfortunate since time durations, an important concept to trip legality, are typically computed as *end time* − *start time*. Rave, like systems such as UNIX's time stamp [28], represent a moment in time to be a positive duration since some starting time.
Even though the range of the subtraction of two positive numbers is generally unknown, if one of the operands is known to be larger than the other, the resulting range can be known:
$\forall a, b \in \mathbb{R} : a > b \rightarrow (a - b) > 0$
$\forall a, b \in \mathbb{R} : a < b \rightarrow (a - b) < 0$
Using knowledge of the generation process and traversers in Rave, it is in many cases possible to be certain of which of the times is largest without having direct access to the data.

3.2.1.2 Time relations and traversers

The values of times may be unknown at compile time, but all legs in the trip in progress are in chronological order because of how the Generator works. Traversers such as `first`, `last`, `prev`, and `next` can be used to deduce whether a subtraction of two unknown times is positive or negative.
Figure 3.2 shows the known size relations between times from traversers contexts and the "current" context. Current is not a traverser, but is used to distinguish that no traverser is present. These relations are transitive as well, although the implementation in this project needs further development before being able to guarantee the relationship between multiple traversers.
Notice that current context and `first`/`last` indexed contexts are not guaranteed to actually be different. This is because, as an example, if the current leg is the

		current	
index	keyword	departure	arrival
first	departure	$\leq$	$<$
first	arrival	?	$\leq$
prev	departure	$<$	$<$
prev	arrival	$<$	$<$
current	departure	$=$	$<$
current	arrival	$>$	$=$
next	departure	$>$	$>$
next	arrival	$>$	$>$
last	departure	$\geq$	?
last	arrival	$>$	$\geq$

Figure 3.2: Relations between arrival and departure keywords in combination with traversers. Rows represent traverser and keyword combination, columns represent the keyword that is compared against. The topmost left cell describes that the first departure is always $\leq$ current departure.

first leg in that instance. Then $departure \not> first(arrival)$. This is not the case for `prev` and `next` traversers, since when `prev` is evaluated where there exist no earlier instance, that result is ignored.

3.2.1.3 Findings as attribute rules

Referring back to the rules for arithmetics in Figure 3.1, the attributes rules for the different cases can be stated like this:

$$\frac{e_1 :_a NonNegative \qquad e_2 :_a NonNegative}{e_1 + e_2 \ :_a \ NonNegative}$$

This describes the fact that the addition of two nonnegative expressions result in another nonnegative expression.
The following rule describes how attributes propagate through conditionals. If all possible outcomes have the same range, then the conditional also has that range.

$$\frac{e_2 :_a NonNegative \qquad e_3 :_a NonNegative}{if\ e_1\ then\ e_2\ else\ e_3 \ :_a \ NonNegative}$$

The following two rules state that the sum-expression have the same range as its subexpression.

$$\frac{e :_a NonNegative}{sum(e) :_a \ NonNegative} \qquad \frac{e :_a NonPositive}{sum(e) :_a \ NonPositive}$$

Informally, this can be said to be true because sums are a sequence of additions, and if the range is proven for the subexpression, then that range is true for all instances of that expression. The `count` expression is a special case of sum with the subexpression 1, so it is trivially considered nonnegative:

$$\frac{}{count(e) :_a \ NonNegative}$$

Finally, all traversers that return a specific instance of an expression have the same range as the expression. This includes all traversers dealing with index - `first`, `prev`, `next`, `last` - along with `max` and `min`. Also, the `avg` traverser also has the same attribute of its subexpression, as the average of any number of nonnegative numbers is also nonnegative. The same also holds for nonpositive.

$$\frac{e :_a NonPositive}{next(e) :_a \ NonPositive}$$

3.2.2 Constantness and Dependencies

These attributes help describe when an expression instance may be considered constant. An expression is considered constant if it always evaluates to the same value during generation.

Some central attributes of this kind

ConstVal Tells that an expression is defined by constant value in the Rave code

DepOnlyConst Tells that an expression only has constant subexpressions, and can be considered constant

LevelConstant Tells that the value of an expression instance does not change in continued instances

LevelDependent Tells that an expression instance may change in continued instances

RefKeyw Tells if the expression is a keyword or reference to a keyword

ConstBranch For conditionals, tells if all outcomes are defined in terms of constants

Expression-types of interest

Definitions, Constant values, Arithmetics, Traversers, Conditionals

3.2.2.1 Attributes in relation to Instances and Context

The only expression types that can be considered constant in *all contexts* are those that are known at compile-time, or at least before runtime. In this project we simplify matters by also parameter definitions to be constant, since they can not be changed after generation has started.

Constant value definitions have the attribute **ConstVal**, and expressions that only depend on constant values have the attribute **DepOnlyConst**.

The evaluation of most traverser expressions change during continued instances, so most traversers imply that an expression has the attribute **LevelDependent** regardless of its subexpression.

Important to note is that practically all *instances* of expressions that are not defined in terms of traversers are constant. This includes most keywords that are used to define leg-expressions, such as flight times. However, since *which* instance of an expression gets evaluated depends on traversers or the context in which the expression is referenced, such expressions can not be considered constant under normal circumstances. Any claims that an expression is constant must be true for

all possible instances of that expression, otherwise no reliable conclusions based on assumptions on constantness can be made.
Other than constant values, it is interesting to know under which conditions an expression instance can be *considered* constant, for example if a duty-level expression always evaluates to the same value in all instances. When the only way to change a duty is to add legs after the last leg, the starting time of the duty is the same no matter how many legs are added. This notion inspired the **LevelConstant** attribute, which tells that an expression belongs to a level and there may be multiple instances, but the instances doesn't change in evaluation in continuations.
This attribute could be considered for many leg-level expressions as well, since leg is a level that has expression instances. However, since there is no way to continue leg instances, leg expressions are not considered **LevelConstant**.
In some sense, constant values can be considered level constant for the trip-level since they are constant for all continuations of a trip. This serves no real purpose since it is much more useful information that an expression is constant than level constant.
Conditionals, `if e1 then e2 else e3`, have some interesting properties depending on its three subexpressions. Important to note is that even if all outcomes are constant, *which* constant gets evaluated depends on the condition. In order to be considered fully constant, the condition has to be constant, and the specific expression that is constantly returned must also be constant. This is not very common since conditionals are typically used to ensure different outcomes, and a totally constant conditional defeats its own purpose. However, even if the whole expression can not be considered constant, some interesting properties can be observed in a conditional when the second and third subexpressions are constant. This is referred to as **ConstBranch**, and this attribute can have interesting implications on other attributes, and perhaps future optimizations. This is briefly discussed in Chapter 5.
Some of these attributes have not aided very much in classifying rules. This is in part due to the fact that rules by their nature describe dynamic properties, and constant rules would never contribute to Generation. However, constantness may still be interesting to study further, since many optimization strategies are possible where constant values occur. In order to be really useful in this context, more work on expression instances and levels in relation to each other need to be done. This project started with a naive approach about level instances that requires further development to be more accurate.

3.2.2.2 Findings as attribute rules

Some basic rules for constants:
Where n is a rave expression defined as a constant value,

$$\frac{}{n :_a ConstVal} \qquad \frac{e :_a ConstVal}{e :_a DepOnlyConst}$$

$$\frac{e_1 :_a DepOnlyConst \qquad e_2 :_a DepOnlyConst}{e_1 + e_2 :_a DepOnlyConst}$$

All expressions defined by a constant value are considered constant, and all constants have constant dependencies. If all subexpressions of an arithmetic operation only

depends on constants, then the operation has the attribute **DepOnlyConst**.
Constantness is a fragile property, and any level-dependency dominates propagation. This is because constantness is a strong claim, and **LevelDependent** can be said to represent "this expression may change in continuations". It doesn't exclude constant expressions, but it is a much more general claim that can not guarantee constantness.

$$\frac{e_1 :_a LevelDependent \qquad e_2 :_a DepOnlyConst}{e_1 + e_2 :_a LevelDependent}$$

$$\frac{e_1 :_a DepOnlyConst \qquad e_2 :_a LevelDependent}{e_1 + e_2 :_a LevelDependent}$$

Dependencies propagate in a similar fashion for all arithmetic operations, so the rest are omitted.
Some rules on conditionals:

$$\frac{e_1 :_a DepOnlyConst \qquad e_2 :_a DepOnlyConst \qquad e_3 :_a DepOnlyConst}{if\ e_1\ then\ e_2\ else\ e_3 :_a DepOnlyConst}$$

$$\frac{e_1 :_a LevelDependent}{if\ e_1\ then\ e_2\ else\ e_3 :_a LevelDependent}$$

$$\frac{e_1 :_a LevelDependent \qquad e_2 :_a DepOnlyConst \qquad e_3 :_a DepOnlyConst}{if\ e_1\ then\ e_2\ else\ e_3 :_a ConstBranch}$$

Rules relating to traversers:

$$\frac{}{sum(e) :_a LevelDependent}$$

Traversers are by their nature level dependent, since they evaluate expressions on multiple instances in a level. This rule also applies to `avg`, `min`, `max`, `next`, `prev`, and `last`.

$$\frac{e :_a RefKeyw}{first(e) :_a LevelConstant}$$

As mentioned, expressions like starting time of a duty is considered **LevelConstant**, because for that level instance it is considered constant. Here, **RefKeyw** refers to the fact that leg-times are defined by keywords. There may exist a need for a specific attribute for leg-times in the future, as keyword reference is a little too general.

3.2.3 Value direction

These attributes describe how the numerical value changes for continued instances of expressions during generation. Examples:

NonIncreasing Tells that the evaluation of an expression can never be greater in a continued instance

NonDecreasing Tells that the evaluation of an expression can never be lesser in a continued instance

As a side note, these attributes describe the semantics of these expressions in relation to generating trips in the model and not the Rave *language* itself.
Most relevant expression-type: Traversers

A very central concept that these attributes were intended to apply to are how time durations and accumulated time grows with generation.
Two ways to model this is to either:

- Take the ending time minus the starting time of a level instance
- Take the sum of time durations

The first case is handled using the same chronological reasoning about traversers in 3.2.1.2. In this project's model, this is simple, since starting times are considered constant, and ending times are considered nondecreasing.
This can be captured by a rule such as:

$$\frac{e_1 :_a RefKeywTime \qquad e_2 :_a RefKeywTime}{last(e_1) - first(e_2) :_a NonDecreasing}$$

Where $RefKeywTime$ is a reference to a keyword describing a time. This is currently done with just $RefKeyw$ in the implementation, with some additional filtering of which keywords are considered times.

The second case is referred to as Monotonic Sum, and that is described in the following section.

3.2.3.1 Monotonic Sums

A large part of the work in this project has been put into looking at *monotonic sums*, their properties, and how they affect the legality patterns of rules during Pairing Generation.
Definition 3.2.1. Monotonic Sum: A monotonic sum is a sum-expression whose subexpression is known to be either nonnegative or nonpositive.
A monotonic sum with a nonnegative subexpression is **nondecreasing**.
A monotonic sum with a nonpositive subexpression is **nonincreasing**.
A sum-expression whose subexpression is not known to be either nonpositive or nonnegative is not a monotonic sum.
Rules describing the direction of monotonic sums:

$$\frac{e :_a NonNegative}{sum(e) :_a NonDecreasing}$$

$$\frac{e :_a NonPositive}{sum(e) :_a NonIncreasing}$$

This is motivated as follows:
A sum starts at 0 and its value is computed as a sequence of additions of the evaluation of its subexpression in a number of instances. In a continued instance, the evaluation of a monotonic sum with a nonnegative subexpression can change in two ways: either the value of the subexpression is added to the sum, or it is not. In any case, the continued instance is nondecreasing. If generation results in a new instance of the expression, the old expression instance is unaffected, which also is considered nondecreasing.
The similar reasoning is used for monotonic sums with a nonpositive subexpression, which is nonincreasing.

3.2.3.2 Other findings and rules

Some other findings were reasoned about that did not have a great impact on rule classification. Here are some examples.
The direction is reversed when the expression with direction is negated.

$$\frac{e :_a NonIncreasing}{(-e) :_a NonDecreasing}$$

$$\frac{e :_a NonDecreasing}{(-e) :_a NonIncreasing}$$

Some additions with directed expressions preserve direction.

$$\frac{e :_a NonDecreasing \qquad e :_a NonDecreasing}{e_1 + e_2 :_a NonDecreasing}$$

$$\frac{e :_a DepOnlyConst \qquad e :_a NonDecreasing}{e_1 + e_2 :_a NonDecreasing}$$

Some other cases were considered where one operand was not considered constant or had a different direction. For example, $e_1 + e_2$ where $e_1 :_a NonDecreasing$ and $e_2 :_a NonNegative$. There is a clear trend here in evaluation but without more information available on e_2, all that is guaranteed is that e_1 may evaluate to a greater value in continuations. This guarantees that the minimum value of the expression instance increases during generation, but if e_2 decreases more between two continuations than e_1 increases, then the expression as a whole is not nondecreasing. Another interesting case considered is $e_1 - e_2$ where $e_1 :_a NonDecreasing$ and $e_2 :_a NonDecreasing$. Consider e_1 and e_2 to be two sums that grows linearly, but e_1 grows at a higher rate; the subtraction is clearly nondecreasing. However, such examples are quite specific and hard to capture using such a general concept as inference rules without some more extensive analysis.

3.2.4 Direction in Legality

These attributes describe properties of boolean expressions, and the patterns that may be observed in continued instances during generation. These are central to classifying rules in this project, as rules are defined in terms of boolean expressions whose evaluation changes during generation.
The attributes developed in this project:

NonIncreasingInLegality Tells that a boolean expression instance that evaluates to false can never have a continuation that evaluates to true

NonDecreasingInLegality Tells that a boolean expression instance that evaluates to true can never have a continuation that evaluates to false

NonFinal Tells that a boolean expression lacks linear direction between instances and their continuations

As these attributes describe change in evaluation during generation, traversers are the most important expression types to this attribute.
To get an intuition of what these expressions could look like, here are some patterns and code examples that held this property of "linear direction in legality" that this project aimed to describe with attributes.

1. Some threshold must not be crossed
   True until violated, then all possible continuations of instance will be violated. Examples:
   ```
   sum(leg(duty), leg.%time%) <= %max_time% ;
   count(leg(trip)) where (%is_deadhead%) <= 5 ;
   ```
2. Some threshold must be satisfied
   False until satisfied, then all possible continuations of instance will be satisfied. Examples:
   ```
   count(duty(trip)) >= 3 ;
   count(leg(trip)) >= 10 ;
   ```
3. Some incompatibility must not hold
   There can be no incompatibilities in a level or between level instances. True until violated, then all possible continuations of instance will be violated. Examples:
   ```
   all(leg(duty), %connection_time% >= %min_connection_time%) ;
   not(any(leg(duty), %prop1%) and any(leg(duty), %prop2%)) ;
   ```
4. Some property must hold for at least one object in level
   False until some condition is satisfied, then all possible continuations of instance will be satisfied. Examples:
   ```
   any(leg(duty), %is_international%) ;
   not(all(duty(trip), %undesired_prop%)) ;
   ```

Here, the patterns in 1 and 2 are tightly related, as are 3 and 4. They can almost be regarded as opposites, in the sense that they share structure but differ in truth patterns. Two of them are true until some condition is violated, and two of them are false until some condition is satisfied.
These patterns are what inspired the attributes "Nondecreasing/Nonincreasing in legality".
Definition 3.2.2. Nondecreasing in legality: A boolean expression is nondecreasing in legality (NdL for short) if an instance can never be violated in a continuation of a satisfied instance.

Definition 3.2.3. Nonincreasing in legality: A boolean expression is nonincreasing in legality (NiL for short) if an instance can never be satisfied in a continuation of a violated instance.

Informally, when adding more legs to a trip, the effect of the addition of legs can only make an expression "more true" and never "less true" if the expression is NdL and vice versa for NiL.

Of the mentioned rule patterns above, 1 and 3 are NiL, and 2 and 4 are NdL.
Important to note is that the attribute only describes one expression instance at a time, while the legality of a rule depends on the truth value for all instances of the rule. For example, consider a duty-level expression being nondecreasing in legality. When generating trips and the expression eventually becomes true, then all instance continuations will be true. When a new duty instance is created, the new instance may be false. This highlights the ambiguity of referring to an expression without specifying which instance of the expression is mentioned.

Looking at the pattern of the conjunction of all rule instances during generation, if the rule is NiL the pattern of the trip is NiL. If the rule is NdL, the pattern of the trip is potentially nonlinear.
Any expression with legality pattern that is not strictly "linear" has the attribute **NonFinal**.

3.2.4.1 Comparisons with monotonic sums

As mentioned in Section 3.1.1, one goal for this project was to be able to classify Final Rules that depend on comparisons with monotonic sums.
This kind of rule was used as the standard example of a Final Rule:

```
rule final_rule =
  sum(leg(duty), leg.%time%) <= %time_limit% ;
end
```

Having identified all necessary attribute rules to prove that the left subexpression of the comparison is a nondecreasing monotonic sum, two other aspects are important in order to claim direction in legality for the expression; the comparison operator, and the attributes of the right subexpression.
First, consider `%time_limit%` to have the attribute **DepOnlyConst**, and the comparison operator to be `<=`.
The rule is valid as long as the sum is below the value of `%time_limit%`. Once the threshold is crossed, no legal continued instance can exist, since the sum is nondecreasing. This implies that an expression with these properties is NiL! The following attribute rule is stated:

$$\frac{e_1 : NonDecreasing \qquad e_2 : DepOnlyConst}{e_1 \leq e_2 :_a NonIncreasingInLegality}$$

The same pattern is observed when the operator and value direction are reversed:

$$\frac{e_1 : NonIncreasing \qquad e_2 : DepOnlyConst}{e_1 \geq e_2 :_a NonIncreasingInLegality}$$

Conversely, when a value with a direction must satisfy a threshold instead of avoiding crossing it, the opposite direction in legality occurs:

$$\frac{e_1 : NonIncreasing \qquad e_2 : DepOnlyConst}{e_1 \leq e_2 :_a NonDecreasingInLegality}$$

What happens when `%time_limit%` is not guaranteed to be constant? If it is not constant, it may change between evaluations. This may be possible if for example, the threshold depends on come condition that may be level dependent.If the value of the threshold changes after the left operand has crossed the threshold, nonlinear legality patterns may emerge. As a consequence, the expression is not linear in legality, but **NonFinal**.

$$\frac{e_1 : NonDecreasing \qquad e_2 : LevelDependent}{e_1 \leq e_2 :_a NonFinal}$$

The same holds for the other comparisons between expressions with direction in value against variables which are not constant.

3.2.4.2 Attribute rules for Direction in legality

Similarly to how `sum` is a sequence of additions, `any` and `all` are a sequence of disjunctions and conjunctions respectively. This line of reasoning lead to the following rules:

$$\frac{}{any(e) :_a NonDecreasingInLegality}$$

$$\frac{}{all(e) :_a NonIncreasingInLegality}$$

This is because as long as one of the operands in a sequence of disjunction is true, the outcome will be true. The opposite holds for conjunction; a series of conjunctions is false as long as one of the operands is false.
As is true for some other rules, with creative use of traversers it is possible to find cases where these rules may be too general and as a consequence they might match some expressions that should not hold these properties. This may be revised in future iterations.

Similarly to (numerical) negation of a direction in value, (logical) negation of direction in legality implies the opposite direction of the negated expression.

$$\frac{e :_a NonIncreasingInLegality}{not\ e :_a NonDecreasingInLegality}$$

$$\frac{e :_a NonDecreasingInLegality}{not\ e :_a NonIncreasingInLegality}$$

Some patterns emerge when considering direction in legality and boolean connectives. For the same direction, the rules are quite straightforward.

$$\frac{e_1 :_a NonIncreasingInLegality \qquad e_2 :_a NonIncreasingInLegality}{e_1\ and\ e_2 :_a NonIncreasingInLegality}$$

$$\frac{e_1 :_a NonIncreasingInLegality \qquad e_2 :_a NonIncreasingInLegality}{e_1\ or\ e_2 :_a NonIncreasingInLegality}$$

$$\frac{e_1 :_a NonDecreasingInLegality \qquad e_2 :_a NonDecreasingInLegality}{e_1\ and\ e_2 :_a NonDecreasingInLegality}$$

$$\frac{e_1 :_a NonDecreasingInLegality \qquad e_2 :_a NonDecreasingInLegality}{e_1\ or\ e_2 :_a NonIncreasingInLegality}$$

However, two different directions combined result in **NonFinal** behaviour. Two examples that explain why "linearity" in legality can no longer be guaranteed are illustrated in Figure 3.3. Consider the patterns that may emerge in continuations of boolean expressions ψ and φ. In this example, $\psi :_a NonIncreasingInLegality$ and $\varphi :_a NonDecreasingInLegality$. As can be seen, the result of conjunction or disjunction of two expressions with differing direction in legality can not always be considered linear. However, an interesting observation is that for conjunction, after ψ turns false there is no possible continuation that is true. The same can be said in opposite direction for conjunction after φ turns true. Although the resulting expression is nonlinear, some linear properties remain and the expression has a "turning point", after which the pattern is linear.

ψ	T	T	F
φ	F	T	T
$\psi \wedge \varphi$	F	T	F

ψ	T	F	F
φ	F	F	T
$\psi \vee \varphi$	T	F	T

Figure 3.3: NonFinal pattern from boolean connectives and direction in legality. The rows represent different expressions and the columns represent instances, where the column to the left is the first instance and the ones to the right are continuations.

$$\frac{e_1 :_a NonDecreasingInLegality \qquad e_2 :_a NonIncreasingInLegality}{e_1 \; and \; e_2 :_a NonFinal}$$

$$\frac{e_1 :_a NonDecreasingInLegality \qquad e_2 :_a NonIncreasingInLegality}{e_1 \; or \; e_2 :_a NonFinal}$$

3.3 Direction in legality and rule classification

The goal of this project was to examine the possibilities of automatically classifying rules.
Having found patterns that implies direction in legality, and attribute rules that can find these patterns, what remains is to make the connection between rule classification and attributes.
Referring back to the definition of Final Rules and Illegal Subchain Rules in section 2.3.2 and comparing them to the definitions of NiL and NdL, a common pattern can be observed.
Proposition 3.3.1. Final Nonincreasing A rule that is defined by an expression that is nonincreasing in legality is a Final Rule.

Proof. A false NiL expression instance can have no continued instance that is true. If no continuation can be true, then all subnodes in the search space are illegal. This property matches the definition of a Final Rule. □

Proposition 3.3.2. Illegal Subchain Nondecreasing A rule that is defined by an expression that is nondecreasing in legality is an Illegal Subchain Rule.

Proof. A false NdL expression instance may have a continued instance that is true. This can be true for no Final Rule so the rule must be an Illegal Subchain Rule. □

Using these propositions, direction in legality can be used to classify rules!
A note on rules defined by **NonFinal** expressions. As the pattern of a **NonFinal** expression is nonlinear, it was initially assumed that these were not good candidates for pruning search space. Rules with this attribute are assumed to be Illegal Subchain Rules, as it is not safe to assume that rule violation implies no legal continuations can exist, but these patterns can probably be useful to study further.

3.4 Notes on implementation

In order to automate rule classification, this project has extended the Rave compiler with some additional functionality that implements the theories developed in this project.
The implementation can be likened to an extended type checking pass, and it takes place after regular type checking during compilation. The ASTs of all expressions are checked against the attribute rules to infer the attributes. An inspiration has been the LLVM compiler infrastructure project, and what is referred to as analysis passes that are used during code optimization [22].
The implementation adds a few notable components.

- The attributes, as a simple enumeration type.
- A datastructure called **ExpressionTree** which has a reference to an expression, and a list that keeps track of the inferred attributes of the expression. If the expression has any subexpressions, then a new ExpressionTree is created for each of those subexpressions. These new ExpressionTrees are referred to as the ExpressionTree's child-expressions.
- A lookup table that keeps track of which variable name refers to which ExpressionTree.
- A function for checking the attribute rules against an expression.

The implementation of the attribute rules is quite straightforward. When checking for attributes, the expression is pattern matched by its expression type, and the appropriate rules are applied. As an example, consider the attribute for addition of two nonnegative integers. The pseudocode for this looks like this:

```
...
switch exprtype
 case e1+e2 :
  if ( e1.has(NonNegative) && e2.has(NonNegative) )
  then attrs.insert(NonNegative);
...
```

The classification, briefly put, checks all ExpressionTrees against the attribute rules and outputs all inferred attributes for all named expressions in a Rave ruleset. The defining expressions and their inferred attributes are listed in a separate output for easy overview.

4
Results

In this Chapter, the results of this project will be presented.

4.1 What has been done in this project

During this project:

- The Rave language and compiler has been studied along with some details of how Pairing Generation is handled at Jeppesen.
- A model has been created to reason about the dynamics of elements in Rave code when used in Pairing Generation.
- Attributes have been introduced to map the semantics of the model to structures in Rave code. Some of which are closely related to rule classification.
- Finally, the implementation of attribute checking and rule classification has been developed inside of the Rave compiler.

4.2 Results of implementation

One measure of the findings of this project is the output the implementation when run on an existing Rave ruleset. Although a subset of the language has been covered in this project, around 40%-50% of the named expressions (variable definitions and rules) have at least some attribute inferred, mostly dealing with range, dependencies, or direction.
Around 20% of the Rules have inferred attributes relating to rule class, i.e. direction in legality.
Many of the example expressions and some rules that were considered important part-goals have appropriate attributes inferred. Some examples can be seen in Section A.1 in the appendix.
Some insights on important attributes and how they propagate has been discovered, as well as some of the dynamics of rule legality depending on expression structure. Some insights and reflections are written in Chapter 5.

4.3 Example of attribute propagation for rule

Here, a visual presentation of what the classification of a certain Final Rule looks like will be presented.

The rule in question and the expressions it depends on looks like this in Rave:

```
/* The rule */
rule final_monotonic_sum =
  %sum_duty_legtime% <= 10:00 ;
end

/* The sum */
%sum_duty_legtime% = sum(leg(duty), leg.%time%) ;

/* The legtime */
leg.%time% = leg.%arrival% - leg.%departure% ;
```

The AST of `final_monotonic_sum` as seen during attribute checking is illustrated in Figure 4.1.

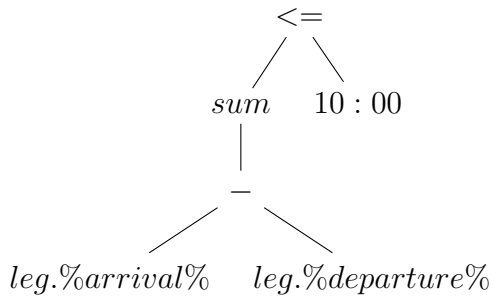


Figure 4.1: AST of rule **final_monotonic_sum** with references "followed"

When checking this rule, the keyword nodes that define `leg.%time%` get checked first. These are keywords known to represent times, and they have the attributes **NonNegative** and **RefKeyw**. The subtraction is checked next, and the operands are keywords with known relations. The result of the subtraction is **NonNegative**, which means that the sum-traverser has the attribute **NonDecreasing**. The right operand of the comparison is defined in terms of a constant, so that expression has the attribute **ConstVal**. Finally, the comparison <= with subexpression attributes **NonDecreasing** and **ConstVal** matches the rule for **NonIncreasingInLegality**. Because the defining rule expression has this attribute, the rule is a Final Rule.

5
Conclusion

This chapter reflects on some of the findings of this project and lists some suggestions for future work.

5.1 Findings of this project

This project approach the problem of classifying rules by static code analysis. Although far from all expression types in the language are covered, and many attribute rules have yet to be found for the expressions that were covered, some patterns have been defined and can be classified automatically at compile-time.
The benefits of this is that with some confidence, some rules can be claimed to behave predictably without the need to run them against actual data, which can save some time during rule writing. Even when not used for classification at compilation, the formalia developed can be used as reference when writing new rules.
As an experimental project, there are many potential continuations to explore.

5.1.1 Special mention on conditional variables

One interesting find when examining comparisons with monotonic sums in Section 3.2.4.1 is the case where the threshold is not constant.
As mentioned, this might result in **NonFinal** patterns in legality, which can be surprising. Without looking at the defining expression of the threshold that is compared against, the expression has the same "shape" as a typical Final Rule.
Depending on how variable the threshold is, some possible solutions may be omitted if this is considered a Final Rule. If the threshold is a **ConstBranch**, and the condition(s) are constant or has a direction in legality, then it may be possible to assign the threshold a direction in value.
It may be very interesting to look further into comparing expressions with direction in value against **ConstBranch**es or other expressions with a direction in value. Some intuition suggest that in most cases these rules should prune search space, but not unconditionally. Similar reasoning might be done on other **NonFinal** expressions with linear properties, as touched upon in Section 3.2.4.2.

5.1.2 Reflections on expressiveness of classification

When developing the formalia for this project, one interpretation of rule classification is to tell whether the search space is "safe" to prune at rule violation. Safe in this case means that no valid solution gets discarded.

The linear approach of using direction in legality-related attributes to differentiate Final Rules and Illegal Subchain Rules may seem a little limiting at first, but it could be argued that most Final Rules would have to be linear in some sense in order to be predictable. Pruning a branch in a search tree without knowing if it may contain a valid solution is a bit limiting. A rule that does not have a linear pattern in legality is therefore not a very strong candidate for a Final Rule, meaning that the suspicion that it is an Illegal Subchain Rule increases.

One important aspect of the attribute rules is that *uncertainty dominates propagation*.

Unknown range of any subexpression in an arithmetic operation implies unknown range. Unknown dependencies of subexpressions makes it impossible to assume that an expression can be considered constant. Unknown direction means that no claims of linearity can be made. To generalize a little, an expression is usually at least as "uncertain" as it's least "certain" subexpression.

This complicates matters a little for classification considering the implications of Illegal Subchain Rules; they state that it is not certain that all subnodes are violated when a node is violated.

Taking this projects cautious approach, if a rule's pattern in legality is not predictable, it probably should not be allowed to prune the search space, at least not unconditionally. As mentioned in section 3.1.2, that a rule's defining expression is not considered Final suggests that a valid case is needed in order for the rule to be useful.

5.2 Other applications of semantic analysis

Can the semantic information developed during this project be of use for more than rule classification, and what is interesting to continue developing?

Cire et al. [10] mention that semantic information of the computations in a domain model is a good source for further optimizations.

The system in use at Jeppesen is very resource intensive and performance is important. The semantic information provided by this project might not affect runtime calculations directly, but it can be used in order to assure that the input has certain properties, or behaves predictably. The more information that can be known about a problem, the more opportunities for further developments are possible. Furthermore, since the semantic information is gathered before runtime, static analyses does not impact performance.

Attributes and classification may be somewhat limited in what they can convey, but they describe an important part of the studied system. Rule class and attributes are static properties, but the existence of legal continuations is a dynamic property. The point of Final Rules is to safely discard computations that can not result in a better solution.

5.3 Future Work

For continued development of semantic analysis in relation to this project, this section brings up some potential areas to explore.

5.3.1 Explore more attributes and rules

One idea is to develop more attributes for further rule classification, or for proving other properties at compile-time.
There are probably many interesting semantic properties than those discovered in this project, as well as many expression types that have yet to be applied to attribute rules. Some particularly interesting attributes to look into are **NonFinal** and **ConstBranch**. Also, extending the current reasoning to handle multiple level instances and traversers may lead to interesting discoveries. There may be more appropriate approaches to instances than attributes however, perhaps some light interpretation is more intuitive.
Flanagan et al. [14] among others point out that static analysis alone is limited in what semantic properties it can detect, and suggest that there are more sophisticated methods for verifying code. However, they also point out that detection of some semantic properties may be undecidable, and impossible to do statically.
There are many approaches to semantic analysis other than using inference rules that can be used. Beckert and Hähnle has written an overview of some approaches to verification of software and code analysis [4].
Attributes, by design, are quite general. An expression attribute has to hold for all instances of that expression. However, when an expression depends on dynamic data, especially when multiple outcomes are possible, it may be desirable to express more specific expression properties than what is "always true for all instances".
For this project, the reasoning has been quite limited to state "what can be known to be true at all times, looking at expressions one by one".
Since all rules in a ruleset have to be valid at the same time in order for a node in search space to be valid, it could be interesting to try to model what conclusions can be drawn by assuming that multiple rules are true at the same time. As mentioned earlier, it may also be interesting to explore the relation between e_r and e_v of a rule and their combined contributions to rule legality patterns.
Cadar and Sen has written an overview of trends in symbolic execution, a technique that may be applicable here [7]. Some expressions can be assumed to hold symbolic values during analysis in order to reason about what outcomes are possible at runtime. The symbolic values of expressions may be decided from other expressions.

5.3.2 Assisting runtime optimization

Aside from assuring that the rules and input data are well formed, the semantic information could possibly be implemented in order to develop new strategies at runtime.
Knowels & Flanagan [20] discusses hybrid techniques that combine both static checking and dynamic checking (at runtime) for deducing type information of programs in

order to gather more specific type information that might not be possible to gather at compile-time.
Since the attributes in this project have much in common with type systems, it could be interesting to explore the possibilities of "hybrid" or "dynamic" attributes that are affected by runtime conditions. It is noted that hybrid techniques such as these do add some computational overhead, which is undesirable in such a resource intensive task such as Pairing Generation. However, in some cases this trade-off may be worth considering.

One interesting approach that was discussed during the developments of the project is the possibility to filter the search space after a rule violation for potential valid nodes without having to check every node against the rules. If one can pinpoint the exact property or expression that caused the rule violation, it may be possible to formulate exactly what properties that a valid continuation must satisfy. If this can be expressed in a simple formula that can quickly decide if a leg holds this property, perhaps this could save time when pruning the search space.

5.3.3 Simplify rule writing

Some modern Integrated Development Environments make use of "semantic awareness" to detect potential errors in code as it is being written. One such example is described by Logozzo et al. [23].
Applying the attribute rules on Rave code as it is being written, classification might find is there is a mismatch between rule structure and inferred class. This way, a rule writer may be notified of the mismatch, and potential logical errors can be avoided.
Flanagan et al. [14] describe a framework for extending a language for automated verification of code. One feature included is the possibility to describe properties that must hold of expressions in the code, which can be verified by code interpretation.
Annotation may make it easier to spot logic errors, or perhaps infer or manually assign attributes, but if the goal is for the rule writer to express rules more easily, this might not be the best approach. Beckert and Hähnle [4] note that specification is a bottleneck of verification, meaning that it takes great effort to specify everything that is expected to be true. Having the possibility to annotate does not mean that it is a necessity though.
An interesting case where formal rules are used during code writing is the functional language and proof assistant Agda, described by Bove et al. [5]. Agda features gradual refinement of expressions using its rich type system. Looking at attributes as an extended type system, perhaps Rave could use the attribute rules along with some additional restrictions in order to assist in writing good rules. This is however much more advanced than what might be desirable for a rule writer, but for proving properties about rules it might be interesting to take inspiration from proof assistants. Flanagan et al. [14] also incorporated automated theorem provers for their verification of Java code.

This project has discussed how the valid case affects generation, but an interesting question not covered is how to produce a good e_v expression.

A very basic approach is "Illegal Subchain Rules should never prune search space". This is simply modeled by having e_v defined as `is_closed` or some other form where `is_closed` dominates the truth value during generation. However, a more complex valid case may result in a rule that doesn't break generation, but is still able to prune search space. As mentioned in Section 3.2.4.2, some **NonFinal** expression have a turning point where all continuations are linear in legality. Finding these patterns to allow **NonFinal** rules to prune search space could be an interesting project.

5.4 Conclusion

This project has developed a model to describe semantic information about expressions in Rave code. The information describes properties that the expressions hold at runtime - during Pairing Generation.
This model has been implemented as part of Rave's compilation process, and as a result it has successfully been able to detect a set of rules with properties that can be used to reason about the patterns in legality during generation. Some rules are identified as Illegal Subchain Rules, and some as Final Rules that are guaranteed *not* to be Illegal Subchain Rules.
The method of analyzing the code can be compared to type checking, using inference rules to propagate semantic information in the ASTs of the expressions in the code. The inference rules look at expressions and the computed properties of its subexpressions, if available. New rules and properties are easy to implement, and more coverage is probably possible with continued work. However, inference rules are somewhat simple and more advanced techniques may be interesting to utilize when looking at future developments.

Revisiting the question: *Is it possible to detect Illegal Subchain Rules at compile time?*
In order to answer this question, some reasoning was done in order to approximate what would be needed to detect these rules. Some patterns were developed that have successfully categorized a few rules to have certain properties. Some of these properties, it has been reasoned, are only true for certain subsets of each of the rule classes.
A few rules have been classified at compile time, and more can probably be found by continued development of the methods. Though there is a limit to how much can be deduced statically, some interesting future work with semantic analysis can probably lead to interesting findings.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[2] Erik Andersson, Anders Forsman, Stefan E. Karisch, Niklas Kohl, and Allan Sørensen. Problem solving in airline operations. *OR/MS Today*, April 2005.

[3] Erik Andersson, Efthymios Housos, Niklas Kohl, and Dag Wedelin. Crew pairing optimization. *Operations Research in the Airline Industry*, pages 228–258, 1998.

[4] Bernhard Beckert and Reiner Hähnle. Reasoning and verification: State of the art and current trends. *IEEE Intelligent Systems*, 29(1):20–29, 2014.

[5] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[6] Maurice Bruynooghe. A practical framework for theabstract interpretation of logic programs. *The Journal of Logic Programming*, 10(2):91–124, 1991.

[7] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[8] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. *SIGPLAN Not.*, 40(6):85–95, June 2005.

[9] Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *ESOP*, volume 6, pages 264–278. Springer, 2006.

[10] Andre A Cire, John N Hooker, and Tallys Yunes. Modeling with metaconstraints and semantic typing of variables. *INFORMS Journal on Computing*, 28(1):1–13, 2016.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[12] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2-3):103–179, 1992.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: past, present and future. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 2. ACM, 2014.

[14] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5):234–245, May 2002.

[15] Christos Goumopoulos and Efthymios Housos. Efficient trip generation with a rule modeling system for crew scheduling problems. *Journal of Systems and Software*, 69(1):43–56, 2004.

[16] Curt A Hjorring and Jesper Hansen. Column generation with a rule modelling language for airline crew pairing. In *Proceedings of the 34th Annual Conference of the Operational Research Society of New Zealand*, pages 133–142, 1999.

[17] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[18] Atoosa Kasirzadeh, Mohammed Saddoune, and François Soumis. Airline crew scheduling: models, algorithms, and data sets. *EURO Journal on Transportation and Logistics*, 6(2):111–137, 2017.

[19] Sarmen Keshishzadeh and Arjan J. Mooij. Formalizing DSL semantics for reasoning and conformance testing. In *International Conference on Software Engineering and Formal Methods*, pages 81–95. Springer, 2014.

[20] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):6, 2010.

[21] Niklas Kohl and Stefan E. Karisch. Airline crew rostering: Problem types, modeling, and optimization. *Annals of Operations Research*, 127(1):223–257, 2004.

[22] Chris Lattner. LLVM. In *The Architecture Of Open Source Applications*. Creative Commons, 2011.

[23] Francesco Logozzo, Michael Barnett, Manuel A Fähndrich, Patrick Cousot, and Radhia Cousot. A semantic integrated development environment. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 15–16. ACM, 2012.

[24] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[25] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[26] Aarne Ranta. *Implementing programming languages. An introduction to compilers and interpreters*. College Publications, 2012.

[27] Yuriy Solodkyy, Jaakko Järvi, and Esam Mlaih. Extending type systems in a library. In *Second International Workshop on Library-Centric Software Design (LCSD'06)*, page 55, 2006.

[28] Ken Thompson and Dennis M Ritchie. *UNIX Programmer's Manual*. Bell Telephone Laboratories, 1975.

[29] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-specific Languages*. CreateSpace Independent Publishing Platform, 2013.

[30] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler design: Syntactic and semantic analysis*. Springer Science & Business Media, 2013.

A

Appendix 1

This document contain some examples of Rave expressions and their computed attributes.

A.1 Example code with computed attributes

Here are some example expressions with their computed attributes inside `/* ... */` comments.
Here are some simple examples with range-attributes.

```
%one% = 1 ; /* NonNegative */
%two% = 2 ; /* NonNegative */

%minus_one% = -1 ; /* NonPositive */

%one_plus_two% = %one% + %two% ; /* NonNegative */
%one_minus_minus_one% = %one% - %minus_one% ; /* NonNegative */

%unknown% = 10 - (%two% - %one%) ; /* ??? */
```

In the case of `%unknown%`, the current attribute system only check the values of constant expressions. `%two% - %one%` is nonnegative, but only this attribute is checked when the other subtraction is checked. The value is "lost" from `%two% - %one%`, and all that is known about the other subtraction is 10 - (Something nonnegative). This is unfortunate, and has led to some constant expressions that are known at compile time not being recognized by the attribute rules to be NonNegative or NonPositive. Especially, a few times that are used in rules are not properly found to be nonnegative, leading to potentially fewer correct classifications.
The only expressions that have been manually assigned attributes in this project are the keyword expressions `arrival` and `departure`. The time is known to be NonNegative, because arrival is always later than departure in a flight. Time with index traversers such as `prev` are known to be earlier than current leg, meaning subtractions can have known range.

```
leg.%start_utc% = keywords.%departure% ; /* NonNegative */
leg.%end_utc% = keywords.%arrival% ; /* NonNegative */

leg.%time% = %end_utc% - %start_utc% ; /* NonNegative */

leg.%start_utc% - prev(leg(duty), leg.%end_utc%) ; /* NonNegative */
```

Here are examples of the traversers sum and count. Both range and direction attributes are inferred.

```
%trip_legcount% = count(leg(trip)) ; /* NonNegative, NonDecreasing */

%duty_legsum_time% = sum(leg(duty), %time%) ; /* NonNegative, NonDecreasing */
```

Expressions with conditionals or multiple possible outcomes need additional rules for proper attribute inference. Attributes are static properties, so the attributes must be applicable to all possible outcomes of an expression in order for that expression to have that expression.

```
%max_time_duty% = /* NonNegative */
if %some_condition%
  then 5:00
  else 8:00 ;
```

`%max_time_duty%` results in a nonnegative expression regardless of the value of `%some_condition%`, so the entire expression is considered nonnegative. However, even though all possible outcomes return a constant value, which constant to be returned depends on `%some_condition%`. Therefore, the dependency of the expression is not constant. However, if `%some_condition%` is constant, then `%max_time_duty%` can also be considered constant.
Here is an example rule. It compares a nondecreasing sum against a variable.

```
rule max_time_duty1 =
  %duty_legsum_time% <= %max_time1% ; /* NiL */
end

%max_time1% = 8:00 ; /* NonNegative */
```

If `%max_time1%` is a constant value, then the rule is nonincreasing in legality, and the rule is a Final Rule.
However, since many limits depend on different conditions, limits compared against may change during generation. If this is the case then, according to the definitions in this project, the rule is an Illegal Subchain Rule. Consider the following example:

```
rule max_time_duty2 =
  %duty_legsum_time% <= %max_time2% ; /* ??? */
end

%max_time2% = /* NonNegative */
if %condition%
  then 12:00
  else 8:00 ;
```

Depending on the variability of `%condition%`, it may be possible that `max_time_duty2` is either a Final or Illegal Subchain Rule.
It may not always be obvious how `%condition%` is defined either, so it's not safe to assume too much. In this project, unless `%condition%` can be considered constant,

the rule is classified as Illegal Subchain.
Another example rule:

```
rule 3legs =
  count(leg(duty)) >= 3 ; /* NdL */
end
```

The rule expression is NdL, so the rule is Illegal Subchain. Since a valid case is missing, this rule is misclassified, and will discard valid solutions during generation.
Two examples where the attribute rules are too "unforgiving" and needs more work:

```
%not_decreasing% = /* NonNegative */
if false
  then 10
  else %leg_duty_count% ;

%leg_duty_count% = count(leg(duty)) ; /* NonNegative, NonDecreasing */

%not_variable% = /* NonNegative */
if %unknown_condition%
  then 20
  else 20 ;
```

Even though `%not_decreasing%` always returns a nondecreasing expression, this is not checked in the current implementation.
Also, because its not sure which expression is returned in `%not_variable%`, it's assumed that the expression is not constant even if the result is always the same.