Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Piotr Sokólski
Nr albumu: 292408

Relaks: a Framework For Declarative Meta-Learning

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
prof. dra hab. Krzysztofa Stencla
Instytut Informatyki

Wrzesień 2015

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Abstract

Working with algorithms whose performance greatly depends on their parameters requires conducting numerous experiments and using expert knowledge in order to achieve best results. In this thesis we evaluate strategies for experimentation and parameter tuning, especially in the context of machine learning. We introduce Relaks – an embedded programming language – as a framework for conducting experiments and automatic parameter tuning. An implementation of the language is presented and later applied in the machine learning domain. We discuss strengths and weaknesses of the proposed solution and suggest directions for further development.

Słowa kluczowe

meta-learning, machine learning, hyperparameter tuning, multi-stage programming, domain specific languages

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

I.2.5 [Artificial Intelligence] Programming Languages and Software – Expert system tools and techniques
I.2.6 [Artificial Intelligence] Learning
D.3.4 [Programming Languages] Processors – Optimization; Interpreters
D.2.11 [Software Engineering] Software Architectures – Domain specific architectures; Languages

Tytuł pracy w języku polskim

Relaks: platforma do deklaratywnego meta-uczenia

Contents

Introduction 5

1. Background 7
- 1.1. The Machine Learning Process 7
  - 1.1.1. Supervised Machine Learning Basics 7
  - 1.1.2. Experimentation 8
  - 1.1.3. Hyperparameter Tuning – an Illustrative Example 8
- 1.2. Algorithms for automated hyperparameter optimization 10
  - 1.2.1. Grid Search 10
  - 1.2.2. Other Model-Free Search Methods 10
  - 1.2.3. Bayesian Optimization 10
- 1.3. Present Approaches to Hyperparameter Tuning in Machine Learning 11
  - 1.3.1. Holistic Approaches 11
  - 1.3.2. Extensions to Popular Machine Learning Libraries 12

2. Relaks 13

3. DSL Basics 15
- 3.1. The Meta-Language: Scala 16
- 3.2. Intermediate Representation 16
- 3.3. Embedding a Language 16
- 3.4. Type Reuse 18
- 3.5. Modular Architecture 19
- 3.6. Drawbacks of the Pure Functional Approach 19
  - 3.6.1. Graph Intermediate Representation 19
  - 3.6.2. Advantages of the Graph Representation 21
  - 3.6.3. Modular Architecture One More Time 22

4. Queries in Relaks 23
- 4.1. LINQ 23
  - 4.1.1. LINQ in Scala 24
- 4.2. Implementation in Relaks 25
  - 4.2.1. Staging Tuples 25
  - 4.2.2. Peeking Inside Functions 28
  - 4.2.3. Staging Queries 29
  - 4.2.4. Improving the Syntax 34

5. Hyperparameter Tuning . . . 37
5.1. Hyperparameter Tuning as Iterative Refinement . . . 37
5.2. Hyperparameter Tuning as a Stream . . . 38
5.3. Stream Processing Basics . . . 38
5.4. Basic Architecture . . . 39
5.5. The Grid Search Optimizer . . . 40
5.6. Concurrency . . . 41
5.7. Spearmint Optimizer . . . 41
5.8. Extending the DSL . . . 43

6. Traversing the IR . . . 47
6.1. Static Analysis for Program Correctness . . . 47
6.2. IR Attribution . . . 48
6.3. IR Rewriting . . . 49
6.3.1. Rewriting LINQ to a Simpler Representation . . . 49
6.3.2. Rewriting for Efficient Execution . . . 52
6.4. The IR Traversal that Could – the Interpreter . . . 52
6.4.1. A Preliminary: Type Materialization . . . 53
6.4.2. The Interpreter . . . 54
6.4.3. Environments . . . 55

7. Advanced Extensions . . . 57
7.1. LINQ to Drill . . . 57
7.2. Calling Native Scala Code . . . 58
7.3. Hoisting and Caching . . . 60
7.3.1. Implementation . . . 62

8. Evaluation . . . 65
8.1. Branin-Hoo . . . 65
8.2. Knn . . . 67

9. Discussion and Future Work . . . 71
9.1. Future Work . . . 71

Bibliography . . . 73

Introduction

Working with algorithms whose performance greatly depends on their parameters requires conducting many experiments and using expert knowledge in order to achieve best results. The advent of systematic approach to experiments and automated methods of parameter space exploration enabled the creation of tools which aid programmers in this process.

Chapter 1 provides an overview of solutions for experimentation and parameter tuning available in the literature, especially in the context of machine learning. Chapters 2 to 7 introduce Relaks programming language as a framework for conducting experiments and automated parameter tuning in machine learning. Later, in chapter 8 examples of real life scenarios in which Relaks shows its strengths are given. Finally, chapter 9 contains a discussion about strengths and weaknesses of the proposed solution and ideas for further development.

Chapter 1

Background

A programmer attempting to solve a problem usually has more than one way of doing so. While some choices can be easily made based on business requirements and other constraints, it is hard to distinguish between the others at the time of designing and implementing the program. Deferring those decisions until later stages of development in the form of parameters is a common practice. Then a program space, rather than a single program, is formed, capable of solving a range of problems efficiently.

One area where algorithms benefit from exposing the parameters rather than hard-coding the decisions based on intuition is machine learning. The goal of these algorithms is to extract information from data, creating a model that summarizes and provides useful insight into the input. Algorithms classified as machine learning are often heuristic in nature, each of them with some expectations of the data's hidden structure. Their parameters (in this case called *hyperparameters* to distinguish them from the parameters of the underlying model) express further assumptions about that structure. Unfortunately, according to the no-free-lunch theorem [Wol95; Wol96; For05] no "rule-them-all" machine learning algorithm exists. A developer interested in achieving maximum performance must look for the best solution on a per-instance basis. Caffe Model Zoo[1] is a model repository, where multiple fine-tuned versions of just one algorithm (Convolutional Neural Networks) along with the image-based data sets they are best suited to work with are listed. From existence of such repositories it is evident that the scale of the problem of finding the best algorithm is vast. Even with data sets built from just several features it can be hard to find the best fitting algorithm with only the "naked eye", so the programmers' method of choice for selecting the best performing classifier is experimentation.

1.1. The Machine Learning Process

In the following sections a basic framework and definitions for supervised machine learning and experimentation are introduced.

1.1.1. Supervised Machine Learning Basics

The first step of applying a machine learning algorithm is called the *learning step*, where a summary (in the form of a model or a classifier) is constructed from a *training data set*. A training data set consists of multiple *instances*. Each instance is made up of tuples (X, y), where X is a vector of values called *feature vector* or *attribute vector* and y is the *label*

[1] https://github.com/BVLC/caffe/wiki/Model-Zoo

associated with the feature vector. The goal of the learning step is to *fit* a function $y = f(X)$ that best predicts the label y associated with each feature vector X. Because each label of the training data set is known beforehand, this is a *supervised learning* task.

In the next step, called the *prediction step* the learned function f is used to predict the labels for given vectors X.

To estimate the usefulness of the learned model both *training* and *test* (also called *generalization*) *errors* can be computed. Error is a measure of how good the predictions given by the model are. One such measure is *accuracy*, a percentage of correctly vs. incorrectly predicted labels. Test error, computed on the data previously unseen by the algorithm is especially important, as the model often tends to *overfit* the data, discovering structures that are present in the training set, but not true for the general data set. Instances used for computing generalization error belong to the *test set*.

1.1.2. Experimentation

> It can not be emphasized enough that no claim whatsoever is being made in this paper that all algorithms are equivalent in practice, in the real world. In particular, no claim is being made that one should not use cross-validation in the real world.
>
> *Wolpert [Wol95]*

Estimating the error of a trained model is important not only for assessing the usefulness of its future predictions but also for choosing a learning algorithm that best suits the task at hand. *Model selection* or *hyperparameter tuning/optimization* is the problem of minimizing the generalization error on an independent data set by choosing the best set of hyperparameters.

Cross-validation is an useful tool for measuring the generalization performance. *K-fold cross-validation* is performed by dividing the training set into k equally sized mutually exclusive subsets $D_1, \ldots, D_k$. Then for each $i \in \{1, \ldots, k\}$ the union of subsets (D_j), where $j \neq i$ is used as the training set and D_i becomes the test set. When the data set is a good representation of our problem and error rates across different folds are similar, we can confidently estimate the general error [Koh95].

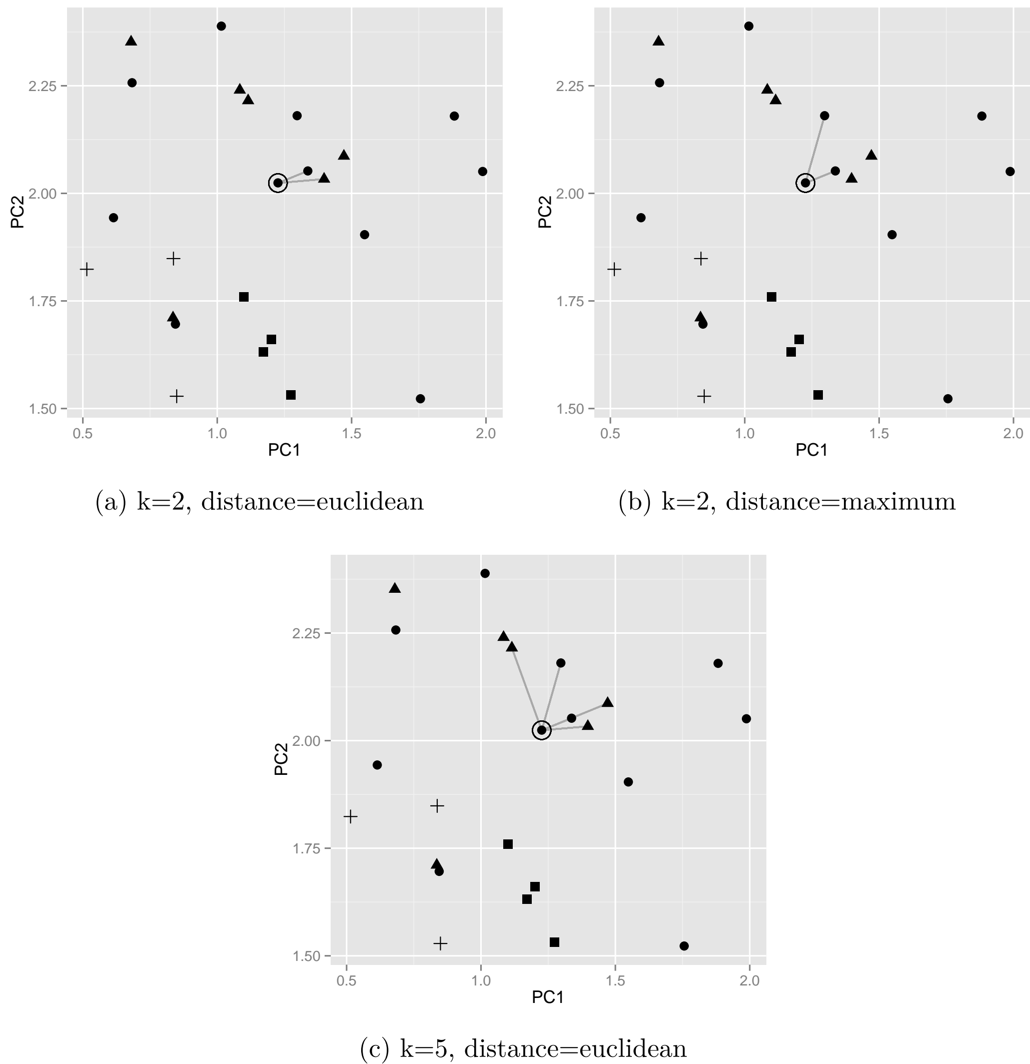
Note that in the case of hyperparameter tuning in order to correctly measure the algorithm's performance some data should be withheld until the meta-learning process is finished and then used with the final algorithm instance to compute the error rate. As hyperparameter tuning is also a learning procedure, in order to correctly assess the risk of overfitting the tests must be run on data that has never been used for training.

1.1.3. Hyperparameter Tuning – an Illustrative Example

K nearest neighbors (knn) is one of the simplest classifiers. Given a training data set, the decision on how to classify a new example is based on the labels of instances in the training data set closest to the example (the nearest neighbors). Even for a simple algorithm such as knn there are several decisions to be made:

1. What distance measure is to be used? E.g. the basic Euclidean distance or maximum distance between two coordinates of the instances (the supremum norm).

2. How many closest instances should be considered (this parameter is the k in knn)?



(a) k=2, distance=euclidean

(b) k=2, distance=maximum

(c) k=5, distance=euclidean

Figure 1.1: Visualizing knn for different hyperparameter configurations

3. Whenever there is lack of consensus on the label among the nearest neighbors, how should the winning label be chosen? The decision could be made based on weighted on unweighted majority.

Figure 1.1 shows different results of a multiclass classification scenario produced by programs run with different hyperparameters.

In each example the task is to predict the label associated with the ◉ instance – its true label is •. The lines connect the k closest neighbors. We can see that in our example among evaluated configurations only $k = 2$ with the maximum distance function (fig. 1.1b) would be able to correctly classify the example (assuming unweighted majority vote).

We do not know if those hyperparameters provide good results for our problem and we can easily imagine scenarios where they would not. The generalization error should be estimated using the cross-validation technique described in section 1.1.2.

1.2. Algorithms for automated hyperparameter optimization

In this section a few methods of exploring the machine learning algorithm space defined by the hyperparameters are presented. Both methods that make no assumptions about the space and those that try to create a model given the error function have been successfully applied the in machine learning context.

1.2.1. Grid Search

Grid search is the most straightforward way of performing parameter optimization. The parameter space is searched exhaustively.

In our knn example (section 1.1.3) the space could consist of pairs (k, d) for $k \in \{1, \ldots, 5\}$, $d \in \{\text{maximum}, \text{euclidean}\}$, so the total number of configurations becomes $|k| \cdot |d| = 10$. In order to find the best solution the grid search algorithm would evaluate each of the ten configurations.

As every possible combination of hyperparameters has to be evaluated, this procedure suffers from a computational burden, which is prohibitive in many cases and with the presence of real-valued hyperparameters becomes unfeasible. On the other hand as subsequent evaluations are typically independent, the algorithm is easily parallelizable and enables the programmer to specify the order in which the configurations are evaluated.

1.2.2. Other Model-Free Search Methods

Other model-free search methods include random search, which can be employed when the parameter space size is unfeasible for grid search and algorithms based on *Iterated Local Search (ILS)* [Hut+09]. The algorithm paired with an error function performs a local search to quickly approach optimal solution. Perturbation phases are introduced to ensure that the search does not converge in a local minimum.

1.2.3. Bayesian Optimization

Bayesian optimization methods treat the machine learning algorithm as a black box function and, given an error measure, construct a probabilistic model by evaluating certain points in the hyperparameter space [EFH13]. An acquisition function associated with each Bayesian optimization algorithm uses the model to asses the useful information obtainable by evaluating each point of the hyperparameter space. This function maximized over the hyperparameter space is used to make "educated guesses" about which configuration should be evaluated next. A good acquisition function finds a good configuration by evaluating just a small part of the space. It is is especially beneficial in the context of machine learning, where many algorithms' iterations' time is measured in minutes rather than milliseconds). Bayesian optimization algorithms are able to outperform grid search and random search and, in some cases, human experts [SLA12; Tho+13].

As iterative model improvement is sequential in nature, those algorithms are more difficult to parallelize than grid or random search.

Spearmint Spearmint is a bayesian optimization method shown to work well with machine learning algorithms such as Support Vector Machines and Neural Networks [SLA12]. An open source implementation exists [Sno14], employing several acquisition functions configurable with a command line parameter and supporting continuous and discrete hyperparameters. It also offers a limited method of parallelization – several propositions for hyperparameter

configurations can be produced at once (based on previous iterations). In this case it is important to update the model as soon as each evaluation completes, as more information results in better propositions.

1.3. Present Approaches to Hyperparameter Tuning in Machine Learning

1.3.1. Holistic Approaches

Programming by Optimisation The article [Hoo10] proposes an approach called *Programming by Optimisation (PbO)* – a general method for exposing choices previously hard coded by the programmer as parameters and then using an automated parameter optimization algorithm in order to find the optimum solution for a problem. The article focuses on (although is not limited to) optimizing SAT-solvers and proposes an implementations based on an additional preprocessing phase for programs written in C. With syntax similar to preprocessor macros the programmer can specify parameters and a range of values each of them can assume directly in the source code and then use it like she would use a `#define` constant. Between each evaluation of different parameter configuration the program would need to be compiled again, with a different set of constants provided by the preprocessor (there called a *weaver*).

Experiment databases The authors of [BV07] emphasize not only the role of experiments in machine learning process, but also recording results of those experiments for future reference. They would be stored, along with detailed conditions of the experiment (e.g. summary of the data set, preprocessing methods) to be queried later by other researches. *Experiment database* could provide useful insights as to how certain algorithms perform under specific conditions and aid future experimentation, narrowing the parameter search scope for data sets with characteristics similar to the data sets recorded in the database. Authors then proceed with defining a schema for a relational database, emphasizing attributes that enable experiment reuse and repeatability. A database containing results of 250000 experiments is provided.

A universal framework for meta-learning MLAs [JG11] and HAL [Nel+11] are complete frameworks for conducting machine learning experiments. The entire machine learning pipeline is represented as a graph of blocks (or machines in MLAs) that exchange data between each other. Hyperparameter tuning is implemented using either ParamILS and racing procedures in HAL or grid search and its variations in MLAs. Designing more experiments is available via the framework's API. An interesting additional parameter associated with the machine learning algorithm in MLAs is complexity estimation. It is used to reorder evaluations during the optimization phase, executing simple algorithms first to ensure that an accurate solution is found as soon as possible. Because the order of execution is known beforehand, it is also possible to cache and reuse some of the framework's artifacts (in this case the machines). For this to work an accurate complexity estimation must be provided by the programmer.

MLbase [Kra+13] is yet another machine framework that allows hyperparameter optimization, but here, unlike the previous examples, an emphasis is put on the simplest possible interface instead of detailed experimentation record. API calls are as simple as

`var (fn-model, summary) = doClassify(X, y)` and the underlying execution and optimization engine is trusted to take care of delivering the best solution. Various heuristic approaches are employed to optimize hyperparameters for each class of machine learning algorithms and data sets of certain type. MLbase is not extensible – its usage is limited to the algorithms and heuristics implemented by the developers.

1.3.2. Extensions to Popular Machine Learning Libraries

Caret The Caret package [Kuh08] is a collection of tools for evaluating machine learning models in R. Among its numerous elements useful in designing a machine learning pipeline, it offers basic parameter tuning with a simple API that takes care of running multiple experiments with cross-validation. The optimization method is restricted to grid search and the machine learning algorithm must integrate with caret's ecosystem. Also, the algorithm choice itself cannot be the subject of tuning.

Auto-WEKA An extension to a well-established data mining package WEKA [Hal+09] for the JVM is descibed in [Tho+13]. It offers several procedures for tuning WEKA's algorithms, both model-free and based on bayesian optimization.

Chapter 2

Relaks

As we have seen in the previous chapter, experimentation is a crucial element of the machine learning process. Meta-learning is a useful framework for automating the hyperparameter optimization task, yet existing, production-ready tools treat it as an add-in rather than an integral part of the machine learning pipeline. We argue that putting it in the foreground of a machine learning tool induces good practice and removes a lot of boilerplate code, allowing the programmer to focus on what is unique for each problem - data and algorithms.

We propose Relaks – a *domain specific language (DSL)* embedded in Scala that reduces the complexity of hyperparameter tuning in machine learning algorithms. Meta-learning adds one more layer to a machine-learning program – it is essentially programming and reasoning about programs. Therefore, we postulate that it requires special treatment in order to make it understandable and more reusable.

At the center of Relaks there is a declarative syntax for specifying hyperparameter optimization scenarios. Optimizing an algorithm can be as simple as:

```
optimize (knnExperiment) by 'errorRate limit 10
```

As a result, 10 iterations of an optimization algorithm are run, exploring the parameter space of a knn classifier, looking to minimize the `errorRate` attribute. This expression would evaluate to a set of ten results of an optimization procedure represented by tuples – literally an experiment database (see 1.3.1)! The database can be further queried with syntax akin to *Language Integrated Query (LINQ)* from C# [BMT07] in order to further investigate the results.

As hyperparameter optimization is the main focus of Relaks, hyperparameters get special treatment as well. Defining a hyperparameter space has its own syntax:

```
val k = choose between 1 and 5
```

Here a parameter space for `k` is defined. Value `k` can then be used as any other value, and the `optimizer` automatically resolves it and adds its space to the optimization's scope.

Algorithms for automated hyperparameter tuning are integrated into Relaks. Model-free or model-based implementations can be swapped without making any other changes to the source code.

Choosing Scala as a host language instead of creating a compiler dedicated to Relaks removes the necessity to implement a parser and a type checker. Simultaneously Scala's type system and syntax is flexible enough to allow a highly customized DSL implementation. This choice also enables Relaks to leverage the rich set of libraries implemented for Java and Scala – the DSL is able to call functions available to the host language.

Relaks assumes that machine learning programs are usually a set of transformations applied to data. This assumption follows the popular perception of machine learning programs as data pipelines, employed by other frameworks, e.g. Spark [Zah+10] or knitr for R [Xie14]. Under this assumption programs are a composition of (often reusable) pure functions that apply transformations to data. Apart from bringing closer the ideal of reproducible research, it also allows further features to be implemented in Relaks, such as laziness and transparent caching of intermediate results.

The following chapters detail the implementation of Relaks. Chapter 3 offers an overview of state of the art methods for embedding a DSL in Scala. It details the basic features of Relaks DSL and their carrying out.

Chapter 4 describes the embedding of LINQ–like syntax in Scala, along with a fully static type system for the queries, made possible by heterogeneous lists and generic tuples.

Chapter 5 leaves for a moment the topic of DSLs and focuses on hyperparameter tuning algorithms. In Relaks two algorithms are available: grid search and Spearmint. They both have a common interface, powered by reactive streams.

Chapter 6 details the transformations applied to the internal representation of a Relaks program. It also describes Relaks' interpreter.

In chapter 7 additional extensions to the language are introduced, including a SQL data backend for Relaks' LINQ and a caching mechanism.

Finally, in chapter 8 we will see how the presented implementation handles optimization scenarios.

Chapter 3

DSL Basics

Let us begin with the general architecture behind DSLs and some terminology.

The main goal of a programming language developer is to create a process transforming an input in a formal language into a set of instructions for a computer. The set of instructions may take form of machine code in an executable file (be *compiled*) or be derived from a representation understandable by an interpreter, which then issues commands to the machine. Building internal representation (usually an *Abstract Syntax Tree (AST)*) is a common practice, regardless whether a compiler or an interpreter is involved.

DSLs differ from standalone programming languages in how an internal representation is computed. To understand them better we present a set of definitions introduced in [Tah99]. *Meta-program* is a program that manipulates other programs. In case of DSLs, the *meta-language* is used to generate and manipulate the representation of an *object-program*. The evaluation of a DSL is split into *stages*: generation-time, compile-time and execution-time for a compiled *object-language*, or generation-time and execution-time for interpreted implementations. Thus meta-program and object-program together form a *multi-stage* program, where object-program is *embedded* in the meta-program.

The meta-program contains representations of object-program's expressions, delaying their evaluation until later stages (*staging* them). Note that the run-time (generation-time in terms of the multi-stage program) of the meta-program is compile-time of the object-program. Values computed in that stage are static from the perspective of the object-program.

From the developer's point of view the main benefit of multi-stage programming is linguistic reuse [Kri01]. The object-language can take advantage of the meta-language's features such as scoping and a type system. Furthermore, a valid and type-checked meta-program will ensure syntax and type correctness of an object-program so the repetitive tasks of creating a parser and a type checker can be skipped. Finally, the multi-stage program can benefit from the ecosystem of the meta-language – the IDE and, if the meta-language and object-language target the same execution environment (as it is in the case of Relaks), allow the object-program to call meta-language constructs.

So why bother with implementing a DSL, if a lot of the meta-language features will be reused anyway? The direct access to the internal representation is not to be underestimated – it not only enables both staged-compile-time and run-time optimizations, but also allows creating semantics different than the semantics of the meta-language. How those traits are employed to the benefit of Relaks will be explained in later chapters.

3.1. The Meta-Language: Scala

Scala is a modern, statically typed language that targets the JVM. It offers full compatibility with the Java code, while implementing an interesting mix of functional and object oriented semantics. Its flexible syntax allows skipping dots and parentheses in an object's method call, effectively allowing method names to become operators. Simultaneously most special characters (e.g. +, -, =) are valid method names in Scala. Defining unary operators is enabled by implementing a `unary_*` method, where `*` is the desired operator. The syntax flexibility makes Scala especially attractive for embedding object-programs.

Scala-virtualized [Cha+10] for lightweight modular staging (LMS) [Rom12] enables even more flexible syntax – a developer can provide her own implementation for class constructors, variable definitions etc. While the theoretical foundations of LMS have had great impact on Relaks' implementation, ultimately we have not decided to base it on Scala-virtualized. At the time of writing the version of Scala it supports was falling behind the official branch and, above all, Scala-virtualized lacks support from Scala's most popular IDEs: Eclipse and IntelliJ, making it much harder to adopt.

Relaks relies heavily on other advanced features of Scala, such as implicit values and implicit conversions, covariance and higher order types. Those are described in more detail in [Ode11]. We will take a closer look at implicits in chapter 4.

3.2. Intermediate Representation

For convenient processing Relaks stores its *intermediate representation (IR)* as an Abstract Syntax Tree (AST). We will begin with introducing a few basic nodes and extend the representation in later chapters, where Relaks' additional functionalities are presented.

The architecture of Relaks makes it easy to incrementally add new features. A clear division is kept between the AST implementation, meta-language manipulation features and various phases of the interpreter.

The only types of nodes Relaks needs to explicitly represent are expression nodes. The basic structure is presented in listing 1.

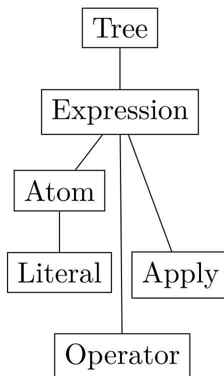
`Product` and `Cloneable` traits will be crucial in later stages – they enable a simple, generic rewriting mechanism for the IR. `Product` trait's main purpose is to provide a tuple-like interface to classes that are products of other classes. Case classes include a `Product` implementation generated automatically by the compiler. `Atom` is a superclass for nodes that are not composed of other nodes. `Leaf` is a mixin that provides similar information, but in the context of tree traversal. We will see the difference later in this chapter.

Statements traditionally seen in other programming languages, including e.g. `let` instructions, are handled by reusing Scala's features. Scope reuse is particularly useful and natural. By leveraging Scala's scoping rules we can ensure scope correctness and get powerful hygiene-related tools such as packages and blocks at no cost. As Relaks implements a no-side-effect language, other control structures like while loops are not present.

3.3. Embedding a Language

Let us take a look at a simple Scala program:

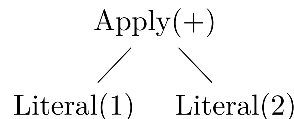
```
val a = 1
val b = 2
a + b
```




```
sealed trait Tree extends Product with Cloneable

trait Leaf extends Product {
  override def productElement(n: Int): Any
  override def productArity: Int
  override def canEqual(that: Any): Boolean
}

trait Expression extends Tree

trait Atom extends Expression

case class Literal(value: Any) extends Atom with Leaf {
  override def equals(other: Any): Boolean
}

sealed case class Operator(name: String) extends
↪   Expression

sealed case class Apply(fun: Expression, argList:
↪   List[Expression]) extends Expression
```

Listing 1: Basic IR nodes

We would like to evaluate `a + b` as an expression of our DSL. Therefore, we want to arrive at an AST representation:



The representation will later be input to an interpreter or a compiler.

The meta-program generates an object-program by manipulating expressions from the object-language. To ensure cross-stage correctness those must not be evaluated until the execution-time of the multi-stage program. But in our example the expression `a + b` will be simply evaluated to 3 at generation-time. In order to differentiate between meta-language (Scala) expressions and object-language (Relaks) expressions that should be evaluated at later stages while keeping a natural syntax, we need a wrapper for embedded expressions:

```
trait Rep[+T] {
  val tree: Expression
}
```

Where `tree` is the root of a AST represented by the wrapper. Next we define an implicit conversion of a Scala's Int to a representation of an integer in Relaks:

```
implicit def intToRep(x: Int): Rep[Int] = new Rep[Int] {
    override val tree: Expression = Literal(x)
  }
```

We also define basic operations for an integer representation:

```
implicit class IntOps(val arg1: Rep[Int]) {
  def +(arg2: Rep[Int]): Rep[Int] = new Rep[Int] {
    override val tree = Apply("+", List(arg1.tree, arg2.tree))
  }
}
```

We can picture implicit classes as on-demand mixins that can be added to certain classes during compile-time. Whenever there is a call to a method that is absent from some class' instance, the compiler looks for an implicit class (or an implicit conversion) that provides this method and transparently converts the instance (or inlines the method).

Now the following expression:

```
val a: Rep[Int] = 1
val b = 2
(a + b).tree
```

will evaluate to our desired AST representation. Note that `Rep`-types are viral, as the + operator requires a `Rep[Int]` argument, `b` will be converted automatically.

3.4. Type Reuse

Note, that in the previous example we are already reusing Scala's type system type to ensure type correctness: a `Rep[Int]` can be added only to another `Rep[Int]` (or an instance of a class that is a subclass of `Rep[Int]`)[1] By taking it further we can generalize `IntOps` to `NumericOps` that would work on doubles, floats, BigInts etc. and save ourselves some work. `NumericOps` header becomes:

```
implicit class NumericOps[T: Field](arg: Rep[T]) { ...
```

The rest is left unchanged. Here `Field` is a witness – an evidence that a `Field[T]` instance exists for some type `T`, so `NumericOps` can be applied to each type `Rep[T]` whenever an object of type `Field[T]` is present. This mechanism is called the *Type Class Pattern* and was first introduced in Haskell. It is used for adding ad hoc behavior (and interfaces) to types. `Field` comes from Breeze package for Scala [Hal15] and defines basic operations for the underlying type: division, multiplication, power etc.[2]

By modifying the conversion to the Literal's representation we have basic operations for all numeric types we would need to implement in Relaks covered. With not too much additional effort in a similar fashion we can implement `OrderableOps` for `Ordering` instances from Scala's standard package and basic boolean operations for `Rep[Boolean]`.

Calls to `==` are desugared by the Scala compiler to an `.equals()` method call, so we will not override it like the other operators. Instead a `===` method is implemented. Having two so similar methods may seem like asking for an avalanche of bugs, but as the return type of `==` is boolean, while for `===` it is `Rep[Bool]`. Therefore, the most bugs should be caught at the compile-time.

[1] `Rep[T]` is called a *phantom type*. A phantom type is a type whose argument type is not related to any component. Indeed, a `Rep[Int]` object does not contain an integer but only a representation of something that behaves a little like an integer.

[2] Although integers are not an algebraic field, Breeze's `Field` is defined for them as well, with integer division. According to the documentation "Not a field, but whatever."

3.5. Modular Architecture

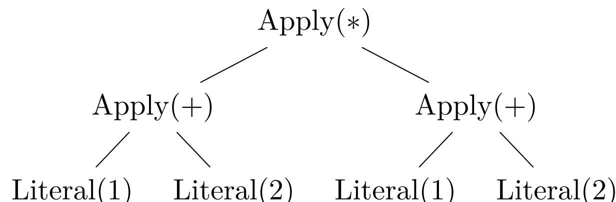
Note that in order to provide an additional functionality to representations of certain type we provide additional methods in independent implicit classes rather than directly extend the `Rep` class. The argument type `T` in `Rep[T]` effectively becomes the single source of truth regarding operations available to expressions of type `T`.

3.6. Drawbacks of the Pure Functional Approach

Consider the following example:

```
val a: Rep[Int] = 1
val b: Rep[Int] = 2
val c = a + b
c * c
```

IR built for this example will be larger expected:



Expression `a + b` would be evaluated twice, following a call-by-name policy, while a call-by-value is preferred. While this still produces a correct program (under the assumption of functional purity), it clearly is suboptimal and would cause serious performance penalties in case of more complicated expressions. A frequent remedy for code generation DSLs mentioned in LMS [Rom12] is let insertion – inserting let bindings in strategic places. Since Relaks does not perform code generation at the time and its representation does not include blocks or statements, we propose a slightly different approach based on the foundation laid out in LMS.

3.6.1. Graph Intermediate Representation

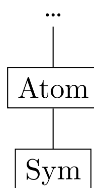
From the tree we will switch to a directed acyclic graph (DAG) IR. We define a following trait extending DSL's functionalities.

```
1  trait Symbols {
2    class Sym extends Atom {
3      val name: Int
4      override def productElement(n: Int): Any
5      override def productArity: Int
6      override def canEqual(that: Any): Boolean
7      override def equals(other: Any): Boolean
8      override def hashCode() = name.hashCode()
9    }
10
11   private var definitions = Map.empty[Sym, Expression]
12   private var symCounter = -1
```

```
13
14   object Sym {
15     def apply(name: Int): Sym
16   }
17
18   protected final def fresh: Sym = {
19     symCounter += 1
20     Sym(symCounter)
21   }
22
23   protected final def freshRep[T](typ: TType): Rep[T]
24
25   private def saveDefinition(sym: Sym, expression: Expression) : Unit = {
26     definitions += ((sym, expression))
27   }
28
29   protected def findDefinition(sym: Sym) : Option[Expression]
30
31   implicit final def toAtom(expression: Expression) : Atom =
32     expression match {
33     case sym: Sym => sym
34     case _ =>
35       val sym = fresh(expression.tpe)
36       saveDefinition(sym, expression)
37       sym
38   }
39 }
```

Listing 2: Symbols

We begin by defining a new IR node.



Symbol (abbreviated `Sym`) represents a "symbolic link" to an expression. Associations are stored in a hash map (line 11.) for efficient retrieval. It is identified by an integer "name". Each time a new symbol is created a global counter is incremented (lines 18. - 21.) so that the names are unique. Additional implementations of methods from the `Product` trait are there to simplify IR traversal – accessing a symbol would transparently follow its symbolic link. By making a small modification to `Rep`'s definition from section 3.3 we can force `Sym`s to be created for each new `Rep`:

```
trait Rep[+T] {
  val tree: Atom
}
```

No additional effort from the programmer's perspective is required – from now on each expression that is not an `Atom` will be automatically converted to a `Sym` by an implicit function (line 31. of listing 2).

For convenient pattern matching on the IR nodes we define the following object:

```
object /> {
  def unapply(expr: Expression): Option[(Option[Sym], Expression)] = {
    val symOpt = expr match {
      case s: Sym => Some(s)
      case _ => None
    }
    findDefinition(s).map(followed => (symOpt, followed))
  }
}
```

Scala's compiler desugars match expressions to calls to an `.unapply()` method. Whenever that method returns an option of a two-element tuple, it can be used as an operator. Using our object pattern matching a symbol linking to an expression has a pleasant syntax:

```
{ case Some(sym) /> expr => ... }
```

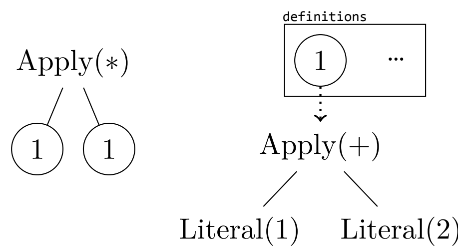
Whenever we do not care about the Symbol (and that is often the case) it becomes even simpler:

```
{ case _/> expr => ... }
```

Therefore we can still use companion objects generated by IR nodes' case classes with just a little additional effort.

3.6.2. Advantages of the Graph Representation

Our representation from the beginning of section 3.6 will now look like this:



We will solve the call-by-name/call-by-value problem while implementing an interpreter in chapter 6. Thanks to Symbols it will be possible to rewrite the definitions map, replacing arbitrary expressions with partially computed results, while interpreting the IR. This way all subsequent references to the same symbol will not evaluate the linked subtree again.

The common subexpression elimination problem can be solved with just a little modification to the `Symbols` trait. By adding a reverse hash map `Expression` → `Sym` to the code in listing 2 we can implement a `findOrReplace(e: Expression)` method that does not allow to link the same expression with two different symbols. Expressions will be hash-compared – in the case of `Product` subclasses two instances' hashes are equal if and only if their elements' hashes are equal.

3.6.3. Modular Architecture One More Time

In order to avoid polluting the global scope with mutable objects, the code in listing 2 is wrapped in a `Symbols` trait. Therefore any code making use of the object and classes defined within this trait must import it into its parent scope. By applying this idea to other extensions and modules we can create a set of traits that the developer can mix in at will in order to import desired functionalities. This design allows for easier testing, higher extensibility and, with the help of Scala's visibility modifiers (`private`, `protected`, etc.) reduced pollution of the scope. Whne, for example, the numeric and ordering extensions described in section 3.4 are needed, the following pattern can be used:

```
object Program extends NumericExtensions with OrderingExtensions
import Program._
```

Chapter 4

Queries in Relaks

SQL is the lingua franca of databases and data warehouses. Its declarative paradigm allows for abstracting over implementation details and concentrating on the desired outcome of a query. And for many databases the implementations differ in more than details – the same queries can be run on an in-process embedded database sqlite [HK07] and software such as Spark [Zah+10] or Pig [Apa15] running on clusters of computers.

We find declarative query approach well suited to performing optimization experiments. First, the implementation details of an hyperparameter optimizer can be hidden. With a reasonable common interface, optimizers can be swapped in and out at will to perform different optimization scenarios. Second, not concentrating on details of how the experiments are performed increases developer's productivity without sacrificing too much performance. Optimizations such as parallelization and caching can be implicitly provided by the query language's representation. Third, a programmer would often want to explore more than just the "best" classifier, the top result of an optimization. She might be interested in a an algorithm that performs best under certain resource requirements such as time or required processing power and memory, factors that are not easy to introduce in an error function. The insights derived from the full record of an experiment can be useful too – see [BV07] and section 1.3.1 on Experiment Databases. Treating experiments and their results as databases makes them an easy target for analysis using SQL queries, a task and a language many developers are already familiar with.

Relaks' implementation of LINQ focuses on hyperparameter tuning experiments, but it is not its only purpose. In later chapters we will present an extension that will allow Relaks to generate some SQL code and submit queries to database engines.

4.1. LINQ

> Ray Boyce and I hoped that SQL would have an impact on the database industry, but its impact did not come in the way we expected. Ray and I thought that we were developing a language that would be used mainly by "casual users" to pose ad hoc queries for decision support applications. We were trying to make databases accessible to a new class of users who were not trained computer scientists. We expected to see SQL used directly by financial analysts, urban planners, and other professionals who needed access to data but did not want to write computer programs. These expectations proved to be too optimistic.
>
> *D. Chamberlin, one of the authors of SQL [Bia+09]*

Language Integrated Query (LINQ) [MBB06] was first introduced by Microsoft in 2007 to C#. Its main goal was to bridge the gap between a general-purpose programming language and querying the database. Originally a string containing the SQL command would be sent to a database and then a wrapping library such as JDBC would be used to retrieve the results. This approach proved to have many flaws – it was necessary for programmer to learn and use two languages instead of one, processing results retrieved that way was prone to bugs, type information was lost etc. Furthermore serious security flaws such as SQL injections [HVO06] were exposed.

Object-relational mapping frameworks were introduced in order to better associate database entities with their representation in the host code – the SQL syntax was replaced with specific framework's syntax.

LINQ's approach is different – its goal is to bind SQL-like syntax to a similar representation available natively in the host language and later translate it to a target language, such as SQL or GPU code. Target code would then be executed and results wrapped in a native collection type.

4.1.1. LINQ in Scala

Since its introduction LINQ has been implemented in many programming languages, including Scala. Slick [Typ15; Vog11] is among the most popular. It offers a type safe translation to queries for most popular SQL databases (such as PostgreSQL and MySQL) along with execution and results fetching. Slick also includes support for queries modifying the database, e.g. INSERT and UPDATE. A sample Slick expression looks like this:

```
coffees.map(_.name).filter(_.price < 10.0)
```

Listing 3: Slick Query, source: [Typ15]

It is equivalent to a SQL query:

```
select NAME from COFFEES where PRICE < 10.0
```

However, the developer is required to provide an explicit schema specification, akin to an object-relational mapping:

```
class Coffees(tag: Tag) extends Table[(String, Double)](tag, "COFFEES") {
  def name = column[String]("COF_NAME", O.PrimaryKey)
  def price = column[Double]("PRICE")
  def * = (name, price)
}

val coffees = TableQuery[Coffees]
```

Automatic schema information generation is possible – it involves code generation and requires connection to the database at compile-time.

After careful consideration we have decided to implement a LINQ version tuned to the specific need of Relaks for a number of reasons.

First, slick's verbose schema definitions would be a poor choice for queries on objects managed by the JVM – this is the case with Relaks' experiment specification. Those objects'

types are already known by the compiler – as we will see later in this chapter schemas can be figured out automatically.

Second, fully typed schemas are not well suited for machine learning datasets either. As mentioned before, Relaks' LINQ can be used to query database engines. Machine learning data sets, this feature's main area of interest, are unlike application databases targeted by Slick. Dimensionality of data sets for machine learning can be very high. Explicitly naming and typing tens of columns is unfeasible. Instead, Relaks allows working with partially typed data sets and mapping subsets of columns to vectors.

Third, Slick's focus is on cooperating with databases. Its unique features like handling NULLs and database management queries are not required by Relaks' immutable and fully typed with Scala's type system data sets. Furthermore, while the LINQ frontend API is rich and well-documented, its backend, including the IR, would be difficult to extend beyond query generation, especially considering the hyperparameter optimizers' interface introduced in chapter 5.

4.2. Implementation in Relaks

As mentioned earlier, Relaks' LINQ implementation can be used both to construct an experiment and query a database. In all cases mostly the same set of IR nodes and manipulation techniques is used. As we have have established a framework for optimization just yet, in the following sections we will use an example database to better explain presented concepts. It consists of two tables, stored in CSV files on a disk:

COF_NAME	SUP_ID	PRICE
Colombian	101	7.99
French_Roast	49	8.99
Espresso	150	9.99
Colombian_Decaf	101	8.99
French_Roast_Decaf	49	9.99

(a) coffee.csv

SUP_ID	SUP_NAME	ZIP
101	Acme, Inc.	95199
49	Superior Coffee	95460
150	The High Ground	93966

(b) suppliers.csv

Table 4.1: An example database [Typ15]

4.2.1. Staging Tuples

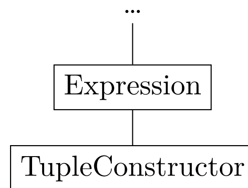
Since we want statically typed queries without explicitly providing an object mapping, we would need another abstraction over tables and rows. We propose (heterogeneous) tuples as the representation.

Scala's standard library does not provide any abstraction over tuples' arity. Tuples with different number of elements belong to distinct classes: `Tuple1`, `Tuple2`, …, `Tuple22` and their common superclass is `Product`. Not only tuples inherit from `Product` – another example are generated case classes.

Traditional way of working with tuples in general was either code generation (e.g. see [Twi15a]) or the mundane act of writing code for each of the 22 tuple classes. We, however, are going to use the shapeless library for Scala [Sab+15] that, among others, introduces heterogeneous lists (*hlists*), type-preserving operations on those lists and means of converting back and forth between them and Tuples of arbitrary arity.

We begin by extending the IR with a new node:

```
sealed case class TupleConstructor(tuple: Vector[Expression], names:
↪  Vector[String]) extends Expression
```



And this is everything we need! As we will see shortly, accessing the tuple's elements will be performed at compile-time of the meta-language.

Our tuples will have names – a useful feature for representing rows in a table.

In order to represent a `TupleConstructor` expression we are going to need new types.

```
sealed trait Tuple
sealed trait Tup[+H <: HList] extends Tuple
sealed trait UntypedTup extends Tuple
```

`Tuple` and its subclasses are *empty types*. They do not provide an implementation – they will never be instantiated and the only information they convey is their type. We do not want to associate them with a concrete table row container implementation – at least not just yet.

We define operations on typed tuples:

```
trait TupleExtensions extends Symbols {
  implicit class TupleOps[B1 <: HList](val arg1: Rep[Tup[B1]]){
    def apply[T](i: Nat)(implicit toInt: ToInt[i.N], at: At.Aux[B1, i.N, T]): Rep[T]
  }
}
```

`.apply()` can extract a correctly typed representation of tuple's element along with an expression from a `TupleConstructor` node. `Nat` is a type-level natural number used by shapeless. It can be automatically converted using macros from an Int literal[1] during compile-time. The `Nat` type can then be used to get the type of a correct element from the hlist present in tuple representation's type.

We also define an object useful for pattern matching and "unpacking" tuples of arbitrary length. Its signature is a lot more complicated – at this level of Scala's type system exploitation using implicits is like programming a simple logic program[2] with a not-so-friendly

[1] By an "Int literal" we mean an integer explicitly written in the source code. For example just `(1)` is such literal, but `(1 + 1)` or `val a = 1; (a)` are not. This is a requirement of the macro system - concrete values have to be available at the compile-time.

[2] In fact it is even possible to specify negation using implicits. Yet Scala's type system, athough powerful, is not equivalent to logic programming lanuages such as Prolog. For more information see http://kralikba.web.elte.hu/implicits0.html#there-is-no-prolog-in-your-scala

syntax. Therefore, from now on we will not post them in full, but rather concentrate on their contribution to Relaks' framework.

```
object Tup {
  def unapply[T <: HList, ..., TUPL <: Product](r: Rep[Tup[T]])(implicit ...):
  ↪  Option[TUPL]
}
```

`TUPL` is a Scala Tuple corresponding to `T` hlist type. The method extracts all expressions from the IR node and can be used in a friendly syntax for unpacking every element in a tuple with correct types:

```
val tuple: Rep[Tup[Int::String::Boolean::HNil]] = ...
val Tup(one, two, three) = tuple
```

The last element is lifting Scala's tuples to an IR, similar to lifting Scala's integers to `Literal` nodes in section 3.3. Again, an implicit conversion method is provided.

One of its important features is unifying the "shapes" of tuple's types. Tuples staged into Relaks' object-language can be of mixed types, some elements already `Rep`s and some requiring a conversion. In order to construct the IR node we need all elements to be lifted into representations. Additionally, we would not want distinct tuple types like `Tup[Int::Int::HNil]` and `Tup[Rep[Int]::Int::HNil]`, since they essentially mean the same thing. What is more, Scala's compiler is unable to perform a two-step implicit conversion, so our staging method must account for objects that are already lifted intro representations and those that are plain Scala values.

To solve this, first we use shapeless' polymorphic functions to map all elements of the tuple to `Rep`s.

```
object asRep extends Poly1 {
  implicit def lifted[T <: Rep[_]] = at[T](x => x)
  implicit def unlifted[T](implicit c: T => Rep[T]) = at[T](x => c(x))
}
//example:
implicitly[Mapper.Aux[asRep.type, Rep[Int]::Int::HNil, X]] //X becomes
↪  Rep[Int]::Rep[Int]::HNil]
```

Implicit conversions are used to perform the mapping, including a special identity conversion, when the object already is a representation. Whenever an implicit conversion is not present, Relaks' tuple construction fails – some of the tuple's element cannot be automatically staged.

Then, for each element type of the output hlist, `Rep[T]` becomes just `T`.

```
trait UnliftType[-L <: HList, M[_]] {
  type Out <: HList
}
object UnliftType {
  def apply[L <: HList, M[_]](implicit unlift: UnliftType[L, M]) = unlift
  type Aux[L <: HList, M[_], R <: HList] = UnliftType[L, M] { type Out = R }

  implicit def unliftType[H, L <: HList, R <: HList, M[_]]
  (implicit ev: Aux[L, M, R]) :
  Aux[M[H] :: L, M, H :: R] =
```

```
    new UnliftType[M[H] :: L, M] { type Out = H :: R }

  implicit def unliftNoType[H, L <: HList, R <: HList, M[_]]
  (implicit ev: Aux[L, M, R], e: H =:!= M[T] forSome { type T }) :
  Aux[H :: L, M, H :: R] =
    new UnliftType[H :: L, M] { type Out = H :: R }

  implicit def unliftNil[M[_]]: Aux[HNil, M, HNil] =
    new UnliftType[HNil, M] { type Out = HNil }
}
//example:
implicitly[UnliftType.Aux[Rep[Int]::Rep[Int]::HNil, X]] //X becomes Int::Int::HNil
```

Taking this step after conversion to `Rep`s is important from the perspective of type declarations' conciseness. We could as well type `Tup[Rep[Int]::HNil]` but omitting the `Rep` makes the whole expression more compact. Note that this is a "weird" operation from an objective language programmer's perspective. It makes sense only in the case of phantom types.

It is possible to either stage a plain Scala tuple and automatically assign the names or define names using an "as" syntax.

```
class TupleField[T](val t: T, name: Symbol)
class AnyAs[T](val t: T) extends AnyVal {
  def as(name: Symbol) = new TupleField(t, name)
}
//example
(value1 as 'val1, value2 as 'val2)
```

Scala's `Symbol` here is not to be confused with Relaks' `Sym`. `Symbol` is essentially a String, the only difference being that it can be defined with just a single apostrophe, which makes for a more concise syntax.

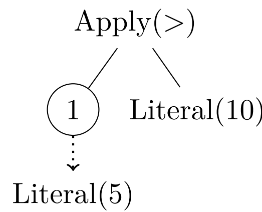
The implicit conversion method for staging tuples of `TupleField`s collects the names and then performs the same operations as the previous conversion function. For consistency the plain tuple conversion function automatically specifies placeholder names for the tuple's elements. These are not meant to be used directly – whenever a developer should need to access tuple's element by name, she is supposed to assign it a meaningful name.

4.2.2. Peeking Inside Functions

Before we can implement LINQ, we will need one more abstraction. All operators from Scala's standard collection library take a higher order function as an argument. Filtering is performed with a function of type `T => Boolean`, mapping takes a `A => B` etc. In order to mimic this functionality we would need a way of working with higher order functions in Relaks. Unlike the previous extensions we will not be staging functions directly. The idea is to reuse Scala's function instead. To give an example:

```
val f: Rep[Int] => Rep[Boolean] = i => i > 10
```

Note that a staged function would have have a `Rep[Int => Boolean]` type. What happens when we call this function? An example call `f(5)` has the following representation:



Where ① is a `Sym` and other `Sym`s were skipped for clarity. The function is "inlined" during the meta-language's run-time. Calling the function with a *fresh* (not assigned to anything) `Sym` gives us a generic representation that may be later filled in with a value when we link something to the fresh `Sym`. By reusing Scala's functions we are getting higher order functions in our meta-program at almost no cost. In order to be able to call them we would need to store just the inlined representation and the name of the `Sym`(s) used to inline it. A class called Generator, which will store those `Sym`s will be introduced in the next section.

On a side note, recursion is problematic – inlining a recursive function at the meta-language run-time would cause stack overflow. Fortunately, we will not be needing recursion in our LINQ implementation.

4.2.3. Staging Queries

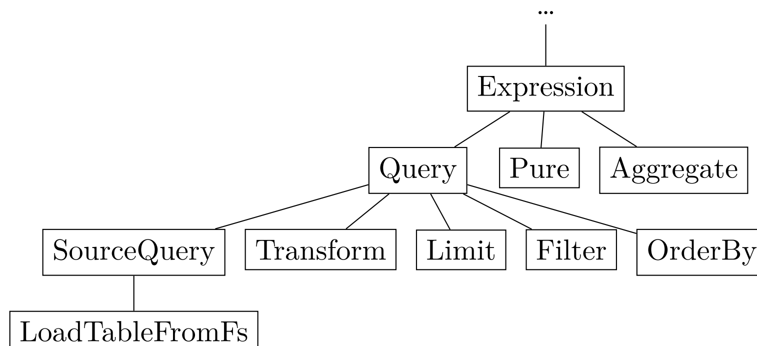
With everything in place we can finally introduce a set of operations on collections of tuples – tables. Our implementation includes only a subset of the features of LINQ. Relaks supports queries created with comprehensions (projection and filtering), sorting, limiting and skipping records and very simple aggregations.

As usual, we will begin by introducing a few more IR nodes.

```
sealed trait Query extends Expression
sealed trait SourceQuery extends Query

trait GeneratorBase

sealed case class LoadTableFromFs(path: String) extends SourceQuery

sealed case class Transform(generator: GeneratorBase, table: Atom, select: Atom)
 ↪  extends Query
sealed case class Limit(table: Atom, start: Atom, count: Atom) extends Query
sealed case class Filter(generator: GeneratorBase, table: Atom, filter: Atom)
 ↪  extends Query

object OrderBy {
  sealed trait OrderDirection
  object Asc extends OrderDirection
  object Desc extends OrderDirection
}

sealed case class OrderBy(table: Atom, ordering: Vector[FieldWithDirection])
 ↪  extends Query

sealed case class Aggregate(fun: Aggregator, query: Atom) extends Expression
```

```
sealed trait Aggregator
object Aggregator {
  object Avg extends Aggregator
}

sealed case class Pure(value: Atom) extends Expression
```

Listing 4: Query IR Nodes



We will present them, along with extensions used for IR manipulation one by one. We will also need new empty types to represent LINQ expressions:

```
sealed trait Table
sealed trait UntypedTable extends Table
sealed trait TypedTable[+T] extends Table
```

Sources

`SourceQuery` is a base class for representations of tuple producers. `LoadTableFromFs` is one such producer. It represents the operation of reading a sequence of tuples from a filesystem source. To read table 4.1a we would use the `load` method:

```
trait TableDefs extends Symbols {
  def load(path: String): Rep[UntypedTable]
}
//example
val coffeeTbl = load("coffee.csv")
```

Note that the schema of a table in a CSV file is unknown. Therefore, the returned representation is of `UntypedTable` type. Then, we can either provide a schema for our table and apply fully typed operations to the representation, or use one of the untyped or partially typed manipulators.

Schema

An implicit conversion for objects of `Rep[Table]` type (which includes both `Rep[UntypedTable]` and `Rep[TypedTable[...]]`) defines an `.apply()` method used to provide a schema and generate an implicit projection. The method takes a tuple of `TypedField[T]` objects as an argument.

The following query represents a fully typed table 4.1a.

```
sealed case class Field(sym: Symbol)
final class TypedField[T](sym: Symbol) extends Field(sym)
final case class FieldWithDirection(field: Field, direction: OrderBy.OrderDirection)
//example
val tf: TypedField[Double] = 'price.::[Double]
```

Listing 5: `Field` and `TypedField` definitions. `FieldWithDirection` is used by the `OrderBy` IR node

```
val typedCoffee: Rep[TypedTable[Tup[String::Int::Double::HNil]]] =
 ↪  coffeeTbl('COF_NAME.::[String], 'SUP_ID.::[Int], 'PRICE.::[Double])
```

Type arguments are extracted from `TypedField` objects and used to construct a hlist and then a proper `Tup` type. A `TypedTable Rep` is returned, representing a simple projection, mapping table's columns according to the provided schema.

Transformations

Transformations are operations that apply a mapping function to each tuple in a table. Here a projection is a special case of a transformation where the mapping function is a permutation of a (possibly improper) subset of the input tuple.

```
1  trait TableExtensions extends Symbols {
2    class Generator(val symsToFields: Map[Sym, Symbol], syms: Option[Vector[Sym]] =
      ↪  None) extends GeneratorBase
3
4    trait TupleGeneratorImpl {
5      protected def tupleGenerator[F <: HList](fields: Vector[Field]): (Generator,
        ↪  Rep[Tup[F]])
6    }
7
8    implicit class TypedTableOps[H <: HList](arg: Rep[TypedTable[Tup[H]]])
9      extends TupleGeneratorImpl {
10     lazy val fields: Vector[Field] = outputFields.apply(arg.tree)
11
12     def flatMap[T <: HList](f: Rep[Tup[H]] => Rep[TypedTable[Tup[T]]]):
        ↪  Rep[TypedTable[Tup[T]]] = {
13       val (gen, tuple) = tupleGenerator(fields)
14       val inlined = f(tuple)
15       val expr = Transform(gen, arg.tree, inlined.tree)
16       new Rep[TypedTable[Tup[T]]] {
17         override val tree = expr
18       }
19     }
20
21     def map[T <: HList](f: Rep[Tup[H]] => Rep[Tup[T]]) = {
22       flatMap(f andThen SingleRow.apply)
23     }
24   }
```

```
25
26    object SingleRow {
27      def apply[T <: HList](t: Rep[Tup[T]]): Rep[TypedTable[Tup[T]]]
28    }
29  }
```

First we define the `Generator` class (see section 4.2.2). As the `BaseGenerator` was defined in the global scope, it had no way of accessing the `Sym` type. Here we provide a concrete representation – a mapping from Syms used to inline a function to their names in a corresponding tuple with an optional Vector specifying their order.

`.flatMap()` (line 12.) is the basic function for manipulating collections and containers in Scala. Unlike `.map()` it expects the mapping function to explicitly create a new container instance – such as a nested query in our context.

In our implementation a Generator and a corresponding (staged) tuple is created first. The tuple is then supplied as an argument to the provided mapping function (here `f`), creating an inlined representation, with arguments as "holes" to be later filled in by linking the Syms stored in the Generator. A new `Transform` node is returned.

In order to construct a Generator we need to traverse the IR searching for the last `Transform` node – the code in line 10. returns a list of Fields that are currently available. IR traversal and attribution will be covered in chapter6. On a side note, supplying the list of Fields would also be possible by extending the `Rep` class and passing the list along when creating a new `Rep` object. We find our design decision superior though – having the IR as the single source of truth contributes to greater clarity and higher maintainability of the code.

`.map()` (line 21.) can be defined in terms of `.flatMap()`. `SingleRow.apply()` wraps a tuple IR in a `Pure` node, returning a `TypedTable` representation.

The following query returns a table of coffee blend names with corresponding prices, including a (moderate) vendor margin.

```
typedCoffee.map({ case Tup(name, _, price) => (name as 'blendName, (price * 1.2) as
↪  'price) })
```

Alternatively we can also write

```
typedCoffee.map(row => (row(0) as 'blendName, (row(2) * 1.2) as 'price))
```

Filtering

We further extend the `TableExtensions` trait.

```
  class ProjectionFilter[In <: Rep[Table], Out](arg: In)(implicit asCmp:
  ↪  BuildComprehension[In, Out])
  extends TupleGeneratorImpl {
    def filter[P <: Product, F <: HList](fields: P)(f: Rep[Tup[F]] =>
    ↪  Rep[Boolean])(implicit ...): Out
  }
```

`.filter()` from `ProjectionFilter` is able to work with Typed and Untyped tables alike. As the first argument it takes a subset of fields to which the filtering function is applied. Then it generates the proper correct type similarly to the schema specification function, computes a representation of the filtering function and returns a `Filter` node wrapped in

a representation of the same type as the current expression. `ProjectionFilter` always returns a table of the same type. The signature is a little more complicated than in previous cases: we need a `BuildComprehension` factory that creates a new representation of type `Out` given an IR node. We choose which factory to fetch based on the type of current expression. Instances of type `BuildComprehension[Rep[UntypedTable], Rep[UntypedTable]]` and `BuildComprehension[Rep[TypedTable[Tup[H]]], Rep[TypedTable[Tup[H]]]]` for each hlist `H` are implicitly defined.

Of course a fully typed version of `.filter()` is provided as well. Its implementation is very similar to the untyped version.

The following query looks for coffee provided by a specific supplier, with prices below 8.0.

```
coffeeTbl.filter('SUP_ID.::[Int], 'PRICE.::[Double])({case Tup(sup, price) => sup
↪  === 101 && price < 8.0})
```

Its typed alternative would look much alike.

```
typedCoffee.filter({case Tup(_, sup, price) => sup === 101 && price < 8.0})
```

The version working on Untyped tables really shines in the case of data sets with many columns, where specifying the schema would be very time-consuming. For example, in such case the LINQ syntax can be used to do some initial filtering on one or two attributes and then the remaining data would be input to a pipeline more flexible with respect to the types.

Sorting and limiting

Two more additions to the `TableExtensions` trait are introduced, with semantics similar to the `ProjectionFilter.filter()` method.

```
implicit class LimitOps[In <: Rep[Table], Out](arg: In)(implicit asCmp:
↪  BuildComprehension[In, Out]) {
  def limit(count: Rep[Int]): Out
  def limit(start: Rep[Int], count: Rep[Int]): Out
}

implicit class OrderByOps[In <: Rep[Table], Out](arg: In)(implicit asCmp:
↪  BuildComprehension[In, Out]) {
  //orderBy's argument is a tuple of FieldWithDirection elements
  def orderBy[P <: Product](fields: P)(implicit ...): Out
  def orderBy(field: FieldWithDirection): Out = orderBy(Tuple1(field))
}
```

Simple aggregations

At this moment only the average function is implemented as an aggregation.

```
implicit class AvgTableOps[T](arg: Rep[TypedTable[Tup[T :: HNil]]])(implicit conv:
↪  Injection[T, Double]) {
  def avg: Rep[Double] = {
    new Rep[Double] {
      val tree: Atom = Aggregate(Aggregator.Avg, arg.tree)
    }
  }
}
```

Unlike previous functions, `.avg` returns a single value instead of a table. `Injection[T, Double]` type class is provided by Twitter's Bijection library [Twi15b]. It witnesses a conversion between type `T` and a Double.

For-comprehension syntax

Supporting methods `.map()`, `.flatMap()` and `.filter()` is enough to take advantage of Scala's for-comprehension syntax.

The code below is simply desugared to the previously seen vendor margin example.

```
for (row <- typedCoffee) yield (row(0) as 'blendName, (row(2) * 1.2) as 'price))
```

Note that more complicated queries are also possible. Specifying nested queries seems particularly natural with this syntax.

```
val suppliers = load("suppliers.csv")('SUP_ID.::[Int], 'SUP_NAME.::[Int])
for {
  coffee <- typedCoffee
  supplier <- suppliers if (supplier(0) === coffe(1))
} yield (coffe(0), supplier(1))
```

The returned representation includes a nested query and a filter node, as expected. Relaks' SQL interpreter implementation does not use hash-joins or indexes, therefore evaluating the nested query would take $\mathcal{O}(m \times n)$ where m and n are the sizes of tables 4.1a and 4.1b. Naively submitting such queries to a SQL backend would cause a query avalanche [Sch+10], resulting in an amount of queries bound to the size of the data. Two solutions to this problem will be proposed later, one leveraging the Sym cache/store system (section 7.3), and the other involving more thoughtful SQL code generation (section 6.3).

For-comprehension desugaring process has one quirk.

```
for (row: Rep[Tup[String::Int::Double::HNil]] <- typedCoffee) yield (row(0) as
↪  'blendName, (row(2) * 1.2) as 'price))
```

Whenever we explicitly provide a type for a row, either for clarity or to help the IDE, an `isInstanceOf` call is included. The code in the last example will be desugared to

```
typedCoffe.withFilter(_.isInstanceOf(Rep[Tup[String::Int::Double::HNil]])).map(...)
```

Fortunately, an implicit conversion function for staging for boolean values should be present, so the argument of `withFilter` will have the correct `[Rep[Tup[...]] => Rep[Boolean]` type. The `isInstanceOf` will be performed at the multi-stage program compile-time though, resulting (most likely) in an unnecessary filter operation with a Literal(true) condition. It should be removed either as a special case of the filtering function, or in a constant folding phase of the multi-stage program.

4.2.4. Improving the Syntax

It is possible, using the shapeless library and dynamic objects in Scala, to have a fully statically typed, call-by-column-name version of the tuple representation. Shapeless implements Records, a heterogeneous version of Map class from the standard library, allowing the user to specify keys for the Record like she would input String or Symbol keys for a Map. A schema or names specified on a tuple could be translated to properties (accessed with the dot syntax)

on a dynamic object, resulting in LINQ queries looking similar to Slick's without the hassle of defining object mappings.

An experimental implementation proved that, although valid and compilable, this syntax has absolutely no support from the leading Scala IDEs (IntelliJ and Eclipse). Their macro and dynamics support is limited and can sometimes cause problems even in the less demanding implementation presented in this chapter. Furthermore, according to the shapeless' GitHub pages [Sab+15], even fairly small Records carry large performance penalties. Even though those are compile-time and do not affect the running program, the possibility to iterate quickly through different versions of a solution or implement an interactive version of Relaks in the future would be greatly diminished. Therefore, we have decided to keep the little less pretty but more tool-friendly syntax, at least until the macro and dynamics issues are resolved.

Chapter 5

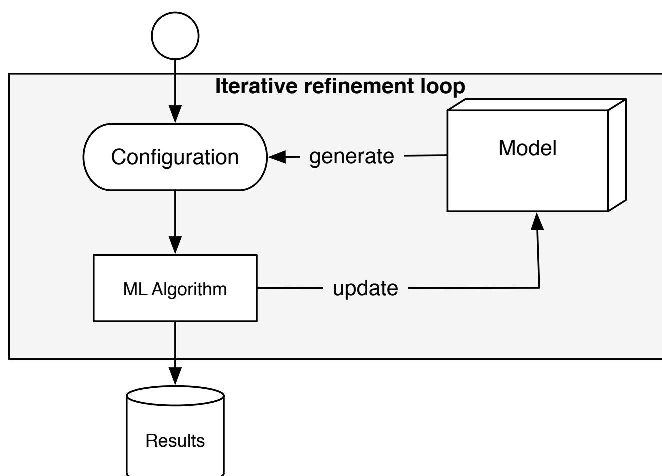
Hyperparameter Tuning

In the previous chapters we have looked into the compile-time phase of the multi-stage program. The following sections will be a short break from the IR representation and manipulation methods. We will take a closer look at hyperparameter tuning algorithms, implement a framework in pure Scala for working with them and prepare for plugging it into Relaks.

5.1. Hyperparameter Tuning as Iterative Refinement

Bayesian optimization for hyperparameter tuning described in section 1.2.3 is a promising method of efficiently finding a good configuration [SLA12; EFH13]. In our tuning framework we will concentrate on these methods.

Improving the hyperparameter space model can be viewed as an iterative refinement process – with each iteration the model becomes more accurate at portraying the reality being modeled. The underlying procedure is fairly basic: acquire a new set of hyperparameters, evaluate the configuration, update the acquisition function, repeat.



Moreover, an iterate-update loop can be used to describe some model-free tuning methods as well. For example, for the grid search algorithm the update function would be just a no-op and configuration acquisition would simply return the subsequent elements of all hyperparameters' cartesian product.

5.2. Hyperparameter Tuning as a Stream

The iterative tuning process generates a configuration of hyperparameters at each iteration. Processing configurations (evaluating the ML algorithm and computing error) is incremental – the entire list of settings is (generally) not known beforehand. Ideally the model should always be updated before the next set of parameters is generated – the quality of the configurations suggested for evaluation depends on how much information about the hyperparameter space has been gathered. Additionally, the tuning process can be stopped at any point, e.g. when the user decides the configurations are good enough, too much time has been spent or subsequent evaluations' improvement is starting to get really small. At this point the iterative process should be halted immediately and any allocated resources should be freed.

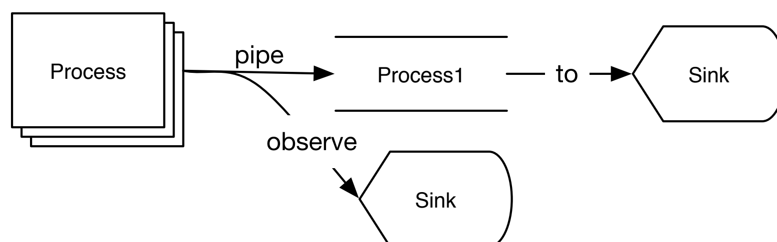
All those requirements are met by *streams with back pressure*, a functional solution for working with "live" streams of data. In Relaks' implementation we will be using scalaz-stream [Chl+15] library, which supports asynchronous I/O, concurrency, resource safety and advanced stream manipulation and composition.

5.3. Stream Processing Basics

The basic type in scalaz-stream is `Process[F, A]`. It represents a (possibly infinite) stream of `A`s wrapped in a monadic context provided by monad `F[_]`. Most often we will have `Task` provided by scalaz as our monad. `Task`'s monadic context is that of an asynchronous operation that can fail (throw an exception) – having something of type `Task[A]` means that `A` may or may not exist yet, and if it is not present, the operation has either failed or not finished yet. `Task` is similar to `Future` from Scala's standard library, the main difference being that unlike `Future`'s, `Task`'s results are not memoized, therefore the computation will re-run each time the `Task` is run.

A stream that produces no output is called a *sink*. We use it when we are interested only in an effect it produces, for example writing each element it receives from a stream to a file. Streams can be either *observed* by sinks (when they do not alter stream's elements) or *piped to* a sink – in that case all elements are discarded and all that is left are the effects.

`Process1` is a stream transducer, used to transform a stream much like Scala's collection methods transform an iterable. Streams are *piped* to `Process1` instances.



Streams in scalaz-stream support back pressure. It essentially means that the stream will not process more elements than are requested – the elements are *pulled* by the user, rather than *pushed* from a source. With back pressure we can easily design a scalable and responsive system without worrying about how the elements are going to be requested and processed in later stages of development.

Once the stream composition is finished we can run the `Process` in one of three ways, depending on what result we are interested in. `.run()` runs the process purely for its ef-

fect, discarding all values. `.runLog()` records all values emitted by a stream as a sequence. `.runLast()` discards all emitted elements but the last one.

The following example illustrates concepts introduced in this section.

```
// a constant, infinite stream of ones wrapped in a Task
val ones = Process.repeatEval(Task.now(1))

// infinite stream of consecutive integers starting from 2
val nat = ones.scan1(_ + _)
val even = nat.filter(_ % 2 == 0) //only even integers
// |> is a shorthand for .pipe
val indians = nat |> process1.lift(_.toString + " little indians")
val stdoutSink: Sink[Task, String] = sink.lift(str => Task.delay(println(str)))
val print = indians.to(stdoutSink)
print.take(5).run.run
```

The code prints the first five even-numbered little indians to standard output while returning none. The first `.run()` method returns an object of type `Task[Unit]]`, so the last call to `.run()` runs the `Task`. The effect of `println()` is captured by a `Task`, it will be evaluated only when the `Task` is run.

Note that scalaz-stream allows transforming a stream as if it were a standard Scala collection. Methods like `.filter()` and `.scan1()` are just aliases for piping the stream to appropriate `Process1` instances.

5.4. Basic Architecture

We propose a basic interface for the hyperparameter tuning process.

```
trait BaseOptimizer {
  type ParamsSpace = Map[String, NondetParam[Any]]
  type Params = Map[String, Any]

  trait Optimizer {
    def paramStream: Process[Task, Params]
    def update: Sink[Task, OptimizerResult]
  }
  sealed case class OptimizerResult(result: Any, params: Params, time: Long = 0)

  sealed class ExperimentStrategy
  object StrategyMinimize extends ExperimentStrategy

  def apply(maxParallel: Int, spaceDesc: ParamsSpace, strategy:
   ↪  ExperimentStrategy): Optimizer
}
```

`BaseOptimizer` is a factory, it will be extended by concrete hyperparameter tuning backend implementations. `NondetParam` is used to describe a hyperparamter's space. The space can be described as a continuous range, discrete (integer) range or a "choose one from a list", which is a special case of a discrete range. Methods for conversion to hyperparameter tuning backends' proprietary formats for specifying configuration are available via type classes.

The class for communicating with the tuning algorithm, `Optimizer`, is instantiated by the `.apply()` method. It takes a hint to the desired degree of parallelism, a hyperparameter space description and an optimization strategy as arguments. At the time the only available optimization strategy is to minimize the result (which presumably is the prediction error).

`Optimizer`'s `paramStream` generates a stream of `Params` – concrete samples from parameter space – to be evaluated. Results should be piped to `update`, along with parameters used to produce them and optionally time of the iteration in milliseconds. Note that the `Optimizer` is stateful – updates may potentially come asynchronously at any time and applying them is meant to be implemented as an effect.

Tuning an algorithm is simple.

```
def loop(params: Params) = (algorithm(params), params)

optimizer.paramStream.pipe(loop)
        .observe(optimizer.update.contramap(getResult))
        .pipe(process1.lift(_._1))
```

Here `algorithm()` is the algorithm to be tuned and `getResult()` is a function that constructs an `OptimizerResult` object from an algorithm output and configuration instance tuple. The update sink merely "observes" the results of an experiment – algorithm's output is passed along to further analysis or storage.

5.5. The Grid Search Optimizer

We will use the proposed interface to implement one of the simplest methods of hyperparameter tuning – the grid search (described in section 1.2.1).

```
object GridOptimizer extends BaseOptimizer {
  class GrOptimizer(spaceDesc: ParamsSpace, strategy: ExperimentStrategy) extends
  ↪  Optimizer {
    override def paramStream: Process[Task, Params] = {
      // a lazy generator for cartesian product of variable space
      def go(acc: Params, rest: Seq[(String, NondetParam[Any])]): Process[Task,
      ↪  Params] =
        rest match {
          case Seq() => Process.emit(acc)
          case (name, vals) +: tail => vals.view.foldLeft(Process.empty[Task,
          ↪  Map[String, Any]]) {
            (proc, value) =>
              proc ++ go(acc + (name -> value), tail)
          }
        }
      Process.suspend(go(Map.empty, spaceDesc.toStream))
    }

    override def update: Sink[Task, OptimizerResult] = sink.lift(x => Task.now(()))
  }
  override def apply(maxParallel: Int, spaceDesc: ParamsSpace, strategy:
  ↪  ExperimentStrategy): Optimizer =
```

```
    new GrOptimizer(spaceDesc, strategy)
}
```

A lazy stream of all possible configurations is constructed from the space description. Grid Optimizer suggests them for evaluation one by one. The update sink is a no-op.

The discrete range `NondetParam` instances can generate the space for their parameter. However, as the space described by a continuous range is potentially infinite, submitting a continuous `NondetParam` to the Grid Optimizer raises a runtime exception.

5.6. Concurrency

Although stream processing paradigm is highly sequential, some programs strongly benefit from concurrency. Relaks is one such case – as the tuning algorithms usually have at least some support for concurrent evaluation, a lot of time can be saved by parallelizing or distributing computation. Fortunately, scalaz-stream supports concurrency, as long as the developer is willing to sacrifice the guarantees on the order of computation and some of the push/pull semantics.

`nondeterminsm.njoin(maxOpen: Int, maxQueued: Int)(source: Process[Task, Process[Task, A]])` is a combinator that enables concurrent processing. It spawns `maxOpen` worker threads and executes operations from a Process of Processes on a first-come first-served basis. Because of the implementation constraints in order to guarantee fairness, stream elements (or *jobs*) are prefetched to a queue, whose size is bound by the `maxQueued` argument.

Parallelizing the Process from section 5.4 is fairly simple.

```
val processes = optimizer.paramStream.map(params =>
  Process.eval(Task(params))
    .pipe(loop)
    .observe(optimizer.update.contramap(getResult))
    .pipe(process1.lift(_._1)))

nondeterminism.njoin(4, 1)(processes)
```

Here the loop evaluation is processed concurrently by up to 4 threads at the same time. Note that the return type on `njoin` is a `Process[Task, A]`. The combinator takes care of merging the results back into a single stream (the order of elements becomes nondeterministic), so that subsequent processing is again sequential.

5.7. Spearmint Optimizer

Spearmint's implementation [Sno14] comes as a command line Python tool. Once the initial configuration is provided, the communication happens through an experiment log file. With each run of the command line tool the model is updated with new entries from the file and then a new configuration marked as pending is written. The tuned algorithm should then be run with the pending parameter configuration and its results appended to the experiment log. Another Spearmint iteration would be run after that in order to update the model and get another set of parameters for evaluation.

```
1 object SpearmintOptimizer extends BaseOptimizer {
2
```

```scala
class Spearmint(spaceDesc: ParamsSpace,
                strategy: ExperimentStrategy,
                maxParallel: Int,
                convergeAfter: Int = 10,
                scriptArgs: String = "--method=GPEIChooser") extends Optimizer {
  //state
  private var spearmintPending = Set.empty[Params]
  private var updatesPending = List.empty[(Params, OptimizerResult)]
  private var tmpDirectoryPath: Path = null
  //pending updates or tickets
  private val updateTicketQueue = async.unboundedQueue[OptimizerResult \/ Unit]

  /**
   * updates experiment log and runs spearmint
   * @return spearmint process exit code
   */
  protected def runSpearmint: Task[Int]

  protected lazy val initializeSpearmint: Task[Unit]

  protected lazy val paramGenerator: Process[Task, Params] = {
    val ticketInit: Task[Unit] //initialize ticket queue
    val init: Task[Unit] = initializeSpearmint.flatMap(_ => ticketInit)
    val updateWithTicket: Process[Task, Seq[OptimizerResult \/ Unit]] =
      updateTicketQueue.dequeueAvailable
    val applyUpdatesOrGetTicket: Process[Task, Unit] = updateWithTicket.flatMap
    ↪ {...}
    def getNextParams: Task[Params]

    Process.eval_(init) ++ applyUpdatesOrGetTicket.evalMap(_ => getNextParams)
  }

  //side effect: stores optimizerResult in updatesPending
  protected def applyUpdate(optimizerResult: OptimizerResult): Task[Unit]

  /**
   * reads from experiment log. Side effect: appends to spearmintPending
   * @return params pending in experiment log or unit if spearmint
   *         requested params that have already been evaluated.
   */
  protected lazy val readPending: Process[Task, Unit \/ Params]

  override def paramStream: Process[Task, Params] = paramGenerator

  override val update: Sink[Task, OptimizerResult] =
    updateTicketQueue.enqueue.contramap(x => x.left)
}
```

```
50    override def apply(maxParallel: Int, spaceDesc: ParamsSpace, strategy:
       ↪  ExperimentStrategy): Optimizer
51
52    def apply(maxParallel: Int, spaceDesc: ParamsSpace, strategy: ExperimentStrategy,
53                        convergeAfter: Int, spearmintArgs: String): Optimizer
54  }
```

In our implementation no effects are evaluated eagerly, as they can potentially be computationally expensive. Temporary directory for Spearmint's intermediate files and the configuration will not be created until the first element of `paramStream` is generated. The experiment log will not be updated and the Spearmint program will not be run until a new set of parameters is requested from the stream.

Simultaneously the implementation is parallelisable. Experiments' results are stored in a (thread safe) queue (line 13.). To ensure that no more than `maxParallel` configurations are generated at the same time (e.g. by the prefetching in `njoin`), we use the"tickets" pattern familiar in parallel programming design. At the beginning exactly `maxParallel` tickets are generated, and in order to generate a new set of parameters with Spearmint the stream must grab another ticket. After finishing the computation (or encountering a failure), the process must return the ticket to the queue. At the same time it is important to apply all available updates before calling Spearmint. Therefore, the updates and tickets are stored together in `updateTicketQueue` – tickets are represented by `Unit` instances and `\/` is scalaz' alias for `Either`. Before parameters are generated, the updates and tickets are requested in bulk (line 26.). All updates are then marked for application and any optionally extra tickets are returned to the queue.

It turns out that some acquisition functions available in Spearmint (e.g. the `GPEIChooser` described in [SLA12]) repeatedly pick the same set of parameters. Therefore, after `convergeAfter` subsequent evaluations of one configuration the stream raises `OptimizerConverged` exception.

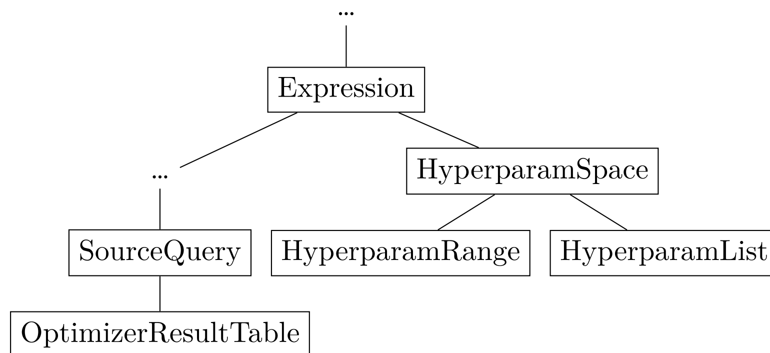
5.8. Extending the DSL

Before using a tuning algorithm with Relaks, we will need to extend the object-language. We are going to need additional syntax to define the hyperparameters and the tuning scenario.

```
//IR nodes
sealed trait HyperparamSpace extends Expression
sealed case class HyperparamRange(from: Expression, to: Exprerssion) extends
 ↪  HyperparamSpace
sealed case class HyperparamList(s: Expression) extends HyperparamSpace
sealed case class OptimizerResultTable(argTuple: Expression) extends SourceQuery

//new empty type
sealed trait UnfinishedGenTable[T] extends Table

trait HyperparamExtensions extends Symbols with TableExtensions {
  sealed trait HasRange[T]
  object HasRange {
    implicit val intHasRange = new HasRange[Int] {}
    implicit val doubleHasRange = new HasRange[Double] {}
  }
```

```
  object choose extends ToTypedTreeOps {
    abstract class Between[T: HasRange] {
      val from: Rep[T]
      def and(t: Rep[T]): Rep[T]
    }
    def between[T: HasRange](frm: Rep[T]): Between[T]
    def from[T](from: Rep[List[T]]): Rep[T]
  }

  def optimize[H <: HList, Out](varTup: Rep[Tup[H]]):
   ↪  Rep[UnfinishedGenTable[Tup[H]]]
}
```

Three new IR nodes are introduced.



This extension enables defining three kinds of hyperparameters.

```
val x = choose between 1.0 and 2.0 //continuous range
val y = choose between 1 and 10 //discrete range
val z = choose from List("Mon", "Tue", "Wed") //discrete range
```

`HasRange` is a simple method for specifying type constraints for range hyperparameters – it is possible to create a range configuration only from Integers and Doubles.

Note that those values have "normal" types (e.g. `Rep[Int]`, `Rep[Double]`), therefore they can be used anywhere an Integer or Double would be used. `val a = x + 5.0` is equivalent to defining a hyperparameter with range from 6.0 to 7.0. This way virtually any object can be turned into a hyperparamter space, e.g. `val str = "My account balance on " + z + " is €" + x` is a space of strings stating a balance between 1 and 2 € on one of the first three days of the week.

A syntax for running optimization scenarios is included as well.

```
optimize (y, str)
```

This code would tune a three hyperparameter configuration (`str` is parameterized by two hyperparamters). Note that the return type of `optimize()` contains the word "unfinished". Indeed, the optimization scenario is incomplete without specifying the error function or a parameter which should be minimized. We will extend the LINQ syntax to allow defining the objective and transforming the experiment. To operate on the newly introduced Table type we will modify the `TypedTableOps` class defined in the previous chapter to work with any Typed Table type.

```
class TypedTableOps[H <: HList, Out[_ <: HList]]
                   (arg: Rep[Table])
                   (implicit mkCmp: BuildComprehensionTyped[H, Rep[Table], Out]) {
```

The implicit argument `BuildComprehensionTyped` is similar in concept to `BuildComprehension` factory described in section 4.2.3. The main difference is that the `Out` type is a type of a higher kind this time, allowing parameterizing the output representation's type with a hlist. This is useful for operations that mutate the type of a Table, such as `.map()` or `.flatMap()` methods. Implicit instances for `BuildComprehensionTyped` exist for `TypedTable` and `UnfinishedGenTable`, allowing both types to reuse the operators defined in `TypedTableOps`.

In order to create a syntax for defining hyperparameter tuning objective, we will need a new IR node and another extension to our LINQ framework.

```
sealed class OptimizeBy(override val table: Atom,
                        override val ordering: Vector[FieldWithDirection])
  extends OrderBy(table, ordering)

trait HyperparamExtensions extends Symbols with TableExtensions {
  implicit class UnfinTablOps[H <: HList](arg: Rep[UnfinishedGenTable[Tup[H]]]) {
        def by(field: Symbol): Rep[TypedTable[Tup[H]]]
  }
}
```

`OptimizeBy` is similar to `OrderBy` node, yet its semantics are a little different – while (some) tuning algorithms will try to suggest configurations that consequently minimize the optimized field, in general the output will not be sorted. Nevertheless, defining optimization objective is just as simple as sorting a Table – all that is needed is the column name, and the optimizer implementation will take care of the rest.

Once the objective is selected, the experiment will behave just as any other Typed Table.

Chapter 6

Traversing the IR

With the basic set of IR nodes and meta-language methods for manipulating them established, we can focus on working with the IR representation generated in the meta-language's run-time stage of the multi-stage program. We are going to explore three contexts of IR traversal: static analysis, rewriting and program evaluation.

We are going to make heavy use of Scala's pattern matching and Kiama[1], a library for language processing in Scala. It contains features such as efficient attribution of IR nodes and powerful combinators for defining rewriting and traversal strategies based on Stratego [Vis01], a language for specifying program transformations. With the help of macros and run-time reflection Kiama integrates seamlessly with IRs based on Scala's case classes. Although its main focus is on tree-based IR, with just a few special considerations we can successfully apply it in our DAG IR.

6.1. Static Analysis for Program Correctness

Some constraints are difficult to express using just the Scala's type system. For example, although we require that arguments to `optimize()` are defined using hyperparameters, the type of a hyperparameter definition is indiscernible from e.g. the type of a literal. To catch such errors before the run-time, we will define a simple incremental scaffolding for specifying rules the program should adhere to.

```
trait Analysis {
  final def analyze(root: Expression): ValidationNel[String, Unit] = {
    val ir = new DAGIr(root)
    val strategy = Rewriter.collect[List, ValidationNel[String, Unit]] ({
      case (expr:Expression) => doAnalyze(expr, ir)
    })

    strategy(root).reduce { _ *> _}
  }
  protected def doAnalyze(node: Expression, ir: DAGIr):
    ValidationNel[String, Unit] = ().successNel[String]
}
```

`DAGIr` is a helper object that constructs a generic traversable representation of the IR. `collect()` is a Kiama strategy combinator that traverses the entire IR, collecting output of a

[1] https://bitbucket.org/inkytonik/kiama

strategy specified in the argument – here provided by the `.doAnalyze()` method. The results are collected in a `ValidationNel` object – another alias of `Either`, that stores failures as Lists. For an error-free program the result would be just a `Unit` instance – a success. Otherwise error message strings are collected in a List. The `*>` operator combines `ValidationNel`s, concatenating the error lists in case of at least one failure.

`Analysis` trait is meant to be extended by concrete analysis rules providers, overriding the `.doAnalyze()` method and stacking the rules.

```
trait HyperparamAnalysis extends Symbols with Analysis {
  private val optimizerValidation: (Expression, DAGIr) ==> ValidationNel[String,
   ↪  Unit] = {
    case (OptimizerResultTable(tuple), ir: DAGIr) =>
      val hyperparams = new Hyperparams(ir)
      if (hyperparams.isHyperparam(tuple))
        ().successNel
      else
        "argument of `optimize` must be hyperparameterized".failureNel
  }

  override protected def doAnalyze(node: Expression, ir: DAGIr):
   ↪  ValidationNel[String, Unit]
    = optimizerValidation.applyOrElse((node, ir), Function.const(().successNel)) *>
        super.doAnalyze(node, ir)
}
```

`.optimizeValidation()` is a partial function restricted to `OptimizerResultsTable` nodes, returning success if the node's argument is constructed with a hyperparameter and failure otherwise. `.doAnalyze()` method stacks the partial function with analysis performed in the superclass. Using Scala's mixin (trait) composition, multiple analysis rules defined for various DSL extensions can be stacked this way, creating a modular static checker architecture. Relaks implements a checker for soundness of column names in LINQ expressions (to analyze whether a Typed Table's column name actually refers to something in the current context) in a similar fashion.

Reporting errors as soon as possible is especially important in the context of long running machine learning experiments, where a bug at a later stage of the program could waste a lot of programmer's time.

6.2. IR Attribution

In the previous listing `Hyperparams` is a Kiama's attribute.

```
class Hyperparams(ir: DAGIr) extends Attribution { self =>
  val hyperparamDependencies: Expression => Set[Sym] = attr {
    case _ /> OptimizerResultTable(_) => Set.empty
    case Some(sym) /> (_: HyperparamSpace) => Set(sym)
    case Fresh(_) => Set.empty[Sym]
    case _ /> (link) => ir.child(link)
                          .map(self.hyperparamDependencies)
                          .foldLeft(Set.empty[Sym])(_ ++ _)
```

```
    case _ => Set.empty
  }

  def isHyperparam(expr: Expression) = hyperparamDependencies(expr).nonEmpty
}
```

Kiama supports attribute grammars as functions that compute properties of IR nodes and their context. The results are memoized and, with the help of macros and reflection, stored directly in the IR nodes. The attributes are automatically reset after an IR rewrite that might alter their values.

Here `.hyperparamDependencies()` traverses the IR recursively to construct Sets of `Syms` linking to the hyperparameter definitions for a node and its children. The `DAGIr` object is required for generic IR traversal, in this case to a node's subgraph. On a side note, in the case of Relaks, using attributes that depend on node's parents should be given a careful consideration or be avoided entirely – what works well with an AST representation might sometimes break in the DAG IR context.

6.3. IR Rewriting

Program transformation has many purposes, including desugaring advanced language constructs in terms of simple ones and applying optimization rules in order to arrive at a more efficient solution.

6.3.1. Rewriting LINQ to a Simpler Representation

Working directly with a tree (or DAG) representation of queries is inefficient both from a cognitive point of view and an efficiency perspective. It is often useful to look at a query as a whole, as in various contexts the representation is traversed top-down and bottom-up. Furthermore, the individual operations can interact with each other within a query, producing a more efficient solution, while their interaction with separate queries has its own laws.

Relaks converts Query subgraphs to atomic `logical` nodes. Here logical refers to an operation that could be submitted to a database, modeled after blocks described in [LMS94]. They cannot be directly manipulated by the meta-language.

```
package relaks.lang.dsl.extensions.ast.logical

sealed trait Comprehension extends Expression
sealed case class LoadComprehension(loadExpression: SourceQuery) extends
 ↪  Comprehension
final case class SelectComprehension(from: Comprehension,
                                     operations: Vector[QueryOp] = Vector.empty)
  extends Comprehension {
    //create a new comprehension by appending an operator
    def append(queryOp: QueryOp): SelectComprehension
  }
```

`LoadComprehension` is a simple wrapper for `SourceQuery` nodes. `SelectComprehension` corresponds to a `SELECT [operations] FROM [from]` SQL query sans `GROUP BY` instructions. `QueryOp` nodes are simplified versions of query transformation nodes defined in section 4.2.3 –

as they all belong to a single query and are stored as a sequence in the Comprehension node, the `table` attribute referring to the subject of a transformation has been dropped.

```
object QueryOp {
  sealed trait QueryOp extends Expression
  case class Transform(generator: GeneratorBase, select: Atom) extends QueryOp
  case class Filter(generator: GeneratorBase, filter: Atom) extends QueryOp
  case class Limit(start: Atom, count: Atom) extends QueryOp
  case class OrderBy(ordering: Vector[FieldWithDirection], isExperimentObjective:
   ↪  Boolean) extends QueryOp

  def unapply(expr: Expression): Option[QueryOp] = expr match {
    case (op: lang.ast.Filter) => Filter(op.generator, op.filter).some
    case (op: lang.ast.Transform) => Transform(op.generator, op.select).some
    case (op: lang.ast.Limit) => Limit(op.start, op.count).some
    case (op: lang.ast.OrderBy) => OrderBy(op.ordering,
     ↪  op.isInstanceOf[OptimizeBy]).some
  }
}
```

The logic behind translating Query trees into Comprehension nodes is expressed using the attribution grammar.

```
object ComprehensionBuilder extends Attribution {
  val comprehension: Expression => Option[SelectComprehension] = attr {
    case Some(querySym) /> NextTable(nextSym) =>
      val _ /> query = querySym
      for {
        select <- comprehension(nextSym)
        updatedSelect <- query match {
          case QueryOp(queryOp) => select.append(queryOp).some
        }
      } yield updatedSelect
    case _ /> (source: SourceQuery) =>
      SelectComprehension(LoadComprehension(source)).some
    case _ => None
  }
}
```

The tree is traversed top-down, up to a `SourceQuery` node, where a `SelectComprehension` is created. Then the Operation nodes are appended recursively.

A rewriting strategy uses the `ComprehensionBuilder` attribute to materialize a new logical node.

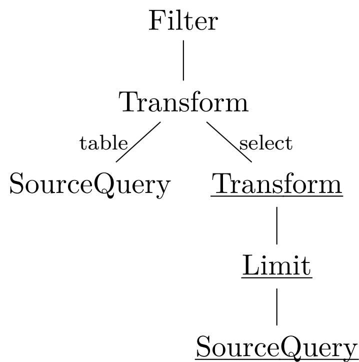
```
def buildComprehensions: Expression => Option[Expression] = {
  def buildComprehensionsImpl: Expression ==> Atom = {
    case Some(sym) /> _ if ComprehensionBuilder.comprehension(sym).nonEmpty =>
      ComprehensionBuilder.comprehension(sym).get
  }
```

```
  repeat(oncetd(rule[Expression](buildComprehensionsImpl))) andThen
    ↪ (_.map(_.asInstanceOf[Expression]))
}
```

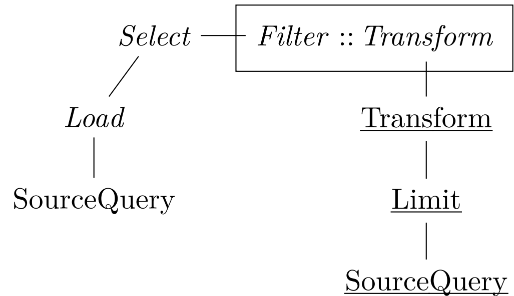
Kiama's combinators are used to apply the simple `buildComprehensionsImpl()` strategy to the entire IR. `repeat()` combinator replays a strategy as long as it succeeds. `oncetd()` tries to apply a strategy to the entire IR traversing it top-down and succeeds after the first successful application.

Repeated application of the strategy can transform Query subtrees located at any depth – including nested queries. Figure 6.1 depicts applying the strategy to the following code.

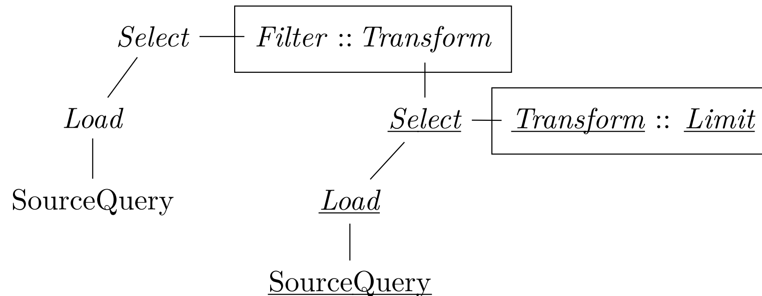
```
val data = load("data.csv")('a::[Int], 'b::[Int])
val nested = load("nested.csv")('c::[Int], 'd::[Int])

val joined = for {
  Tup(a, b) <- data
  Tup(c, d) <- nested.limit(...)
} yield (a, b, c, d)

joined.filter(...)
```



(a) IR without tranformations (the representation of `val nested` is underlined)



(b) IR after first iteration of the strategy (logical nodes in *italics*)



(c) IR after second iteration of the strategy

Figure 6.1: Applying the `buildComprehensions` strategy to `joined.filter(...)` representation

Note that Relaks' Syms enable common subquery factorization for query expressions. While, with a reasonable caching mechanism this property has the potential for improving

the efficiency, a successful implementation requires careful consideration of query's backend, table size etc. It is not within the current scope of the project, but remains an important future work. Presently a common subquery is split into distinct logical nodes.

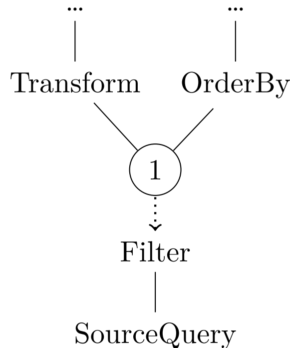


Figure 6.2: A representation of two queries with a common child (subquery). It will be rewritten as two distinct comprehensions.

6.3.2. Rewriting for Efficient Execution

One of the reasons why Relaks is centered around a LINQ implementation is the huge potential for improving query efficiency by IR rewriting either by predicate move-around [LMS94], operator merging (e.g. Spark Catalyst in [Arm+15]) or operator specialization. Even though Relaks implements only a small subset of a query language, multiple optimization rules are still applicable.

One example would be specializing a frequently seen duet – Order By plus Limit operators. Whenever there are no intermediate Filter operators between an Order By and Limit nodes a more efficient Top-K algorithm (based on a priority queue with bound size) can be used instead of the generic sorting implementation (which is realized by an unbounded priority queue in Relaks).

Future Work

Ultimately, Relaks should implement a predicate move-around algorithm to push down the predicates directly into hyperparamter definitions whenever possible. Investigating how useful predicate reordering could be for constraining hyperparameters' space or automatically deriving interactions between them[2] remains an important future work.

The query avalanche problem (section 4.2.3) is frequently solved by rewriting nested comprehensions to Join nodes. We have implemented a proof of concept rewriting strategy for Relaks IR, however due to time constraints and lower usefulness of joins in experiment databases, it has not been included in the query rewriting phases yet.

6.4. The IR Traversal that Could – the Interpreter

In the previous chapter we have already seen bits that will build the object-program – streams and stream transducers. In this section we will implement methods that execute instructions derived from the IR nodes and run Relaks programs.

[2]Some model-based tuning algorithms such as SMAC [HHL11] support conditional parameters, where one parameter is active only when another parameter takes values from a given set.

6.4.1. A Preliminary: Type Materialization

As a typed programming language, Scala requires type information to issue commands to the JVM. Although it could be derived at run-time using reflection, a simpler and clearer solution is to store some type information in the IR nodes during meta-language compile time. We will briefly describe how to extend the current implementation in order to provide useful type information for the interpreter.

First we will define a basic set of phantom types.

```
sealed trait TType

//as type parameters are "erased" during Scala's compile-time
//materializing the TypeTag is useful for printing the type and
//for equality checks
sealed abstract class ArgType[T: WeakTypeTag]
  (implicit val order: Order[T] = null,
   val toText: Injection[T, String] = null)
  extends TType

//represents a type supported by Relaks object-language
sealed class LiftedArgType[T] extends ArgType[T]

final class TupType[T <: HList : WeakTypeTag]
  (val length: Int, val lowerBound: TType, val childrenTypes: Vector[TType])
  extends LiftedArgType[Tup[T]]

//represents a type equivalent to Scala type T
//objects of type T can be directly lifted into Literals nodes
sealed class ScalaType[T: ClassTag] extends LiftedArgType[T]

class ScalaNumType[T : ClassTag : Order : Field] extends ScalaType[T]

object UnknownType extends TType

object ScalaTypes {
  val boolType = new ScalaType[Boolean]
  val stringType = new ScalaType[String]
  val intType = new ScalaNumType[Int]
  val doubleType = new ScalaNumType[Double]
  val longType = new ScalaNumType[Long]
  val anyType = new ScalaType[Any]
}
```

Along with the essential type information we are storing some useful optional implicit objects: `Order`, `Field` and `Injection[T, String]` instances. They will become extremely useful while implementing generic expression evaluators later in this chapter.

We include a new field `val tpe: TType` for storing the type information in the basic `Expression` IR node. Adding type information does not complicate the existing code by much – some types, such as `TupType` can be derived automatically from the IR node's children and others can be made implicitly available and fetched based on type parameter `T` in `Rep[T]`.

6.4.2. The Interpreter

Relaks' interpreter is structured similarly to the analysis framework descibed in section 6.1. Various traits providing the interpreter implementation for different language extensions are stacked together to form a complete language interpreter. Several overridable methods provide implementation for different classes of expressions in order to guarantee some type safety of the expected result of the evaluation.

```
protected def evalComprehensionPartial: Expression ==> Process[Task, impl.Row]
protected def evalExpression(expr: Expression): Any = expr match {
  case _/>Literal(v) => v
  case Some(sym) /> (expr: Expression) => evalExpression(expr)
  case _ =>
    throw new NotImplementedError(s"Evaluating $expr not implemented")
}
```

Query nodes' interpretetation functions are stacked by overriding the `.evalComprehensionPartial()` method. Its return type is a stream of `impl.Row` objects, which is a common interface for Tuple expressions evaluation results.

```
package relaks.lang.impl

trait Row {
  def get[T: ArgType](i: Int): T
  def get[T: ArgType](name: String): T
  def apply(i: Int): Any)
  def apply(colname: String): Any
  //BrVec is breeze.Vector, ClassTag specializes the Vector with
  //a more efficient storage for primitive values whenever possible
  def apply[T: ClassTag: ArgType](rng: Range): BrVec[T]

  def colNames: Seq[String]
  def values: Seq[Any]
}
```

The concrete implementation of `Row` trait is chosen by a query backend – we will see a specialized row type in chapter 7.

As mentioned before, Query Operators are implemented as stream transducers and combined together with Source Queries, which become stream generators. Aggregation and sorting is implemented using algebird's Aggregations [Twi15a], where the aggregation objects form a commutative semiring structure[3], so that integrating them with streams is straightforward.

```
def mkOrder(fld: FieldWithDirection): Order[Row]
...
//returns a Process1[Row, Row]
case OrderBy(fields, false) =>
  implicit val ordering: scala.Ordering[Row] =
  ↪  fields.map(mkOrder).fold.toScalaOrdering
```

[3]For more information on how semirings fit into databases see [GKT07]

```scala
  val aggregator = new UnboundedPriorityQueueAggregator[Row]

  process1.lift(aggregator.prepare) |>
    process1.reduce(aggregator.reduce) |>
    process1.lift(aggregator.present) flatMap { (x: Seq[Row]) =>
    Process.emitAll(x)
  }

case TopK(fields, skipExpr, countExpr) => //top k specialization mentioned
↪  previously
  implicit val ordering: scala.Ordering[Row] =
   ↪  fields.map(mkOrder).fold.toScalaOrdering
  val skip = evalExpression(skipExpr).asInstanceOf[Int]
  val count = evalExpression(countExpr).asInstanceOf[Int]
  val max = skip + count
  val aggregator = new PriorityQueueAggregator[Row](max) //pq's size bounded by max

  process1.lift(aggregator.prepare) |>
    process1.reduce(aggregator.reduce) |>
    process1.lift(aggregator.present) flatMap { (x: Seq[Row]) =>
    Process.emitAll(x)
  }  |> process1.drop(skip).take(count)
```

`mkOrder()` combines Order objects stored in type information for fields to an Ordering instance for the whole row.

Type information is also useful for evaluating simple expressions.

```scala
case Apply("+", lExpr :: rExpr :: Nil) =>
  val l = evalExpression(lExpr)
  val r = evalExpression(rExpr)
  val field = lExpr.tpe.asInstanceOf[ScalaNumType[Any]].field
  field.+(l, r)
```

Here `.field` is a `breeze.Field` described in section 3.4.

6.4.3. Environments

In order to implement the higher order functions introduced in section 4.2.2 the interpreter must be able to construct and access a mutable Environment (state). In Relaks Environments are stackable `Sym` → `Expression` Maps placed on the top of the basic definitions Map provided by the `Symbols` trait (listing 2, section 3.6.1). Whenever a function is about to be executed, a new environment is pushed on the top of the stack, filled with the function arguments' values linked to Syms listed in a Generator. After exiting the function, the environment is popped from the stack. The trait providing Environments overrides the `.findDefinition()` method from listing 2, so that accessing Sym links is recursive, starting at the top of the stack.

Syms can be written and overwritten at any level of the stack. In chapter 7 we will see how to use this property to perform run-time optimizations of the IR such as hoisting and caching.

Chapter 7

Advanced Extensions

Relaks extensions we have seen so far provided the essential functionality for the object-language and object-program. The extensions we will introduce in this chapter will further expand Relaks' capability for interacting with different types of data and code and improve programs' efficiency.

7.1. LINQ to Drill

With a LINQ implementation already in place it would be a shame not to include an option for querying data tables, especially as operations on tables are often a part of a machine learning pipeline.

Apache Drill[1] [HN13] is a query engine without the datastore layer. It can operate on data stored in various formats (including CSV and JSON) in the filesystem or in external data warehouses without the overhead of data set import.

Drill can be queried using a SQL subset. Communicating with Java programs is possible using JDBC, the standard library for taking to relational databases. We will focus on CSV files and execute a Drill query while interpreting the `LoadTableFromFs` node defined in section 4.2.3.

```
case _/> Select(LoadComprehension(LoadTableFromFs(path)), seq) if
↪   path.endsWith(".csv") =>
  val initialProjections: Seq[(Field, TType)]
  val projection: String = toProjection(projectL)
  val query = s"SELECT ${projection} FROM dfs.`${path}`"
  val rows: Process[Task, WrappedResultSet]

  if (projection != "*") {
    val schema = Schema(initialProjections.map(x => x._1.name -> x._2).toVector)
    rows |> process1.lift { wrs =>
      logger.debug("got drill row")
      new JDBCRow(wrs, schema)
    }
  } else {
    rows |> process1.lift { wrs =>
      val array = wrs.any(1).asInstanceOf[JsonStringArrayList[Text]]
```

[1] http://drill.apache.org/

```
      val vals = (0 until array.size()).map(i => array.get(i).toString).toVector
      new CsvAllRow(vals)
    }
  } // returns Process[Task, Row]
```

There are a few quirks of CSV processing with Drill. First of all, it does not process header rows and for each row all columns are stored as a list named `columns`. Drill is capable of working with collections and performing type casts, but we would explicitly need to specify the CSV file schema. In the case of Typed Table representations we already have that, so it is enough to traverse the operator sequence `seq` and collect field names and types. Following Relaks' semantics, the cast is constructed from initital Operators up to the first Transform Operator. If the Transform Operator is Untyped, then we must issue a `SELECT * FROM ...` query to Drill, selecting all columns in the CSV file. Otherwise, Fields and types stored in the IR are collected and SQL code for casting is generated.

```
val table = load("table.csv")
table.filter("columns[0]".::[Int])
        (case Tup(first) => first > 10)
     .apply("columns[1]".::[Double],
      ↪ "columns[2]".::[String])
//returns
 ↪ Rep[TypedTable[Tup[Double::String::HNil]]]
```

```
SELECT CAST(columns[0] as INT),
 ↪ CAST(columns[1] as DOUBLE),
 ↪ columns[2] FROM dfs.`table.csv`
```

Listing 6: A LINQ query and a corresponding SQL query issued to Drill.

Note that, albeit very limited at the moment, we have effectively introduced another object-language to our now multi-language, multi-stage program. With an additional effort we could lift Operators (including predicates, sorting, simple expressions etc.) into SQL queries and rely on the database backend come up with an efficient execution plan.

In section 6.4.2 we have seen the interface for a Table Row implementation. The Drill Interpreter chooses one of two implementations of the `impl.Row` interface, based on the executed query. `impl.JDBCRow` and `impl.CsvAllRow` differ on the underlying storage type. `impl.JDBCRow` contains a JDBC `ResultSet` with column name information and concrete types, while `impl.CsvAllRow` is just a String Vector, as by default CSV files are read as rows of Strings. In the latter case the `Injection[T, String]` stored with type information becomes useful for working with columns of effective types other than String[2].

7.2. Calling Native Scala Code

Calling native Scala functions in the multi-stage program run-time stage can be useful for many reasons. First of all, it is no secret that interpreted languages are usually slow compared to compiled ones. As the JVM performs various optimizations, such as JIT compilation, and the interpreter language is simpler than Relaks' compiled Scala functions should be faster than interpreted expressions. Secondly, the lack of effects and recursion in Relaks can be limiting in some cases. When there is no need for IR inspection and the function behaves like a pure function from the outside, the constraints can be loosened. Last but not least, Scala and Java benefit from a rich set of libraries created the over many years they have been used

[2] Although the Injection is of type `T` → `String`, usually an `Injection.inverse()` method is implemented as well enabling conversion both ways.

in the industry. Since Relaks programs already run on the JVM heap, we can access all those libraries at virtually no cost.

As usual, we will begin by introducing a set IR nodes for Native Scala Calls.

```
sealed case class ApplyNative(fn: Any, argTuple: Expression) extends Expression
sealed case class Native(value: Any) extends Atom
```

`ApplyNative` represents a reflective call to some Scala function `fn` using tuple `argTuple` as arguments. `Native` wraps the result of a function call.

The expression

```
to (knn _) apply (k, dist, test)
```

is represented by an `ApplyNative` node. `knn()` is a Scala function `(Int, DistMatrix, impl.Row)` → `Decision` and the tuple `(k, dist, test)` is of type `Rep[Tup[Int :: DistMatrix :: Tup[..] :: HNil]]`. The Native Type extension makes heavy use of Scala's implicits to enforce type safety. First, shapeless macros are used to convert a function of type `(A, B, ..., X)` → `Out` to type `A::B::...::X::HNil` → `Out`. Next, the argument hlist element types must be translated to their corresponding empty types whenever applicable. We have mentioned before that several types appearing as type arguments in `Rep[T]` are empty – they are never instantiated and provide no useful implementation. On the other hand, a Scala function such as `knn()` must operate on concrete objects, so the meta-language must know which empty types correspond to which objects created by the interpreter. The task is straightforward with types like `Int` or `String`, but translations for others, such as `Tup` or `TypedTable`, must be explicitly provided.

```
class Translation[X, Y]
implicit val tableTranslation = new Translation[Table, impl.Table]
implicit val untypedTableTranslation = new Translation[UntypedTable,
 ↪  impl.UntypedTable]
implicit def typedTableTranslation[H <: HList] = new
 ↪  Translation[TypedTable[Tup[H]], impl.TypedTable[H]]
implicit val rowTranslation = new Translation[Tupl, impl.Row]
implicit val untypedRowTranslation = new Translation[UntypedTup, impl.UntypedRow]
```

Here the translations are written inversely, to emphasize that it is an empty type that is being translated. For modularity the translations should be provided by appropriate DSL extensions.

The case of identity translations is particularly interesting, as we would like to keep the type only if an explicit translation is not defined. Explicitly defining all possible translations, including identity translations, is unfeasible, as with the introduction of Native Call Extensions a representation can be of any valid Scala type. Fortunately, Scala's implicit system has a limited (and probably accidental) support for negation. Scala will not be able to provide an implicit whenever more than one implicit instance exist in the scope at the same time. We get negation for some type by introducing ambiguity – purposefully defining two implicit values of that type.

Here, we would like to be able to negate the `Translation` type, if a Translation already exists. Therefore, we instantiate an implicit value of type `Not[Translate[X, Y]]` (where `Not` is just some trait) for each `X` and `Y`, and again for each `X` and `Y` whenever an implicit `Translate[X, _]` or `Tanslate[_, Y]` instance exists.

Once finished with the function's parameters, we modify the function to wrap its results in a `Native` node whenever the return type is not a `Rep` already.

To give an example, let us go back to our `knn()` expression – in this case the following steps will be performed:

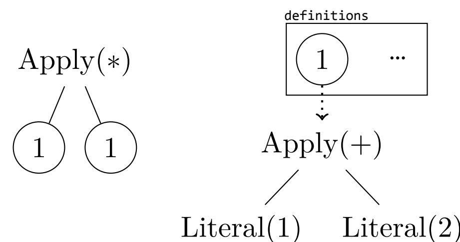
1. Functions is transformed from `(Int, DistMatrix, impl.Table)` → `Decision` to an `Int :: DistMatrix :: impl.Row :: HNil` → `Decision` function.
2. The argument hlist is translated to `Int::DistMatrix::Tupl::HNil`.
3. The tuple `(k, dist, test): (Int::DistMatrix::Tup[H]::HNil)` is checked for whether it can be applied to the function. As `Tup[H] <: Tupl` the application is possible.
4. As `knn()` does not return a `Rep`, the function is wrapped in a higher order function that will return a `Native` node applied to `knn()`'s result. The type of the expression becomes `Rep[Decision]`.

7.3. Hoisting and Caching

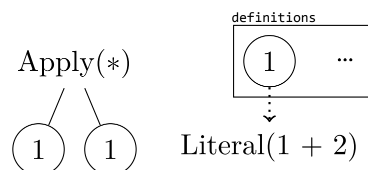
In section 3.6 we have stated the need for a let insertion/caching mechanism. Let us look at a few examples to build a better understanding of what we are trying to achieve.

```
val a: Rep[Int] = 1
val b: Rep[Int] = 2
val c = a + b
c * c
```

This is the example from section 3.6. Without any optimizations the following IR will be processed by the interpreter.



Even though Relaks recognizes common subexpression, it will sill be evaluated twice, as no intermediate results are stored. As stated previously, at some point we would like to transform the representation to a more efficient version, so that the same expression will not be evaluated twice.



This transformation can be described in simpler terms using an exemplary imperative language.

```
var a = 1
var b = 2
fun c() = a + b
c() * c()
```

⟹

```
var a = 1
var b = 2
fun c() = a + b
var c_memo = c()
c_memo * c_memo
```

Higher order functions were introduced to express call-by-name semantics. With this example we are starting to see the solution we are looking for – subexpression `c` should be evaluated once, and its results stored and reused instead of `c()` in the future.

Here is another example, with even greater capacity for optimization.

```
loop (var i = 0; cond(i); i++) {
  var a = doSomethingUnrelated()
  doSomethingElse(i)
}
```

$\Longrightarrow$

```
var a = doSomethingUnrelated()
loop (var i = 0; cond(i); i++) {
  doSomethingElse(i)
}
```

This transformation is called *hoisting*. The value of `a` is loop invariant – it does not change with the iterations of the loop, therefore it can be moved outside. A variable is loop invariant when it does not depend on anything created in the loop's scope, including the iterator.

Although Relaks does not have an explicit construct for blocks, stacking environments produced by Generators while executing inlined functions (sections 4.2.2 and 6.4.3) produce similar effects. Therefore it still matters at which level of the stacked environment a value is stored – too high on the stack means that the value will be evicted before we can use it again, too low will just take up memory space.

Loops in Relaks are based on generators and lists (or tables) rather than conditions. We can often extract useful information about the size of the loop or the number of unique elements. Let us consider a bit more complicated example.

```
nloop (longList, shortList) { i, j =>
  var a = doSomething(j)
  a + i
}
```

```
var cache = emptyCache
nloop (longList, shortList) { i, j =>
  var a = cache.tryGet(j) || doSomething(j)
  a + i
}
```

`nloop(list1, list2, ...)` is a nondeterministic loop over a cartesian product of a (possibly infinite) set of `lists`. This is a simplified model of what happens inside the iterative hyperparameter tuning process – we can think of the `longList` as of an optimization over a continous parameter, while `shortList` represents a discrete parameter's space. Note that this is substantially more complicated than having two nested loops, which could be optimized by loop reordering. The order of evaluation is not determined, and there may even be some configurations that would never be evaluated. The cache mechanism is a workaround for this problem, although it should not be used eagerly. Let us consider what happens when `i` and `j` change places in the previous example:

```
nloop (longList, shortList) { i, j =>
  var a = doSomething(i)
  j + a
}
```

When `longList` is very large, much bigger than the `shortList`, it does not make sense to memoize the results of `doSomething(i)` – the probability of a cache hit would be close to zero.

The last example will show how to effectively use a cache in even less friendly conditions.

```
nloop(superLongList) { k =>
  nloop (longList, shortList) { i, j =>
    var a = doSomething(j, k)
    a + i
  }
}
```

```
nloop(superLongList) { k =>
  var cache = emptyCache
  nloop (longList, shortList) { i, j =>
    var a = cache.tryGet(j) || doSomething(j, k)
    a + i
  }
}
```

As the superLongList is even bigger than the longList, it does not make sense to cache every possible value of doSomething(j,k). Instead of inserting the cache at the top level, we can make it local to the outer loop's iteration – it will be useful while the inner loop is working and then automatically removed for an iteration with a new k.

7.3.1. Implementation

Relaks includes a proof-of-concept implementation of hoisting and generic caching. It can reason about the sizes of loops and the number of times each value may appear in it and tries to come up with the best caching scenario. Caching is introduced by providing an extension and a slight modification to the basic stackable interpreter function from section 6.4.2.

```
def evalExpression(expr: Expression): Any = expr match {
  case _/>Literal(v) => v
  case _/>(c: CachedExpression) => retrieveFromCache(c)
  case Some(sym) /> (expr: Expression) =>
    cache(sym)
  case _ => throw new NotImplementedError(s"Evaluating $expr not implemented")
}

case class CachedExpression(key: Int, cacheLevel: Int, cardinality: Int, args:
 ↪  List[Sym], original: Expression) extends Atom with Leaf
}

trait BaseCachingInterpreter {
  protected def retrieveFromCache(cachedExpression: CachedExpression): Any
  protected def cache(s: Sym): Any
}
```

Matching all nodes of that are Sym links automatically includes a caching layer for all stacked interpreters. Upon first sight the original linked expression is replaced with a CachedExpression logical node, which includes information on how to access the cached value. key plus args instantiated to concrete values are used to uniquely identify the memoized object. cardinality is an upper bound on expected appearances of a distinct value (the number of different objects that may be cached for this CachedExpression) in a loop. level refers to a place in a stack to look for the appropriate cache. The stack of caches works alongside Environments stack, pushing and popping new layer happens at the same time for both data structures – the lower the level, the shorter a value that will be stored. Finally, the original Expression can be used to construct another value for the expression – useful for either reconstructing an evicted object or instantiating the value with a new set of parameters.

Caches are implemented with Google Guava's Cache for Java[3]. Currently the values are stored as Java's soft references[4], which are garbage collected by the JVM once the memory begins to run out. With Guava it is straightforward to implement other eviction strategies, such as based on manually specified maximum size.

The most complicated part of the caching mechanism in Relaks is cardinality (or the-number-of-distinct-values) estimation for expressions. Cardinality itself is implemented as a simple structure with a plus operation $(\mathbb{N} \cup \mathrm{Inf}, \oplus)$, where

[3] https://github.com/google/guava/wiki/CachesExplained

[4] http://docs.oracle.com/javase/6/docs/api/java/lang/ref/SoftReference.html

$$a \oplus b = \begin{cases} \text{Inf} & \text{if } a = \text{Inf} \lor b = \text{Inf} \\ a \times b & \text{otherwise} \end{cases}$$

For each estimated node dynamic dependencies are collected first. By dynamic dependencies we mean Syms from the node's subtree (or transitively from the current loop's context), that are not accessible before run-time, either because they link to a hyperparamter definition or are the "holes" to be filled in by the inlined function's parameters. Loop context includes the Source Query of current LINQ expression and dependencies derived by traversing the Comprehension IR node. Next, a simple heuristic is used to determine cardinality for each dependency.

```
val cardinalityEstimation: Expression => Cardinality = attr {
  case _/>> (HyperparamRange(_, _) :@ tpe) if tpe == ScalaTypes.doubleType =>
  ↪  Infty.left
  case _/>> HyperparamRange(_/>>Literal(left), _/>>Literal(right)) =>
    (right.asInstanceOf[Int] - left.asInstanceOf[Int]).max(1).right
  case _/>> HyperparamList(list) => cardinalityEstimation(list)
  case _/>> ListConstructor(lst) => lst.length.right
  case _/>> Literal(l:List[_]) => l.length.right
  case _/>> TableFromList(lst) => cardinalityEstimation(lst)
  case _/>> LoadTableFromFs(_) => Infty.left
}
```

Then we must look for the best cache `level` to insert the cached value. The cardinality of an expression is evaluated incrementally against the environment stack in the current context. Starting from the last environment on the stack (the least nested), we are looking greedily for the smallest sub-stack that has a finite cardinality. In each iteration we merge all dependencies' estimations using the $\oplus$ operator. If the result is not finite we remove all dependencies originating in the current environment and try again with the environment higher on the stack. The mechanism defaults to putting the cached value in the topmost cache – an operation equivalent to let insertion within the current scope.

Currently the implementation eagerly inserts the cache whenever the cardinality is "something less than infinity". It could be easily modified to allow only caches of size below a certain threshold.

Future Work

The cardinality estimation and caching scope is very simplistic at the moment, although it is a substantial improvement over the call by name policy. Instead of fully materialized tables only a `Process` instance is stored – accessing it again would recompute the whole query. Better cardinality estimation for queries would allow a bolder table caching policy and might be a partial solution to the query avalanche problem (section 4.2.3) – it remains an important future work.

Note that the `CachedExpression` carries a huge overhead when used to store primitive values. Succeeding versions of Relaks should specialize the caching mechanism for working with simple types.

Chapter 8

Evaluation

This chapter presents hyperparameter tuning scenarios that validate the basic Relaks framework and its advanced extensions.

8.1. Branin-Hoo

First, we will consider a simple example of finding the minimum of a two-dimensional continuous function.

$$f(x) = (x_2 - \frac{5}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6)^2 + 10(1 - \frac{1}{8\pi})\cos(x_1) + 10$$

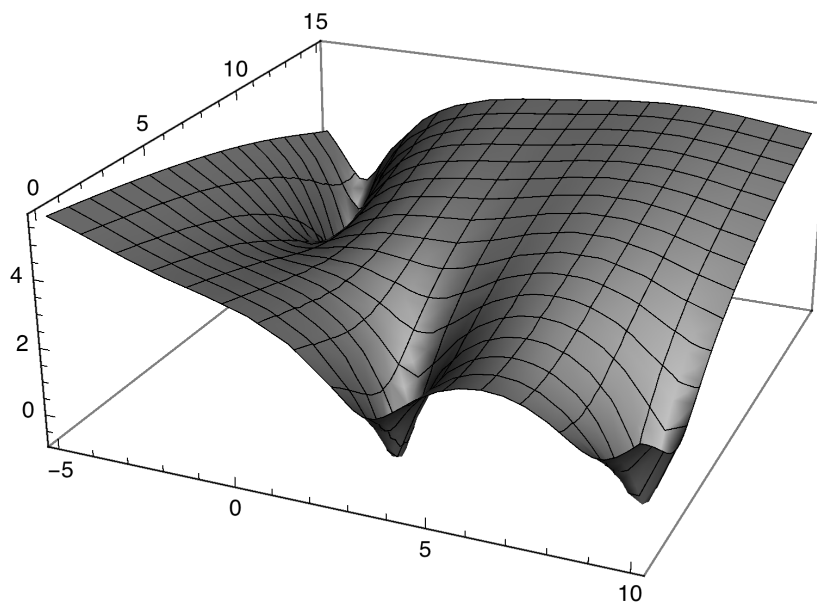


Figure 8.1: Branin-Hoo function, log scale.

Branin-Hoo[1] or Branin is commonly used for evaluating Bayesian optimization techniques. Within the domain $-5 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 15$ it has three global minima $f(\hat{x}) = 0.397887$.

The optimization scenario implementation in Relaks is fairly simple.

[1] http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page913.htm

```
object Program extends Relaks(SpearmintOptimizer) {
  val x = choose between 0.0 and 15.0
  val y = choose between -5.0 and 10.0

  val result = optimize (x, y) map { case Tup(x, y) =>
    val res = to (branin _) apply (x, y)
    (x as 'x, y as 'y, res as 'result)
  } by 'result limit 200

  run(result)
}
```

The tuning performance varies on the acquisition function used by Spearmint. We have evaluated two of the available functions: EI and EI Opt. Both of them use *expected improvement (EI)* to choose the next candidate, but EI Opt uses a more thorough and expensive sampling method for computing the EI. It tends to achieve better results, but with time each Spearmint evaluation takes substantially longer.

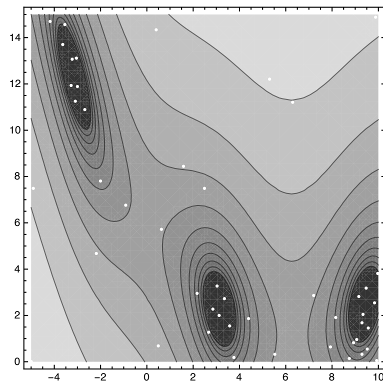


Figure 8.2: Dots mark the configurations chosen by the EI function. Log scale, darker is smaller.

x	y	result
-5.0	0.0	308.129
2.5	7.5	24.129
6.308	11.191	121.417
-1.982	7.822	9.545
⋮	⋮	⋮
2.866	2.270	0.812
9.985	3.823	2.564
-3.103	11.271	1.234
-3.227	13.051	0.757
5.517	0.322	17.610

Table 8.1: An experiment database generated for the Branin experiment.

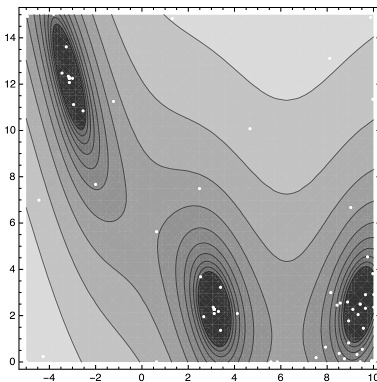


Figure 8.3: Choices of the EI Opt function.

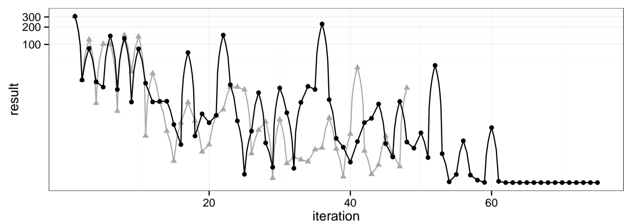


Figure 8.4: Results produced by each tuning iteration, cut off after the 75th. Triangles mark the EI function, circles is EI Opt.

Both methods were set to run 200 evaluations, although EI usually converged after about 50 iterations. The minima found were 0.4764 and 0.3979 for EI and EI Opt respectively, which is in line with previous results achieved with Spearmint [EFH13].

8.2. Knn

In section 1.1.3 we have already mentioned the issues with tuning a knn classifier. In the next example we will be looking for the best set of hyperparameters for a knn classifier to use with Vowel Recognition Data Set [Lic13]. The data set consists of 15 speakers saying one of eleven vowels six times. Each instance consists of a ten dimensional vector of speech data. The goal is to train a classifier to identify a vowel using only a subset of speakers and then use it to classify vowels spoken by previously unseen speakers. The data has been preprocessed for 4-fold cross validation, partitioned by the speakers, so that all instances of a speaker fall into the same fold. An additional column identifying the fold has been added.

We are going to use a knn classifier, testing the performance with k ranging from 1 to 5 and one of four distance measures: the Minkowski distance for $p = 1$ (Manhattan), 2 (Euclidean) and distances based on infinity (Chebyshev) and zero (Hamming) vector norms.

Since the parameter space is fairly small (5 distinct values for k and 4 different distance functions, 20 possible configurations) and the classifier is simple we are going to use the grid search algorithm for hyperparameter tuning.

```
1  object Program extends Relaks(GridOptimizer)
2  import Program._
3
4  case class FeatureVec(features: Vector[Double], klass: Int)
5
6  val distFn = choose from List(normDistance(1).asRep,
7                                normDistance(2).asRep,
8                                hammingDistance.asRep,
9                                infNormDistance.asRep)
10 val k = choose between 1 and 5
11 val ds = load("./train-vowels.csv")
12
13 def dataSet(filter: Row[Int] => Rep[Boolean]) =
14   ds.filter(Tuple1("columns[11]".::[Int]))(filter)
15     .map({(row: Rep[UntypedTup]) =>
16       val vec = row.liftMap(rowi => FeatureVec(rowi[Double](0 to 9),
          ↪  rowi.get[Int](10)))
17       vec as 'features
18     })
19
20 val f1_avg: Rep[Double] = List(1,2,3,4).asTable.map({ case Tup(Tuple1(fold)) =>
21   val train = dataSet(row => row(0) neq fold)
22   val test = dataSet(row => row(0) === fold)
23
24   val tree = (makeTree _).pure.apply(distFn, train)
25
26   val confMatrix: Rep[DenseMatrix[Double]] = (knn _).pure.apply(k, tree, test)
27   val f1_score = confMatrix.liftMap(mx => -f1_accuracy(mx)._1)
```

```
28      f1_score as 'result
29    }).avg
30
31    val experiment = optimize ((k as 'k, distFn as 'dist, f1_avg as 'result)) by
       ↪  'result
32
33    run(experiment sortBy 'result)
```

First, the hyperparameter configuration is specified and the data is loaded from a CSV file. The `dataSet` function takes a filter as a parameter and applies it to the data set. The intention is to use the filter to select data belonging to a correct fold.

Then the data set is mapped into a Feature Vector, which is a native Scala class. Since the data has 12 dimensions, it would be too strenous to specify the full schema by hand. Instead, we use an Untyped Table and map a subset of columns from each instance to a Vector. `row.liftMap(f)` (line 16.) is a shorthand for `to (f _) apply (row)`.

In line 20. a (staged) `List[Int]` is transformed (transposed) into a `TypedTable[Tup[Int::HNil]]`. The list represents the four cross validation folds. For each fold the data set is split into a test and a training set. The training set is used to construct a M-tree – a structure for slightly more efficient spatial queries. Scala's native function `makeTree()`, given a distance function, constructs a M-tree from a Table (line 24.).

The tree is then used to predict the vowel label from the test set. `knn()`, another host function takes the `k` paramter, the M-tree and the test Table and returns a confusion matrix (line 26.). We will need the confusion matrix to compute the error. We are estimate the error with F_1 score[2], a measure frequently used to evaluate multiclass classification problems. Since it is an accuracy measure (the bigger the better), we need to multiply the result by -1 because the hyperparameter tuning algorithm will try to minimize it (we are using grid search at the moment, so it does not really matter, but we might want to replace it with Spearmint in the future).

Finally an average error across all folds is computed (line 29.) and the optimization scenario is run. Note that if we did not care for storing the hyperparameters' values in the experiment database, we could optimize just the `f1_avg` object – the interpreter would figure out the hyperparameter dependencies automatically.

Caching

The knn example uses all the features available in Relaks – loading data from files and processing it with LINQ, native function calls, automatic hyperparamter tuning and, behind the scenes, the caching algorithm. The most interesting instance is the `makeTree()` result (line 24.). We can easily see that there is some potential for memoization here – the function works with a data table and thus can take some time to compute, so the gain from caching would most likely be noticeable. Furthermore, the function uses just one of two hyperparameters, which means a higher probability of a cache hit during the optimization.

Unfortunately, the call is made from a cross validation inner loop and it cannot be hoisted. All in all, the result of `makeTree()` depends on the `distFn` hyperparameter and the `train` parameter, which in turn depends on the `fold` value generated by the List Table.

Relaks' cache extension can figure out those dependencies. It will resolve the parameters required to create a unique cache key for each instance of `makeTree()`'s result (in this case the value of `fold` and `distFn`) and place the cache in the global (last) scope. Based on the

[2] https://en.wikipedia.org/wiki/F1_score

size of the `distFn` list (4) and the List Table (4), the estimated cardinality will be 16. As the total number of iterations for the example is 80 (16 times the size of `k`'s space), we can expect a cache hit rate of 75%.

We have compared running the experiment with and without a cache. In this instance caching is super effective, as it saves us a trip to the database and processing the full set of rows.

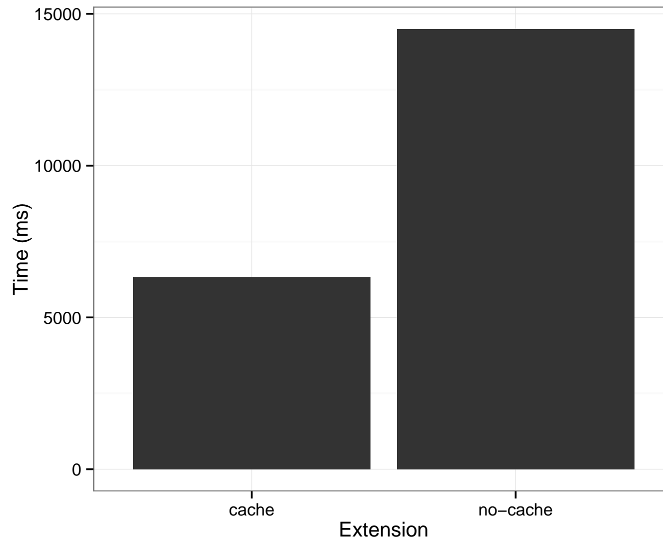


Figure 8.5: Experiment timed with and without cache, averaged over 10 runs.

Results

It turns out that the best hyperparameters are $k = 4$ and the Manhattan distance measure. The best configuration evaluated on a validation data set, set aside in the beginning, achieves a 49.5% accuracy, with 229 out of 462 test samples classified correctly.

Can we do better? The only way to be sure is to run more experiments.

With just a small modification to the original code

```
object Program extends Relaks(SpearmintOptimizer)
...
val p = choose between 0.1 and 4.0
val tree = (makeTree _).pure.apply(p.liftMap(p => normDistance(p)), train)
```

we sacrifice caching `makeTree()`, but in turn we can test Minkowski's distance with the p parameter ranging from 0.1 to 4.

Relaks evaluates 200 iterations of hyperparameter tuning in under 9 minutes. The F_1 score achieved this time is slightly better, and the validation data set accuracy improved as well, up to 51.73%, closer to the state of the art result for knn classification on this data set – 56% [Lic13].

k	dist	result
4	normDistance(1.0)	-0.453
3	normDistance(1.0)	-0.451
1	normDistance(1.0)	-0.437
2	normDistance(1.0)	-0.437
5	normDistance(1.0)	-0.435
1	normDistance(2.0)	-0.395
2	normDistance(2.0)	-0.395
3	normDistance(2.0)	-0.392
4	normDistance(2.0)	-0.384
5	normDistance(2.0)	-0.366
3	normDistance(inf)	-0.324
5	normDistance(inf)	-0.321
1	normDistance(inf)	-0.284
2	normDistance(inf)	-0.284
4	normDistance(inf)	-0.279
3	hammingDistance	-0.113
4	hammingDistance	-0.108
1	hammingDistance	-0.100
2	hammingDistance	-0.100
5	hammingDistance	-0.053

Table 8.2: (Sorted) results of the knn experiment.

k	dist	result
1	1.23755	-0.47187
1	1.61976	-0.46158
5	1.11552	-0.46103
5	1.11418	-0.46103
5	1.11457	-0.46103
5	1.11610	-0.46103
5	1.11594	-0.46103
5	1.11370	-0.46103
5	1.11516	-0.46103
5	1.11489	-0.46103
5	1.11426	-0.46103
5	1.11688	-0.46103
5	1.11497	-0.46103
5	1.11716	-0.46103
5	1.11396	-0.46103
5	1.11658	-0.46103
5	1.11458	-0.46103
5	1.11525	-0.46103
5	1.11441	-0.46103
	⋮	

Table 8.3: Top results of the knn experiment with a continuous p value. Note that when the hyperspace represented by the data set is sparse (which is often the case with high-dimensional data) and the distance function changes only slightly, the predictions might not change at all.

Chapter 9

Discussion and Future Work

Relaks successfully implements a framework for meta-learning. It integrates creating hyperparameter optimization scenarios into a Domain Specific Language. We have shown that its syntax is simple and familiar to the developers already acquainted with SQL or LINQ in other frameworks. Even major changes in tuning configuration and strategies can be specified with ease.

We have validated the choice of implementing the framework as a DSL by presenting the potential for analysis and optimization allowed by the direct access to the IR. Rewriting strategies present in other compilers and interpreters and those specific to application with meta-learning were introduced.

We have introduced a type safe LINQ implementation and integrated it with two hyperparameter tuning algorithms, allowing the user to control declaratively their behavior, with no knowledge of their implementation required.

We have shown how Relaks fits in a real-life machine learning pipeline, enhancing it and removing some burden from programmers' shoulders.

The work presented in the thesis is but a first step in implementing a framework for declarative meta-learning. While it is already useful, there is still room for improvement. In the rest of this chapter we will revisit the problems and weaknesses of Relaks mentioned in the previous chapters and sketch a plan for future enchancements.

9.1. Future Work

Extending LINQ

Our ultimate vision for declarative meta-learning experiments includes a portfolio of machine learning algorithms aware of the hyperparameter tuning system. We would like to reach a point when a user can take a generic algorithm from the portfolio and seamlessly tune it for her machine learning pipeline using a natural syntax.

```
optimize(knnAlgorithm).filter(knn => knn.k < 5 and knn.distFn == manhattan).map(knn
↪  => knn.train(...))
```

We have laid an important groundwork already, but to execute effectively such query Relaks should implement the aforementioned predicate move-around techniques [LMS94], tailored to meta-learning's specific demands.

Improving the capabilities of LINQ implementation is also in order. Both operations natural to traditional databases (such as grouping) and those useful in our context (e.g. a limit for the total time an experiment will take) are important extensions. Furthermore, the

query avalanche problem, for which the caching extension is just a work-around, should be solved while introducing a full support for joins.

Code Generation

Right now, while we introduce some extensions to improve efficiency, Relaks does not put much emphasis on the speed of execution. Substituting the interpreter with a code generator for Scala (or Java) and SQL will be a crucial improvement in this matter. Since big parts of Relaks – IR representation, its manipulation extensions, static analysis and rewriting – are indifferent to what happens at later stages of a multi-stage program, the transition will only affect an isolated part of the current system.

Concurrency

Distributed or parallel processing is an important part of any machine learning solution. With an ever-growing data sets, exploiting just one processor of just one machine is often not enough.

While the hyperparameter tuning algorithm framework is implemented with concurrency in mind, there are some things to be considered in other parts of Relaks, especially with the transition to code generation in mind. Concurrency could either be implemented directly in Relaks or achieved via integration with other systems, such as Spark [Zah+10].

Persistent Caches and Experiment Databases

Extending the cache to support off-heap storage could offer benefits to some long-running, data-heavy pipelines (see [Li+14]), but also take the experiment database paradigm one step further. With a resilient caching system capable of storing different kinds of data come numerous advantages, namely the capability to interrupt a long running optimization scenario and resume it later, or the ability to review and query all previous experiments' results.

Bibliography

[Apa15] Apache. *Apache Pig*. 2015. URL: http://pig.apache.org/.

[Arm+15] Michael Armbrust et al. "Spark SQL: Relational data processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[Bia+09] Federico Biancuzzi et al. *Masterminds of programming: Conversations with the creators of major programming languages.* " O'Reilly Media, Inc.", 2009.

[BMT07] Gavin M Bierman, Erik Meijer, and Mads Torgersen. "Lost in translation: formalizing proposed extensions to C#". In: *ACM SIGPLAN Notices*. Vol. 42. 10. ACM. 2007, pp. 479–498.

[BV07] Hendrik Blockeel and Joaquin Vanschoren. "Experiment Databases : Towards an Improved Experimental Methodology in Machine Learning". In: (2007), pp. 6–17.

[Cha+10] Hassan Chafi et al. "Language virtualization for heterogeneous parallel computing". In: *ACM Sigplan Notices*. Vol. 45. 10. ACM. 2010, pp. 835–847.

[Chl+15] Pavel Chlupacek et al. *scalaz-stream – Compositional, streaming I/O library for Scala*. 2015. URL: https://github.com/scalaz/scalaz-stream.

[EFH13] Katharina Eggensperger, Matthias Feurer, and Frank Hutter. "Towards an empirical foundation for assessing bayesian optimization of hyperparameters". In: *...Bayesian Optimization ...* (2013), pp. 1–5. URL: http://automl.org/papers/13-BayesOpt%5C_EmpiricalFoundation.pdf.

[For05] Malcolm R Forster. "Notice: No-Free-Lunches for Anyone, Bayesians Included". In: *Department of Philosophy, University of Wisconsin–Madison Madison, USA* (2005).

[GKT07] Todd J Green, Grigoris Karvounarakis, and Val Tannen. "Provenance semirings". In: *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2007, pp. 31–40.

[Hal+09] Mark Hall et al. "The WEKA data mining software: an update". In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[Hal15] David Hall. *Breeze – a numerical processing library for Scala*. 2015. URL: https://github.com/scalanlp/breeze.

[HHL11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *Learning and Intelligent Optimization*. Springer, 2011, pp. 507–523.

[HK07] D Richard Hipp and D Kennedy. *SQLite*. 2007.

[HN13] Michael Hausenblas and Jacques Nadeau. “Apache Drill: Interactive Ad-Hoc Analysis at Scale”. In: *Big Data* 1.2 (2013), pp. 100–104. ISSN: 2167-6461. DOI: 10.1089/big.2013.0011. URL: http://online.liebertpub.com/doi/abs/10.1089/big.2013.0011.

[Hoo10] Holger H Hoos. “Programming by Optimisation”. In: (2010), pp. 1–28.

[Hut+09] Frank Hutter et al. “ParamILS: an automatic algorithm configuration framework”. In: *Journal of Artificial Intelligence Research* 36.1 (2009), pp. 267–306.

[HVO06] WG Halfond, Jeremy Viegas, and Alessandro Orso. “A classification of SQL-injection attacks and countermeasures”. In: *Proceedings of the IEEE International Symposium on Secure Software Engineering*. Vol. 1. IEEE. 2006, pp. 13–15.

[JG11] Norbert Jankowski and Krzysztof Grąbczewski. “Universal meta-learning architecture and algorithms”. In: *Meta-Learning in Computational Intelligence*. Springer, 2011, pp. 1–76.

[Koh95] Ron Kohavi. “A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection”. In: *International Joint Conference on Artificial Intelligence* 14.12 (1995), pp. 1137–1143. ISSN: 10450823. DOI: 10.1067/mod.2000.109031.

[Kra+13] Tim Kraska et al. “MLbase : A Distributed Machine-learning System”. In: (2013).

[Kri01] Shriram Krishnamurthi. “Linguistic reuse”. In: *Computer Science, Rice University, Houston* (2001).

[Kuh08] Max Kuhn. “Building Predictive Models in R Using the caret Package”. In: *Journal of Statistical Software* 28.5 (Nov. 10, 2008), pp. 1–26. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v28/i05.

[Li+14] Haoyuan Li et al. “Tachyon: Reliable, memory speed storage for cluster computing frameworks”. In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–15.

[Lic13] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: http://archive.ics.uci.edu/ml.

[LMS94] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. “Query optimization by predicate move-around”. In: *VLDB*. 1994, pp. 96–107.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. “Linq: reconciling object, relations and xml in the. net framework”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 706–706.

[Nel+11] Christopher Nell et al. “HAL: A Framework for the Automated Analysis and Design of High-Performance Algorithms”. In: *LION*. Vol. 6683. 2011, pp. 600–615. ISBN: 978-3-642-25565-6. DOI: 10.1007/978-3-540-92695-5.

[Ode11] Martin Odersky. *Scala By Example*. 2011. URL: http://www.scala-lang.org/docu/files/ScalaByExample.pdf.

[Rom12] Tiark Rompf. “Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming”. PhD thesis. EPFL, 2012.

[Sab+15] Miles Sabin et al. *shapeless – Generic programming for Scala*. 2015. URL: https://github.com/milessabin/shapeless.

[Sch+10] Tom Schreiber et al. “Thirteen new players in the team: a ferry-based linq to sql provider”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 1549–1552.

[SLA12] Jasper Snoek, Hugo Larochelle, and RP Adams. “Practical Bayesian optimization of machine learning algorithms”. In: *Advances in Neural Information ...* (2012), pp. 1–9. URL: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.

[Sno14] Jasper Snoek. *Spearmint*. 2014. URL: https://github.com/JasperSnoek/spearmint.

[Tah99] Walid Taha. “Multi-stage programming: Its theory and applications”. PhD thesis. Oregon Graduate Institute of Science and Technology, 1999.

[Tho+13] Chris Thornton et al. “Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms”. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2013, pp. 847–855.

[Twi15a] Twitter. *Algebird – Abstract Algebra for Scala* *https://twitter.com/scalding*. Aug. 2015. URL: https://github.com/twitter/algebird.

[Twi15b] Twitter. *Bijection – Reversible conversions between types*. Aug. 2015. URL: https://github.com/twitter/bijection.

[Typ15] Typesafe. *Slick*. Mar. 2015. URL: http://slick.typesafe.com/.

[Vis01] Eelco Visser. “Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5”. In: *Rewriting techniques and applications*. Springer. 2001, pp. 357–361.

[Vog11] Jan Christopher Vogt. “Type safe integration of query languages into Scala”. PhD thesis. Diplomarbeit, RWTH Aachen, Germany, 2011.

[Wol95] David H Wolpert. “Off-training set error and a priori distinctions between learning algorithms”. In: *Sante Fe Institute, Santa Fe, NM, USA, Tech. Rep. SFI-TR* (1995), pp. 95–01.

[Wol96] David H. Wolpert. “The Lack of A Priori Distinctions Between Learning Algorithms”. In: (1996).

[Xie14] Yihui Xie. “knitr: a comprehensive tool for reproducible research in R”. In: *Implementing Reproducible Research* (2014), p. 1.

[Zah+10] Matei Zaharia et al. “Spark: cluster computing with working sets”. In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Vol. 10. 2010, p. 10.