Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Developing a Type System for a Configuration Specification Language

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering and Internet Computing**

eingereicht von

**Armin Wurzinger, BSc.**
Matrikelnummer 1528532

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam
Mitwirkung: Dipl.-Ing. Dr. Markus Raab

Wien, 30. August 2018

Armin Wurzinger    Franz Puntigam

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Developing a Type System for a Configuration Specification Language

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering and Internet Computing**

by

**Armin Wurzinger, BSc.**
Registration Number 1528532

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam
Assistance: Dipl.-Ing. Dr. Markus Raab

Vienna, 30th August, 2018

| Armin Wurzinger | Franz Puntigam |
| --- | --- |

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

# Erklärung zur Verfassung der Arbeit

Armin Wurzinger, BSc.
Schubertstraße 7
3371 Neumarkt/Ybbs
Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. August 2018

Armin Wurzinger

# Acknowledgements

Many thanks to Dipl.-Ing. Dr. Markus Raab for guiding me through this thesis, for all of his valuable inputs and ideas regarding Elektra and for the professional way in which he maintains the whole project.

I would also like to thank René Schwaiger and Lukas Winkler for helping me out with build-related questions and concerns. Thanks to Stefan Oberender for proofreading my thesis.

I also want to give gratitude to Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam for all his helpful input, ideas and help regarding type systems and formal methods.

Last, I am grateful of all the support I got from my family while I wrote my thesis, and thanks to all the other people who supported me during that time.

# Kurzfassung

Elektra ist eine Initiative zur Entwicklung, Wartung und Bereitstellung einer universellen Programmbibliothek für Konfiguration. Die Programmbibliothek abstrahiert über verschiedene Konfigurationsbestandteile um einheitlich darauf über Schlüssel-Werte-Paare zugreifen zu können. Es können auch Metainformationen zu diesen Schlüssel-Werte-Paaren gespeichert werden. Diese Metainformationen können verwendet werden um Konfigurationsspezifikationen zu definieren, welche die Semantik von Konfigurationen beschreibt. Beispiele für solche Metainformationen sind Bedingungen auf dem Wert eines Schlüssels, Verbindungen um Beziehungen zwischen Schlüssel-Werte-Paaren zu spezifizieren, und Transformationen, um Konfigurationsbestandteile aus anderen ableiten zu können. Konfigurationsspezifikationen können fehlerhaft sein, was zu unerwartetem oder falschem Verhalten zur Laufzeit führen kann.

In dieser Diplomarbeit wird das Problem von fehlerhaften Konfigurationsspezifkationen gelöst indem ein Typsystem für Elektra entwickelt wird. Dieses Typsystem erkennt manche Fehler in einer Konfigurationsspezifikation statisch, um daraus resultierende Fehler zur Laufzeit zu reduzieren. Das HM(X)-Gerüst wird als die formale Basis des zu entwickelnden Typsystems HM(RGX) verwendet. HM(X) ist ein generisches formales Gerüst um domänenspezifische Hindley-Milner-artige Typsysteme zu spezifizieren. Die nötigen Beweisauflagen, die vom HM(X) Gerüst vorgegeben werden, werden erfüllt, um zu zeigen, dass das Typsystem in sich konsistent ist.

Ein Prototyp dieses Typsystems wird in Form einer eingebetteten domänenspezifischen Sprache (EDSL) in der Programmiersprache Haskell entwickelt. Dafür wird ein Plugin für den Glasgow Haskell Compiler (GHC) geschrieben, mit dem die Semantik von HM(RGX) bei der Typüberprüfung umgesetzt wird. Es wird ebenso ein Plugin für Elektra entwickelt das Konfigurationsspezifikationen, die mit dieser EDSL beschrieben werden, auf Typfehler überprüft. Darüber hinaus wird ein weiteres Elektra Plugin geschrieben, das Metainformationen von Elektra über Schlüssel-Werte-Paare in die EDSL übersetzt. Eine Fallstudie wird durchgeführt um zu analysieren welche von Elektra aktuell unterstützten Metainformationen mit HM(RGX) beschrieben werden können.

# Abstract

Elektra is an initiative for developing, maintaining and providing a universal library for system- and application configuration. This library abstracts over various configuration items to provide a unified access, based on key-value pairs. Key-value pairs can store meta-information. This meta-information can be used to write configuration specifications describing the semantics of configuration. Examples for such meta-information are conditions on the value of a key, links to specify relationships between keys, and transformations to derive information based on other keys. Configuration specifications can be erroneous, leading to unexpected or faulty behavior at runtime.

We improve the problem of erroneous configuration specifications by developing a type system for Elektra. This type system detects some errors in a configuration specification statically to reduce failures at runtime. We use the HM(X) framework as the foundation to describe our type system HM(RGX). HM(X) is a generic formal framework for specifying Hindley-Milner-style type systems. We fulfill the necessary proofs as imposed by this framework in order to show that our type system is sound and supports type inference.

We implement a protoype of our type system by creating an embedded domain specific language (EDSL) in the general purpose programming language Haskell. We extend the Glasgow Haskell Compiler (GHC) by using a typechecker plugin to realize the semantics of HM(RGX). We develop a plugin for Elektra that typechecks configuration specifications written in our EDSL. We create another Elektra plugin that translates key-value pairs and their meta-information into our EDSL so it can be typechecked. We conduct a case study where we analyzed what kind of meta-information currently supported by Elektra can be described using HM(RGX).

# Contents

**Kurzfassung** **ix**

**Abstract** **xi**

**Contents** **xiii**

**1 Introduction** **1**
1.1 Goal of this Thesis . . . . . . 2
1.2 Methodological Approach . . . . . . 3
1.3 Structure of this Thesis . . . . . . 4

**2 Background** **5**
2.1 Elektra . . . . . . 5
2.1.1 Unified Configuration Framework . . . . . . 5
2.1.2 Plugins . . . . . . 6
2.1.3 Configuration Specifications . . . . . . 8
SpecElektra . . . . . . 8
Sharing Configuration . . . . . . 10
Type Checking . . . . . . 11
Advantages . . . . . . 12
2.2 Type Systems . . . . . . 12
2.2.1 Advantages . . . . . . 13
2.2.2 Simply Typed Lambda Calculus . . . . . . 13
2.2.3 The polymorphic lambda calculus (System F) . . . . . . 17
2.2.4 HM(X) . . . . . . 19

**3 HM(RGX)** **23**
3.1 Case Study . . . . . . 23
3.1.1 Categorizing the Metakeys . . . . . . 23
3.1.2 Mapping the Categories to Type System Features . . . . . . 24
Theoretical Foundation . . . . . . 25
Checks . . . . . . 26
Transformations . . . . . . 29
Links . . . . . . 29

Structural Types . . . 30
3.2 Formal Definition . . . 31
3.2.1 Defining RGX . . . 32
3.2.2 Proofing the Soundness of RGX . . . 38
3.2.3 Type Inference for HM(RGX) . . . 42
3.3 Examples . . . 46

**4 Implementation** **49**
4.1 Specelektra . . . 51
4.1.1 Haskell Extensions . . . 52
4.1.2 GHC Typechecker Plugins . . . 53
4.1.3 EDSL . . . 54
4.1.4 GHC Typechecker Plugin . . . 56
4.2 Spectranslator . . . 58
4.2.1 Libelektra Haskell Bindings and Plugins . . . 58
4.2.2 Parsing . . . 59
4.2.3 Translation . . . 60
4.2.4 Example . . . 61
4.3 Elektra Typechecker Plugin . . . 62
4.3.1 Libelektra Haskell Plugins . . . 62
4.3.2 Type Checking . . . 64
4.4 Regexdispatcher . . . 65
4.5 Case Study . . . 66

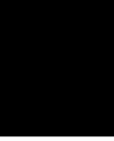
**5 Related and Future Work** **71**
5.1 Related Work . . . 71
5.1.1 Dhall . . . 71
5.1.2 ConfigV . . . 72
5.2 Future Work . . . 72
5.2.1 Error Messages . . . 72
5.2.2 Data Types . . . 73
5.2.3 Structural Types . . . 73
5.2.4 Dependent Types . . . 73
5.2.5 Implementation of Metakeys . . . 73
5.2.6 Contextual and Circular Links . . . 73

**6 Conclusion** **75**

**Bibliography** **77**

**Appendix I** **81**

CHAPTER 1



# Introduction

Currently software systems are getting more and more complex. To cope with the rising complexity systems are usually built by combining and configuring existing components to create new systems. A frequent issue when combining such modular components is that they specify their configuration in different syntaxes with a varying amount of expressiveness. This makes it difficult to have a uniform way of accessing the configuration of other software towards developing a system-oriented integration between applications. Therefore an external specification of configuration items is required to abstract differences between incompatible component configurations. [Raa16]

To unify configuration access, the open-source initiative *Elektra* has been formed, developing the library *libelektra*. The initiative maintains several tools and libraries related to libelektra, we refer to the whole ecosystem as Elektra. It is in essence a key-value database providing unique keys as a common denominator for accessing configuration items. It can integrate existing configuration files in different widely-used file formats like INI, XML and JSON to directly manipulate and interact with configuration files using its unified key-value approach. [Raa16] [RB17]

Elektra offers a unified way of accessing configuration items, thus abstracting syntactical differences between them. Some issues still remain to be solved. When integrating configuration items of several different origins to achieve system-wide integration, configuration items can differ in terms of their semantics. An example of such semantics is that a given configuration item is a positive number to be interpreted as seconds, whereas another configuration item represents a positive number to be interpreted as minutes. Initially there was no uniform way of specifying the semantics of configuration items in Elektra. This gap has then been filled by the configuration specification language *SpecElektra*. [RB17]

SpecElektra allows us to specify the semantics of configuration items by describing them with meta-information, stored as metakeys, on keys residing in a special part of the key-

value database. Following the modular approach of Elektra the effects of those semantics are implemented at runtime via plugins. The task of writing such specifications is error-prone, as inconsistencies or missing transformations within a configuration specification can lead to problems during execution. Therefore it is beneficial to utilize formal methods to help ensuring that configuration specifications are sound. This helps in avoiding these pitfalls, easing the task of specifying. According to [Pie02] there are various formal methods which can be used for this goal with varying amounts of power, such as Hoare logic and denotational semantics. An important goal of SpecElektra is its ease of use, which rules out such sophisticated approaches. A popular, lightweight and easily usable approach is to develop a type system to specify rules for configuration specifications to ensure their correctness. [Raa15] states that SpecElektra also includes code generation techniques to generate statically typed configuration access code, which also promotes the usage of a type system for configuration specifications instead of other methods.

## 1.1 Goal of this Thesis

We propose to implement a type system for SpecElektra which allows us to assign types to various configuration items described in specifications. By encoding the semantic meaning of configuration items into types, it allows us to detect erroneous configuration specifications before they can result in runtime failures. For instance, type abstractions can be used to declare some configuration items to be of different units of measurement. Linking between such configuration items without a proper transformation would then yield a type checking error, as their semantics differ.

The main goal of this thesis is to formalize a type system for the configuration specification language SpecElektra. This type system should be easy to use while still providing increased safety during the development of configuration specifications. SpecElektra itself is highly-modular with hardly any built-in metakeys. Instead keys describe various aspects of configurations, like constraints or links.

Newly created plugins can introduce additional metakeys which may or may not have an influence on the type of configuration items. This yields the necessity to keep the type system itself extensible to support new plugins. Towards fulfilling this requirement, we intend to allow users to express the effect of metakeys on the type of keys using functions. Doing so we avoid having to adjust the type system specification every time to support new metakeys.

We develop a formal specification of a type system including a few predefined metakeys. It is used to cover the basic functionality required for expressing the effects of other metakeys. We implement our formal specification as an instance of the *HM(X)* framework, a general framework for defining domain-specific type systems based on the Hindley-Milner type system presented in [Sul00]. Using this basic feature set users of the type system should be able to express the typing information for most of the currently known metakeys of SpecElektra. We formally prove the soundness and completeness of our specification.

Plugins for Elektra that implement the runtime behavior of metakeys can be written in various programming languages. We only create a type system using functions to express the influence of metakeys on the type of keys. However, we do not intend to enable plugin developers implementing the actual runtime effect of such metakeys, i.e., we do not create a programming language, just a type system. Users of our proposed type system have to guarantee that their plugins handle the effects of metakeys according to the definition of the corresponding function in our type system.

Moreover, we develop an implementation of the proposed type system for Elektra which supports all the features of the specification. We also develop a plugin for Elektra which checks the type of a configuration specification when it is being used, notifying the user about errors in a specification that might lead to runtime issues.

## 1.2 Methodological Approach

To implement the proposed goal, we first perform a literary research on the lambda calculus and HM(X) to form a theoretical foundation for our thesis. We then categorize the metakeys of SpecElektra that are currently known to be used by plugins to decide which features we include in our type system.

Based on that we formalize an instance of HM(X) by defining each element of the five-tupel parameterizing HM(X). Along the way we also show how our type system fulfills the required axioms imposed by the HM(X) framework to guarantee that the instance is also sound with a sound and complete type inference algorithm. This step constitutes our first research question:

**Research Question 1.** *Is the developed type system specification for SpecElektra sound and complete, i.e., does it fulfill the required axioms imposed by the HM(X) framework?*

The next step is to implement a prototype of our type system to use in Elektra. We implement it in the functional programming language Haskell. Haskell has a powerful type system that can be utilized to specify our own type checking rules without having to worry about the underlying type checking algorithms. This allows to experiment with different type system features offered by Haskell. As a prerequisite we need to develop a language binding to use Elektra within Haskell. Furthermore we provide a plugin which uses the developed type checker to check specifications that are being used in Elektra, notifying the user about detected errors. To show the usefulness of our type system we answer our second research question:

**Research Question 2.** *How many metakeys of SpecElektra that are currently known to be used by plugins, i.e., specified in the* METADATA.ini *file of the Elektra project with the status of being implemented, can have their behavior described by our type system?*

## 1.3 Structure of this Thesis

Chapter 2 introduces some background knowledge about Elektra, SpecElektra and type systems. This aims at helping the reader to understand the topics and concepts of this thesis. It explains general information about type systems resulting from our literary research. The other results of the literary research will be introduced later, when we discuss various specific type system techniques building upon the basics.

Chapter 3 starts with an analysis of the currently available metakeys of SpecElektra, their implementation status and their requirements on a type system towards answering our second research question. Building upon the analysis we discuss which type system techniques are useful for typing the analyzed metakeys, based on the remaining results of our literary research. The chapter then continues with a formal specification of our type system based on the findings presented in Chapter 2. First we introduce the HM(X) framework and explain the foundation it offers and how to define a domain-specific instance of it. Then we continue to define each parameter of the five-tuple that parameterizes HM(X) for our target domain. Along the way we show that our instance fulfills the axioms imposed by the HM(X) framework to have a sound instance with a sound and complete type inference algorithm, answering our first research question. We finish the chapter with a small example showing the type checking according to our defined rules.

After we have established the formal ground of our type system we describe its implementation in Chapter 4. We will present the different modular parts of our type system. Furthermore we explain how they interact with each other to check configuration specifications for correctness. At the end of the chapter we answer our second research question. Building upon the analyis of the currently available metakeys of SpecElektra in Section 3.1.1 we check which of them can have their effects expressed via our type system.

We present some related works on configuration type checking in Chapter 5 and also state some ideas on areas where our type system can be improved. Chapter 6 finishes this thesis by concluding and recapping its outcome.

CHAPTER 2

# Background

In this chapter we introduce the results of our literary research as specified in the methodological approach. First we introduce the universal configuration library *liblektra* in Section 2.1. We start with some general information about libelektra and its architecture in the sections 2.1.1 and 2.1.2. Then we introduce *SpecElektra*, a configuration specification language for which we develop a type system, in Section 2.1.3. After having elaborated on Elektra, we recap some general information about type systems in Section 2.2. We explain some advantages of using type systems in Section 2.2.1. Afterwards we introduce a variant of the lambda calculus, a basic model of computation that is often used for expressing type systems and even programming languages, called the simply typed lambda calculus in Section 2.2.2. In Section 2.2.3 we introduce more expressive variants of the lambda calculus. We finish this Chapter by introducing HM(X), a framework for creating a domain-specific type system that we are going to use for our own problem domain in Section 2.2.4.

## 2.1 Elektra

*Elektra* is an initiative for developing, maintaining and providing a universal library for configuration. The core library itself is called *libelektra*. As the goal of this thesis is not directly related to the library's core and uses other libraries provided by the initiative, we will refer to the project using its general name Elektra. It serves as the basis of several additional components built on top of it. [Raa10]

### 2.1.1 Unified Configuration Framework

Elektra abstracts over various configuration items such as user settings or system preferences in a uniform way as key-value pairs, referred to as *keys*, serving as the atomic unit of the library. Keys also store additional meta-information about them. A number

of different keys together form a *keyset* to describe an application's whole configuration in an easy way [Raa10]. For persisting an application's configuration over time, Elektra stores keysets in a global database called *KDB*. The KDB and keysets are organized in a hierarchical manner. Each key consists of a unique name formed out of its own name appended to the parent key names, separated by slashes. Therefore the naming scheme resembles the naming scheme of file systems. [Raa10]

A key's name does not have to be unique in the KDB as a whole, but actually Elektra's concept of namespaces requires it to be unique per namespace. A namespace is simply another component of a key's name, prepended to it. Thus it is possible to distinguish between keys with the same name residing in different namespaces. At the time of writing, the following namespaces are supported:

- *system*, containing default configurations settings predefined by system administrators
- *user*, containing configuration settings for particular users, where each user has his own user namespace isolated from each other.
- *dir*, containing configuration settings for particular working directories
- *proc*, containing in-memory keys which are not stored in the KDB
- *spec*, containing specifications and constraints for another key to ensure they will behave as expected

The main purpose of namespaces is to support the concept of *cascading*. Keys can be either adressed directly in regard to a specific namespace, or they can be adressed in a cascading style by omitting a specific namespace. When accessing a specific key from a keyset, Elektra will first check the *spec* namespace. It will only retrieve meta-information from the spec namespace, but never use a key's value from there. This meta-information is used to verify a key's specification later on, given that a specification is defined for it. After eventually having collected some metadata, it will look for a key's value in the proc namespace. If there is no key with that name in this namespace, it will fall back to the dir namespace, then to the user namespace and ultimately it will refer to the system namespace. This concept allows users to override default settings provided by application developers by overriding the configuration using the user namespace. The system-wide configuration will not be altered and other users will not see such configuration changes. The system-wide configuration is read-only for users without administrative privileges. [Raa10]

### 2.1.2 Plugins

Elektra uses a modular architecture. The core libelektra defines a plugin interface, which can be used to extend it with additional functionality while keeping the core small, fast

and platform-independent. A plugin's interface consists of five functions, though it is not necessary to implement all of them depending on its purpose. Each function of a plugin deals with different concerns according to Elektra's plugin interface. The first function is *Open*, which allows plugins to initialize themselves. The next function is *Get* which gets called whenever Elektra reads from the KDB. *Set* complements Get by being called upon writing to the KDB. In case of an error, *Error* is called to give plugins a chance to cleanup allocated resources. *Close* gets called to allow plugins to deallocate all resources they use for their functionality. In Elektra a keyset gets passed between all plugins in a certain order according to the backend's composition. There are various placements where a plugin can intercept the default behavior of Elektra when reading from or writing to the KDB. The placements that are going to be used in the plugin that we develop in the course of this thesis are *postgetstorage*, allowing us to check keys when they are being retrieved from the KDB, and *presetstorage*, to check keys before they are being written to the KDB.

Plugins can have different dependencies and can possibly be much more heavy than the core. Plugins use a contract which is a description of their properties. For instance, a contract can describe the version of Elektra a plugin is written for or additional plugins that a plugin depends on. A contract can also contain various metakeys describing different aspects of a plugin in terms of their maintenance status, implementation details, potential memory leaks, and similar. There are different types of plugins, the most important being *storage* plugins, *check* plugins and *filter* plugins. [Raa10]

A major goal of Elektra is to unify access to configuration, so applications do not have to know where a configuration is actually stored. This goal is backed by the concept of a *backend*, which implements the detail of actually accessing a configuration. A backend can be seen as the combination of several plugins, executed in a specific order.

**Storage Plugins** Storage plugins are used to read configuration settings in a given format from the system and expose it in Elektra's key-value format. They are also responsible for persisting keys in a system in a specific configuration format, a common example of such a file format is the *INI* format. The KDB stores all the information in a single configuration file per default. Users often want a more fine grained control over where different parts of the whole system configuration is stored. In UNIX-based systems it is usual to have an application's configuration reside in a folder called */etc*. A variety of different configuration file formats is common nowadays and some software is using customized file formats for their configurations. This creates the need for integrating all the different configuration formats into the global KDB, while giving the KDB the possibility to persist different parts of the global configuration into different files. This requirement is fulfilled by having the concept of *Mounting*, which resembles mounting file systems into other existing file systems. In the context of Elektra this means that different parts of the configuration can be mapped into a mountpoint. In Elektra storage plugins are utilized to take care about syntactic differences between the KDB and the target configuration mountpoint. [Raa10]

**Checker Plugins** Checker plugins ensure that keys fulfill certain pre- and post-conditions. In case there is a violation, such a plugin will prevent a key not fulfilling the conditions from being persisted to the KDB, issuing an error message instead. Therefore this kind of plugins can be utilized to keep a configuration valid, consistent and complete [Raa10]. Metakeys in the spec-namespace are used to specify for which keys of a keyset a checker plugin should be executed. Such guarantees are useful to avoid implementing the same check into each program that is able to modify a certain part of a configuration. [Raa10]

**Filter Plugins** Filter plugins are used for pre- and/or post-processing keys and values. Their main purpose is to deal with different encoding and decoding problems. Such problems are the conversion between different string encodings like ASCII, UTF, and base64. Such plugins should work in both directions. If we read keys encoded in a specific format, we want to be able to write modifications back in that format, and vice-versa. Another notable problem handled with filter plugins is the handling of null-values. A key can store a special value null that is different from containing an empty string. Depending on the context it can be desirable not to have such a distinction, so a filter plugin can take care of appropriate renaming of null values. [Raa10]

### 2.1.3 Configuration Specifications

One of Elektra's goals is the system-wide integration of application configurations via one universal interface while still maintaining modularity. However, software often is not part of a global integration strategy. Software usually only supports changing its settings through specific configuration files, or via graphical user interfaces at runtime. This makes system administration more complicated because each application has to be adjusted individually without a common denominator. Even if several applications share a common configuration file format like INI, there might be still semantic differences between them. For instance, one application may store a hypothetical update interval setting in seconds, while the other stores it in milliseconds. From the outside both configuration settings would look like numbers. Without documentation, reverse-engineering, or trial-and-error, it is often not possible to infer this extra unspecified semantic information. [Raa16]

Towards achieving this goal of system-wide integration of all applications, Elektra uses *configuration specification*s to externally describe the semantics of various configuration items independent of their syntax and target configuration file format. In Elektra a configuration specification is described by the configuration specification language SpecElektra.

#### SpecElektra

SpecElektra enables us to specify the semantics of unstandardized configuration file formats in an external und modular way. It is a semi-structured language built on top of Elektra's concept of keys and metakeys. Such a specification is simply a collection of

metakeys in the spec-namespace describing the properties of a part of the KDB. One example of such a part are the semantics of keys appearing in a mounted configuration file. Metakeys used to represent configuration specifications are called *specification metakeys*. However, as we generally refer to specification metakeys in this thesis, we leave the prefix "specification" away for brevity. A notable property of SpecElektra is that it has hardly any built-in language constructs. Following Elektra's goal of modularity, the actual functionality of such configuration specifications is realized using plugins for processing the metakeys in appropriate ways. Further notable differences to other configuration specification languages are that a specification is intended to be present on every system making use of Elektra. Therefore a specification can be dynamically rechecked and altered to meet system-specific goals. A configuration specification based on SpecElektra also specifies how configuration items are accessed. It aims to be simple to use. [Raa16]

Another way to view specification keys is that they represent different contexts. Each context depends on the requirements it describes. Contexts are usually composed through a combination of layers, each layer being associated with a certain run-time behavior. To connect configuration items present in the KDB in new ways, we can also use configuration specifications to describe existing data in new ways by altering the contextual composition. This way only small parts of a system's configuration have to be adjusted when contexts change. [Raa16]

SpecElektra tries to improve modularity in two different dimensions, *vertical* and *horizontal modularity*. Vertical modularity describes the separation between different applications throughout a system. Applications using a common configuration provider, like the *Registry* present on the Windows operating system, are tightly coupled through this provider. It is not possible to exchange the configuration provider to other means, for instance to configuration files, without changing the applications. Therefore SpecElektra uses three concepts to improve this kind of modularity. One approach is that applications continue to access their respective configuration files directly, but they are then also exposed via Elektra to other applications. This way while the main application still accesses the configuration file directly, other applications can make use of its configuration items in a standardized way without having to know about the underlying access. The second approach is that an application uses an adapter library to integrate with Elektra directly by translating the required key structures. While this requires changes in the original application, it is possible to use code generation techniques to create code that matches the underlying configuration specification directly. Thus SpecElektra can enforce a specification's properties. The third way is to intercept an application's configuration library, replacing it with an implementation that uses Elektra instead of the original way of modifying its configuration. This way the application does not have to be modified but can fully benefit from Elektra. [Raa16]

The second degree of modularity is horizontal modularity. This refers to how modular a single application's configuration access code is. Ideally it is possible to combine any configuration access code in a modular way, support code reuse and to abstract differences in the way each application accesses configuration. Some applications may

include comprehensive validation checks and others use custom file formats, yet Elektra aims at providing a uniform access to them. This is not solely achieved via the uniform key interface provided by Elektra. SpecElektra helps by combining several generic metakeys with each other. This procedure can be seen as analogous to the so called pipes-and-filters pattern known by various UNIX-like operating systems. The actual implementation of these metakeys via plugins is separated from the specification itself. Several plugins may fulfill the requirements of a metakey, so the best plugin under the given circumstances can be used. As mentioned in Section 2.1.2 plugins provide a contract. In this contract a plugin may define which metakeys it will handle. SpecElektra allows us to specify how mountpoints integrate different configurations into the KDB by specifying plugins required for the mounting process. Cross-cutting concerns can be implemented which effect a whole part of the KDB as opposed to individual keys. A notable example of such a concern is for instance the en- and decryption of configuration items. [Raa16]

As we can see SpecElektra is an important piece towards Elektra's vision of a modular system-wide integration of configuration. The unified configuration access enables having a single API to modify the configuration of a whole system. It is possible to automatically generate suitable user interfaces for nearly any configuration item just based on SpecElektra's abstraction of a configuration item's semantics. Due to the vertical modularity provided by SpecElektra different applications do not have to be coupled to work together in a system. There are generic ways to access common configuration file formats like INI or *JSON*. A configuration specification language is important for integrating all applications together by following the pipes-and-filters pattern to view existing configuration items in a different context. Combined with the possibility of specifying cross-cutting concerns with generic plugins, SpecElektra provides a solid foundation for specifying configuration specifications. [Raa16]

#### Sharing Configuration

Another use case of configuration specifications is sharing. There are not many factors that determine whether an application is integrated in a certain software system. The main aspects are logging, user interfaces and user interactions like key shortcuts. While these points are usually configurable in modern-day software, these configurations often only apply to a single application, or to a closed software system like a specific desktop environment. However, an integration crossing those boundaries towards a system-wide global integration is usually not possible. Therefore this has to be done several times in different configuration files, which is a cumbersome and error-prone process. [Raa15]

To cope with this shortcoming, Elektra introduces some additional metakeys describing the relationship between different keys, namely *fallback*, *override* and *default* [Raa15]. This concept is called *links*. As keys are often not directly compatible, there is an additional category of metakeys called *transform*, describing how one key can be converted to another key while preserving its respective semantics. Corresponding metakeys are added to a configuration specification, so these possibilities are not built into the core of Elektra.

As the name suggests, the metakey *fallback* can be used to specify a list of arbitrary length of other keys. In case the annotated key contains the special value null upon a lookup, Elektra will use the value of a linked key, respecting the order of the keys in the configuration specification. If no key specified by fallback has a non-null value either, it will still result in null. Complementing *fallback*, the metakey *override* follows the same but inversed principle. If any of the keys specified in a key's *override* metakey has a value, that value will be used instead of the current value of the annotated key. The metakey *default* is used to specify a default value in case there is neither an existing value in any of the keys specified by *fallback*, nor a value in any key overriding the one annotated with *override*. The values covered by an *override* or *fallback* mapping may share the same logical information, but their format can be different. Therefore one can use the *transform* metakey to specify how to translate from a given key to another key. A key associated with *transform* does not get persisted but is just a virtual construct to allow linking in all kinds of situations. [Raa15]

**Type Checking**

Given the possibility offered by the configuration specification language SpecElektra, described in Section 2.1.3, it is essential to provide methods of validating such configuration specifications. Generally one can view a key's specification as its *type*. A type is basically an abstract representation of its semantics. In our case the semantics are described via metakeys. Hence keys that share the same properties can be seen as having the same type. In general there are two different approaches regarding type checking, i.e. ensuring the validity of the semantics:

1. *Static* type checking happens when a specification is read. It verifies that configuration specifications themselves are valid, i.e., uses its types consistently.
2. *Dynamic* type checking happens when a configuration is stored or read. This is done by plugins that implement the functionality of metakeys. If a key to be read or written does not comply with its specification, Elektra will issue an error as this violation can lead to unwanted behavior.

This concept leads to the separationt of the metakeys residing in the spec namespace to the actual validation of the respective target keys in other namespaces. First it utilizes a plugin called *spec*. This plugin copies all the metakeys of a configuration specification into the target keys, thus effectively preparing the checks. Structural checks are executed in this phase, these kind of checks ensures that keys can have interrelations between them. The second phase executes the plugins on the keys before they are getting written back into the KDB, preventing the commit in case a validation fails and thus resembles dynamic type checking. [Raa10]

Static type checking is especially interesting in a global integrated scenario where new configuration items can be formed by linking or transforming existing configuration items

in new ways, as explained in Section 2.1.3. Using static type checking for specifications ensures that only compatible configuration items are linked. This prevents possible runtime inconsistencies beforehand as common pitfalls can already be detected when a configuration specification is being written. Imagine different units of measurement. Assume a configuration item that is intended to represent seconds, and another one that represents milliseconds. While both are numbers, their semantics differ. It is easy to see that they should not be linked without a proper transformation that converts between the two different units of measure. Such a mistake is difficult to detect at runtime if there is only a check whether the two configuration items are numbers. Therefore it is important to specify such semantical differences already in the configuration specification and additionally having a way of checking such specifications for their validity.

Currently there is no way in Elektra to statically check configuration specifications. It is the goal of this thesis to develop an extensible type system supporting the validation of specifications described using SpecElektra. This type system has to comply with the modular nature of Elektra and work on the level of metakeys. It abstracts away the underlying plugins implementing the effects of such metakeys. The implementation of these metakeys may be written in various programming languages with different and incompatible type systems. These type systems are usually specialized to the general-purpose needs of programming languages.

**Advantages**

As [Raa15] states, a configuration specification combined with a global configuration framework instead of application-specific configuration handling yields several advantages. As an application's configuration resides in a global database, it can be easily edited with external tools and not only inside an application itself. While this makes it easier for advanced users to configure their system to their liking, it also helps administrators in deploying configuration for a system environment. A configuration specification makes it possible to apply several constraints on single configuration items. Applications accessing such configuration items can already assume that the constraints are enforced, thus they do not have to check constraints again.

## 2.2 Type Systems

To help software developers ensuring that an application works correctly, a variety of formal methods has been developed. The power of such formal methods varies. There are complex logical frameworks like denotational semantics that can be utilized to prove various properties. The downside is that such advanced techniques can hardly be applied automatically to a program, and the user needs a deep and thorough understanding of how to utilize them. Therefore various more lightweight techniques have been developed which can be easily implemented directly in a compiler. Another goal was that those techniques can be used without having to have a deep understanding of the logical principles behind them. The most well-known example of such a lightweight verification

technique is a type system. While a type system is typically associated with programming languages, these use cases are basically just an implementation of various techniques developed in the field of *type theory*, incorporating both mathematics and logic. [Pie02]

### 2.2.1 Advantages

According to [Pie02], making use of a logical framework such as a type system yields several advantages. While due to the origin of type systems most of these advantages seem to only apply to their use inside programming languages, they also have applications in further disciplines such as computer security, theorem proving and database safety.

The most prominent advantage is that type systems support catching common mistakes in advance, before a program is actually executed. [Pie02] states that programmers using such languages that make a rich use of types tend to work once checking the types is successful. Catching mistakes in advance is also the most useful application of type systems for specifications. According to [LP99] it is important that specifications get checked automatically using a type checker for instance. They state that in practice specifications are not validated otherwise. A second advantage is that types aid in creating useful abstractions. For instance types can be used to describe the interface of certain parts of a program, thus already expressing various constraints of the interface without having the programmer to know its exact implementation. Therefore they document programs additionally and can make some comments obsolete. Type checking occurs on every change, forcing the types to be correct. Opposed to simple textual comments the documentary value of types is enforced to be up-to-date as such. Ultimately this leads to the notion of *language safety*. While this seems to coincide with the ability to prevent certain kinds of errors, it actually refers to a language's ability to enforce its own abstractions. Depending on the expressiveness of the used type system, some properties cannot be guaranteed when compiling the program, such as boundaries of an array. To make up for that it is possible to add additional checks in a program's execution which enforce the behavior which is formulated by types. Ultimately type systems allow more efficient programs by enabling a series of further optimizations depending on certain guarantees that can be ensured by using types.

### 2.2.2 Simply Typed Lambda Calculus

Over time different techniques and approaches have been developed for type systems. The design space of a type system can be seen as a trade-off between various boundaries, such as the ease of use, the time- and space-complexity of the type checker, the expressiveness of a type system, and many more [Pie02]. A popular formal foundation for type systems is the *lambda calculus*, which can be used to describe the definition of mathematical functions combined with their parameters. While the lambda calculus itself only expresses whether a function can be calculated or not, it cannot be used to formulate logic properties that a computation has to fulfill. Due to that reason this formulation is known as the untyped lambda calculus. For expressing certain logical properties on top of the lambda

calculus, typed variants have been developed. Simply speaking the main principle of a typed lambda calculus is that parameters of a function can be enriched with certain labels (the types), and a series of inference rules governs how those parameters can be used together in the context of a function.

The basic rules constituting the simply typed lambda calculus, often referred to as $\lambda_{\rightarrow}$, are shown in Figure 2.1. Note that we present the *pure* simply typed lambda calculus only involving functions, lacking any actual types likes numbers or boolean values. These additions can be easily added later on to the pure version of $\lambda_{\rightarrow}$ and may be proved separately. As we only want to show the basic principle of the simply typed lambda calculus, we decided to omit such extensions.

We first express the syntax of a language with *expressions* and *values*, followed by a collection of *transition functions* that specify how an expression translates to another expression until the system finally reaches a value. These transition rules represent the semantics of $\lambda_{\rightarrow}$ and are expressed using *Small-Step Style Operational Semantics* in our case, a variant of *Operational Semantics*. This representation specifies semantics for languages by assuming some abstract machine executing it. This abstract machine "executes" our language following the rules specified by our transition functions. They define how an expression in a language, representing the current state of execution, gets mapped to the next state, i.e. the next expression. Another common way of specifying semantics is *Denotational Semantics*, which does not assume a series of states being executed by an abstract machine, but map expressions to mathematical objects. Therefore it abstracts over the details of evaluation and shows the main concepts. The last common way of semantics specification is via *Axiomatic Semantics*. Opposed to the other two approaches it does not define the behavior of a language and then introduce rules based on that behavior. Instead it introduces rules directly which dictate the behavior. While denotational and axiomatic semantics seem more powerful at first, due to their more mathematical nature, they tend to be complicated to handle for certain properties of a type system such as procedures or concurrency.[Pie02]

As we will stick to the described notation first seen in Figure 2.1 in the course of this thesis, we give a explanation upfront. Note that our representation differs slightly from the original one in [Pie02] to match the notation used in the HM(X) framework. Instead of *terms* $t$ we say *expressions* $e$, and instead of types $T$ we say types $\tau$. The first part describes the syntax of our lambda calculus written in a variant of the *Backus-Naur-Form (BNF)*, a common meta language for describing context-free grammars. It basically represents an inductive definition of abstract syntax trees of our lambda calculus, including:

**expressions** arbitrary expressions in our lambda calculus

**values** fully normalized expressions. An expression is normalized if it matches no evaluation rule. Values are often not treated as a separate syntactical domain,

however [Pie02] uses this approach to allow an easier understanding of evaluation rules

**types** containing the possible types, which is only the function type in our simple calculus

**contexts** storing information about variable bindings during type checking

The evaluation rules are written as derivation rules. The upper part of a derivation rule above the line represents conditions, the lower part the result of the given evaluation step. When evaluating some arbitrary expression in our variant of the simply typed lambda calculus, we always apply the rule that fits in each step. Fitting means that the syntax of the expression applies to one of the evaluation rules, so the front part under the line of each derivation rule before the $\Rightarrow$ arrow. The $\Rightarrow$ arrow can be read as "evaluates to" and it is used to specify the result of a rule given that its premises are fulfilled. Only one rule per step is evaluated to keep the computation deterministic. We encounter the symbol $\vdash$ in the presence of contexts $\Gamma$. This symbol denotes entailment, meaning that the right side of the entailment is true under the assumptions being specified on the left side of it, which is often a context containing current assumptions of types during some state of evaluation. Obviously if the assumptions do not conclude the right side of the entailment, the rule does not fit.

Therefore *E-App1* from Figure 2.1 reads as "If $e_1$ evaluates to $e_1'$, then the application of $e_2$ on $e_1$ evaluates to $e_1'e_2$". As we can see, nothing got actually applied but $e_1$ got evaluated one step further. Note that $e$ and derivations like $e_1$ represent the corresponding part of the syntax, expressions in that case. The same principle applies for the other so-called metavariables used in derivation rules. At some point it is no longer possible to evaluate $e_1$ any further. It is then represented in its normal form $v_1$. Hence *E-App2* reads as "If $e_2$ evaluates to $e_2'$ then the application of $e_2'$ on $v_1$ evaluates to $v_1$ $e_2'$". This already implies the order in which expressions are evaluated. [Pie02] uses this approach to allow beginners in the domain to easily understand the process of evaluation. From a type checking point of view, the order of evaluation is irrelevant as long as the typing rules are obeyed. The last evaluation rule *E-AppAbs* is slightly more interesting, it has no preconditions so the only implied condition is that the syntax matches. It reads as "If we have a lambda abstraction which expects a parameter $x$ of type $\tau_{11}$ with the expression $e_{12}$ as its body, and we apply an arbitrary value $v_2$ to it, then we get the expression $e_{12}$ where every occurrence of the bound parameter got replaced with the value $v_2$".

Last, the typing rules are also written as derivation rules similar to the evaluation rules. Upon evaluation of an expression, these rules have to be evaluated. If there is a possible evaluation rule of a step of a computation, but no possible typing rule applies, then the computation is treated as being stuck and cannot continue without breaking our rules. We also see how the context syntax is now introduced in the rules, which basically stores information about free variables and their type assumptions in an evaluation steps. The first typing rule *T-Var* simply states that if the type assumed for $x$ is $\tau$ in the current

**Syntax** *Meaning*

| | | | |
|---|---|---|---|
| $\langle v \rangle$ | $::=$ | $\lambda x : \tau . e$ | fully normalized expression with regards to a type |
| $\langle e \rangle$ | $::=$ | $x$ | variable |
| | $\mid$ | $\lambda x : \tau . e$ | abstraction with regards to a type |
| | $\mid$ | $e\ e$ | application |
| $\langle \tau \rangle$ | $::=$ | $\tau \rightarrow \tau$ | function type |
| $\langle \Gamma \rangle$ | $::=$ | $\emptyset$ | empty context |
| | $\mid$ | $\Gamma, x : \tau$ | expression variable binding |

$\langle v \rangle$ ... values $\langle e \rangle$ ... expressions $\langle \tau \rangle$ ... types $\langle \Gamma \rangle$ ... contexts

**Evaluation Rules**

$$\text{E-App1}\ \frac{e_1 \Rightarrow e_1'}{e_1 e_2 \Rightarrow e_1' e_2}$$

$$\text{E-App2}\ \frac{e_2 \Rightarrow e_2'}{v_1 e_2 \Rightarrow v_1 e_2'}$$

$$\text{E-AppAbs}\ \frac{}{(\lambda x : \tau_{11} . e_{12}) v_2 \Rightarrow [x \mapsto v_2] e_{12}}$$

**Typing Rules**

$$\text{T-Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\text{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e_2 : \tau_1 \rightarrow \tau_2}$$

$$\text{T-App}\ \frac{\Gamma \vdash e_1 : \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 : \tau_{11}}{\Gamma \vdash e_1 e_2 : \tau_{12}}$$

Figure 2.1: The Simply Typed Lambda Calculus according to [Pie02]

context $\Gamma$, then the context entails the variable of the given type, thus the typing rule is correct. The next rule *T-Abs* is a bit more complex and describes the typing semantics of function abstraction. It can be interpreted as "If the variable $x$ of type $\tau_1$ is entailed in the expression $e_2$ of type $\tau_2$ given the current context $\Gamma$, then the context entails the abstraction $\lambda x : \tau_1 . e_2$ which has the type $\tau_1 \rightarrow \tau_2$". Last, the rule *T-App* deals with the typing semantics of function application and states "If the expression $e_1$ evaluates to a function that maps arguments of the type $\tau_{11}$ to results of type $\tau_{12}$, and the expression $e_2$ has the type $\tau_{11}$ in the current context, then the application of $e_2$ on $e_1$ will yield a result of type $\tau_{12}$".

**Proofing the Simply Typed Lambda Calculus** After having formulated a variant of a type system, it is generally interesting to mathematically proof that the properties expressed in a type system "make sense". According to [Pie02] this is done by proofing its *safety*, also known as *soundness*. This can be seen as the proof that any expression which corresponds to the rules expressed with our type system cannot "go wrong" and will at least successfully be reduced to a final expression. To guarantee that this will happen two theorems have to be shown. The first theorem is called the *progress* theorem. This

means that an expression will not get stuck, so it is either a value or a further evaluation step applies. The second theorem is called the *preservation* theorem, this means that in case an expression gets evaluated further, these intermediary steps always have to remain well-typed and not only the final result. These theorems are typically proofed using the proof technique of induction.

### 2.2.3 The polymorphic lambda calculus (System F)

In Section 2.2.2 we looked at a variant of the simply typed lambda calculus which can be used to describe the evaluation and typing semantics of a simple form of functional abstraction. As one can imagine, the simply typed lambda calculus serves mainly as a theoretical foundation and there is a number of more powerful variants of it, which actually serve as the foundation of whole programming languages.

[Bar91] tried to classify the expressiveness of such typed variants of the lambda calculus with the *lambda cube*. As it can be seen in Figure 2.2 the cube's axes represent a way of abstracting the relationship between values (acting as a function's parameter) and their respective types. The different variants of lambda calculi are increasing their power following the direction of the arrows starting from the simply typed lambda calculus $\lambda_{\rightarrow}$ , but therefore their complexity increases. In $\lambda_{\rightarrow}$ values can only depend on other values, but more complex relationships are not possible. Following the vertical axis we get the second order lambda calculus $\lambda_2$, also known as *System F*, where values can additionally depend on types, commonly known as *polymorphism* in programming languages. Following the first horizontal axis we get types which depend on other types, so called *type operators* known as $\lambda_{\underline{\omega}}$. Type operators are like normal functions, except they exist on the type-level and thus can be used to add additional rules to an existing type system. The other horizontal axis leads to types depending on values, commonly known as *dependent types*. An example usage of dependent types is the division operation. It is commonly known that a division through zero is not defined and thus not possible. Using dependent typing one could express this by adding the constraint "a number that is not zero" for the divisor, so the type of the divisor depends on the value. The main difference to normal programming languages, which usually handle this case with exceptions or additional checks, is that this property can be statically guaranteed and thus an additional check is not necessary. The downside is that depending on the conditions it is often hard to prove such properties, and manual proofs by the programmer are often required. As Figure 2.2 shows these three directions can be arbitrarily combined, steadily increasing the expressiveness. For instance, the combination of $\lambda_2$ and $\lambda_{\underline{\omega}}$ is known as $\lambda_{\omega}$, or *System* $F_{\omega}$, forming the theoretical foundation for advanced functional programming languages like Haskell. The lambda cube originates back to 1991 and is primarily used for languages based on the lambda calculus. Object oriented programming languages usually make use of *Subtyping*, a technique used to express a hierarchy between types. There are systems built upon the lambda calculus describing such techniques, for instance *System* $F^{\omega}_{<:}$.

For this thesis we have chosen to use a variant of System F as the formal basis for our

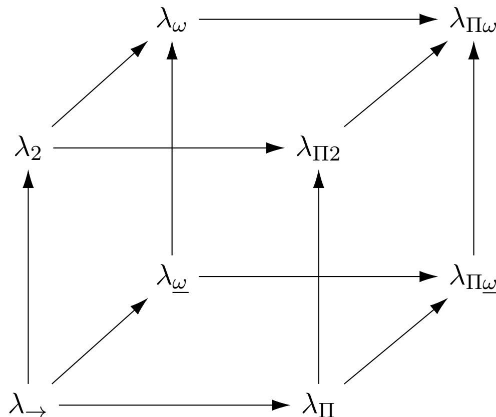



Figure 2.2: The Lambda Cube according to [Bar91], showing the relationship of expressiveness between various variants of the lambda calculus

own type system. More specifically, we use a subset of it known as the *Hindley-Milner* type system [Pie02]. A Hindley-Milner type system is a subset of System F that allows full *type reconstruction*. Type reconstruction, also known as *type inference*, is the task of inferring the type of an arbitrary expression from the context without having to explicitly specify it. Type inference in Hindley-Milner style type systems typically works by creating unification problems out of a term. It is always decidable while yielding efficient type checking algorithms. One of the original algorithms for a Hindley-Milner type system is *Algorithm W* [DM82]. Large and complex configurations can be checked efficiently. The functional model without state is suitable for our proposed type system. We primarily want to describe the effects of checker plugins, links and transformations as explained in Section 2.1.3. We do not intend to perform any actual computations, though it is possible. This can be easily expressed as a chain of function applications.

As [Gre31] states it is a difficult objective to create a customized type system. Therefore [Gre31] proposes the idea to take an advantage of existing type systems by utilizing a certain subset of it to express domain-specific needs. We also use this approach to implement our own type system by expressing it in the type system of the functional programming language Haskell. Therefore we can implement our own type system without having to develop our own type checking- and inference algorithms, which is a non-trivial and time-consuming task.

According to [WHE13] the *Glasglow Haskell Compiler (GHC)*, a popular compiler for Haskell, uses a type system called System FC. It is an extension to System F$\omega$ with other ideas from System F$\Pi$. System F$\omega$, being an extension to System F, allows us to express everything that we can express in System F. So ultimately we can make use of System FC to implement our own simple Hindley-Milner type system. We can take advantage of the expressiveness of System FC to implement a few domain-specific extensions for our type system for the target usage. This approach gives us quick prototyping and

exploration of typing techniques inside the design space of System FC.

### 2.2.4 HM(X)

In [Sul00] a generic framework for Hindley-Milner type systems gets introduced called *HM(X)*. It allows us to retrieve a domain-specific type system by parameterizing the framework with a specific *term constraint system* describing the types and their relations. As introduced in [Sul00], a term constraint system is a *cylindrical constraint system* with a special notion of substitution suitable for expressing type systems. A cylindrical constraint system is a quadruple $(\Omega, \vdash^e, \mathit{Var}, \{\exists\alpha|\alpha \in \mathit{Var}\})$. The first two elements $(\Omega, \vdash^e)$ form a *simple constraint system.* A simple constraint system is a structure $(\Omega, \vdash^e)$. $\Omega$ is a set consisting of both tokens and primitive constraints that are built on top of tokens. The entailment relation $\vdash^e \subseteq p\Omega \times \Omega$ describes the entailment of an arbitrary finite subset of $\Omega$ to the primitive set $\Omega$. *Var* is an unbounded set of variables, and the projection operator is an operator that allows to replace variables with concrete tokens or primitive constraints. A term constraint system over a term algebra $T$ requires predicates $p(\tau_1, ...\tau_n)$ to only work on terms $\tau$ from the term algebra $T$. One of those predicates has to be an equality predicate $\tau_1 = \tau_2$.

The main advantage of using such a framework is that it is easy to experiment with various constraint domains while having a generic basis. This eases the burden of proofing the type system. The generic basis provided by HM(X) has already been proven to be sound. Domain-specific instances of HM(X) are sound as long as the underlying constraint system is sound. This is shown by proving a few axioms about the constraint system. It is not required to fully proof every part of the type system. There is already a generic type inference algorithm available in case the *principal type property* can be shown. The principal type property means that for an arbitrary program the most general type can be given to, so that all other possible types are just instances of that type. [Sul00]

The basic logical system behind HM(X) is shown in Figure 2.3. The syntax basically resembles a standard Hindley-Milner type system based on the lambda calculus with let-polymorphism. Values are defined as variables $x$, constants $c$ and the basic lambda abstraction $\lambda x.e$. Expressions are either values $v$, application $e\ e$ or let statement $let\ x = e\ in\ e$. Types $\tau$ contain a placeholder in the generic basis, meaning it is intended to be extended to support a domain-specific type language. Types either consist of type variables $\alpha$ or function types $\tau \to \tau$. Last, *type schemes* $\sigma$ are either a specific type $\tau$ or a type scheme $\forall\alpha.C \Rightarrow \omega$ that acts as a placeholder for types fulfilling the imposed constraints $C$. The quantifier allows us to bind type variables $\alpha$ into the constraints. An additional restriction is that such constraints always have to be satisfiable. Generally speaking types $\tau$ are called *monomorphic* types. This means there is only one specific representation for a type that is only equal to itself. Conversely type schemes $\sigma$ are called *polymorphic* types, meaning that a polymorphic type describes a range of monomorphic types given certain constraints. A *typing judgement* is a judgement of the form $\gamma \vdash e : \sigma$. The context $\gamma$ describes how variables are bound to types in a context. Such a judgement

is valid in case the given context $\gamma$ implies the type scheme $\sigma$ for a given expression $e$ is derivable through the typing rules. [Sul00]

The typing rules, as shown in Figure 2.3, mostly correspond to standard typing rules of a Hindley-Milner type system. We are shortly going to recap them. *T-Var* states that a variable $x$ of schema $\sigma$ has to appear in the typing context $C, \Gamma$. *T-Sub* expresses subsumption and means that an expression $e$ of type $\tau'$ is entailed in the context if the context entails the same expression of type $\tau$ and that $\tau$ is smaller or equal to $\tau'$ according to the subsumption relation. *T-Abs* expresses abstraction and states that $\lambda x.e$ is of type $\tau \rightarrow \tau'$ if the given context entails the type $\tau'$ given some $x$ of type $\tau$. The notation $\Gamma_x$ refers to the context excluding the variable $x$. *T-App* describes application and means that the application of an expression $e_2$ onto an expression $e_1$ yields the type $\tau_2$ if $e_1$ is a function type $\tau_1 \rightarrow \tau_2$ and $e_1$ is of type $\tau_1$. *T-Let* describes let bindings and states that the expression $e$ bound by the binding variable $x$ is entailed by the context if the type scheme $\sigma$ of the expression $e$ is entailed by the context and that the inner expression $e'$ of type $\tau'$ has to be entailed by the context given some type scheme $\sigma$ for the binding variable $x$. The last two rules are a bit more interesting. *T-$\forall$-Introduction* deals with the binding of type variables into a constraint. It builds type schemes by appending a new constraint $D$ into the inference chain. The existential quantification $\exists\overline{\alpha}.D$ ensures that the final constraint of an inference chain requires that all intermediate constraints are satisfiable. Technically speaking it states that the extended context by combining the existing constraints $C$ with the newly introduced constraints $D$ that follows from a type scheme on an expression $e$ of type $\tau$ is entailed by the context. This is only the case if the combined context entails the expression of type $\tau$ and the bound variables from the new constraints $D$ are not free in the existing contexts. *T-$\forall$-Elimination* then deals with the checking of such introduced constraints. It describes that when the type variables $\overline{\alpha}$ of a constraint $C$ get substituted with some actual types $\overline{\tau}$ the actual types have to satisfy the constraints specified by their type scheme. [Sul00]

As elaborated in [Sul00] an instance of HM(X) is defined using a five-tuple $(X, \preceq, T, S, \Gamma_0)$ that is usually just being referred to as X. The first parameter of the tuple $X$ is a term constraint system describing the domain-specific properties. $\preceq$ is a subsumption relation that has to fulfill reflectivity, antisymmetry, transitivity and contravariance. It has been added for formalizing type systems that have some kind of subsumption relation, for instance a subtyping relation. It can be an equality relation, as that fulfills all the four axioms. $T$ is the term algebra describing the basic building blocks of the type system. $S$ is the so called set of solved forms, i.e. constraints that may appear in actual typing judgments in the type system. The constraint language defined in $X$ may be richer than $S$. The last parameter $\Gamma_0$ describes the initial type environment and is used to define built-in operators and functions for the type system.

If it a given constraint domain has the *principal normal form* property, the *principal type property* is fulfilled. This allows us to benefit from a generic sound and complete type inference algorithm without having to design and proof one ourselves as described in [Sul00].

**Syntax** *Meaning*

| | | | |
|---|---|---|---|
| $\langle v \rangle$ | $::=$ | $c$ | constant |
| | $\mid$ | $x$ | variable |
| | $\mid$ | $\lambda x.e$ | abstraction value |
| $\langle e \rangle$ | $::=$ | $v$ | value |
| | $\mid$ | $e\ e$ | application |
| | $\mid$ | $let\ x = e\ in\ e$ | let statement |
| $\langle \tau \rangle$ | $::=$ | $\alpha$ | type variable |
| | $\mid$ | $\tau \rightarrow \tau$ | function type |
| | $\mid$ | ... | further types for the target domain |
| $\langle \sigma \rangle$ | $::=$ | $\tau$ | type |
| | $\mid$ | $\forall \alpha. C \Rightarrow \sigma$ | type scheme |

$\langle v \rangle$ ... values $\langle e \rangle$ ... expressions $\langle \tau \rangle$ ... types $\langle \sigma \rangle$ ... type schemes

**Typing Rules**

$$\text{T-Var}\ \frac{}{C, \Gamma \vdash x : \sigma}\ (x : \sigma \in \Gamma)$$

$$\text{T-Sub}\ \frac{C, \Gamma \vdash e : \tau \qquad C \vdash^e (\tau \preceq \tau')}{C, \Gamma \vdash e : \tau'}$$

$$\text{T-Abs}\ \frac{C, \Gamma_x.x : \tau \vdash e : \tau'}{C, \Gamma_x \vdash \lambda x.e : \tau \rightarrow \tau'}$$

$$\text{T-App}\ \frac{C, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1 e_2 : \tau_2}$$

$$\text{T-Let}\ \frac{C, \Gamma_x \vdash : e : \sigma \qquad C, \Gamma_x.x : \sigma \vdash e' : \tau'}{C, \Gamma_x \vdash let\ x = e\ in\ e' : \tau'}$$

$$\text{T-}\forall\text{-Introduction}\ \frac{C \wedge D, \Gamma \vdash e : \tau \qquad \overline{\alpha} \notin fv(C) \bigcup fv(\Gamma)}{C \wedge \exists \overline{\alpha}.D, \Gamma \vdash e : \forall \overline{\alpha}.D \Rightarrow \tau}$$

$$\text{T-}\forall\text{-Elimination}\ \frac{C, \Gamma \vdash e : \forall \overline{\alpha}.D \Rightarrow \tau' \qquad C \vdash^e [\overline{\tau}/\overline{\alpha}]D}{C, \Gamma \vdash e : [\overline{\tau}/\overline{\alpha}]\tau'}$$

Figure 2.3: The HM(X) system according to [Sul00]

As stated in [Sul00] the basic Hindler-Milner type system can be derived from the HM(X) framework by parameterizing $X$ to be a Herbrand constraint system over the types $\tau$, $\preceq$ to type equality based on their syntax, $\tau$ to be a set of types that can be compared using syntactic equality and $S$ to be the empty set of constraints that is trivially always satisfiable. As it can be seen it does not support any domain-specific constraints to appear. This instance is called *HM(HERBRAND)* in [Sul00].

CHAPTER 3

# HM(RGX)

Before expressing a type system for configuration specifications, we first need to evaluate our design space in Section 3.1. We formalize an instance of the HM(X) framework for our target domain called *HM(RGX)* in Section 3.2. While defining our instance we fulfill the proof obligations imposed by the HM(X) framework to show that our instance is also sound and supports a sound and complete type inference algorithm. We close this chapter by providing an example in HM(RGX) that we explain and type check manually according to our rules in Section 3.3.

## 3.1 Case Study

Towards answering our second research question presented in Section 1.2 we first evaluate which metakeys of SpecElektra are relevant:

**Research Question 2.** *How many metakeys of SpecElektra that are currently known to be used by plugins, i.e., specified in the* METADATA.ini *file of the Elektra project with the status of being implemented, can have their behavior described by our type system?*

First we categorize the metakeys in Section 3.1.1. Then we elaborate why we have chosen to create a Hindley-Milner style type system and evaluate how we model the domain-specific extensions that we need for expressing configuration specifications in Section 3.1.2.

### 3.1.1 Categorizing the Metakeys

We have categorized the currently existing metakeys in Table 1 in the Appendix. We sum up the results of this categorization in Table 3.1. We have recognized five categories of metakeys. The first category, referred to as $U$ in the table, stands for metakeys which are unrelated to the type system. Examples are simple comments or additional information

| Category/Status | implemented | proposed | reserved | idea | unclear | deprecated |
|---|---|---|---|---|---|---|
| U | 21 | 4 | 8 | 6 | 0 | 2 |
| L | 4 | 0 | 0 | 0 | 0 | 0 |
| C | 21 | 3 | 0 | 7 | 0 | 2 |
| T | 4 | 0 | 0 | 2 | 0 | 0 |
| S | 5 | 2 | 0 | 0 | 3 | 0 |
| **Relevant** | 34 | 5 | 0 | 9 | 3 | 2 |
| **Total** | 55 | 9 | 8 | 15 | 3 | 4 |

Table 3.1: An analyzation of the metakeys categorized in Table 1.

stored by plugins playing no role for configuration specifications. The second category $L$ stands for links, therefore referring to metakeys that define some kind of link between keys as mentioned in Section 2.1.3. The third category $C$ refers to checks imposed by checker plugins explained in Section 2.1.1. The fourth category $T$ relates to transformations between keys regarding their values. This is a concern when linking between keys, also explained in Section 2.1.1. This category also applies to plugins which do transform a key's value unrelated to links, such as the encryption or renaming of keys. The main difference to category $C$ is that a transformation replaces the current value and thus alters the types. It does not simply limit the values a key can contain. The category $S$ also refers to transformations, but this time it refers to structural transformations altering a key's structure, such as its name or the relationship between a key and other keys that are independent of its value, such as arrays.

As Table 3.1 shows of the 55 metakeys which are currently present and implemented, **34 are relevant** i.e., have not been categorized as unrelated to types. Therefore we try to design our type system in such way that a majority of those metakeys can be expressed with it. Most metakeys are simply used to store some additional information used by some plugins, or to add comments and similar meta-information. 21 of the implemented relevant metakeys fall into the category $C$ and thus resemble checker plugins. Other notable results are that there are only 4 metakeys which are related to links, all of which are implemented. Transformations make up only 4 relevant metakeys for our research question while there are 5 metakeys dealing with structural constraints.

### 3.1.2 Mapping the Categories to Type System Features

We categorized and analyzed the current metakeys of a configuration specification in Section 3.1.1. Based on that we now inspect which type system features we are going to use to model the effects of relevant metakeys of configuration specifications. First we decide the theoretical foundation for our type system. We already discussed this question at the end of Section 2.2.3. Afterwards we evaluate which type system techniques could be used to express the effects of different metakey categories.

### Theoretical Foundation

As indicated in Section 2.1.1 the effects of metakeys are implemented by plugins for Elektra, following its modular nature. An exception to this rule are the five linking metakeys, which are implemented directly in Elektra's core library [Raa15]. A configuration specification is enforced and realized by using a variety of different plugins combined as a backend. In Elektra all the different plugins of a backend are executed in a serial fashion according to a plugin's contract. Following this serial execution of plugins we can model their effects basically as a composition of function calls on a raw key, further refining or transforming its type in each function.

This leads us to the decision that a typed variant of the lambda calculus is a good basis for a type system for Elektra as it is suitable to describe functions and their types. A major aspect is that the lambda calculus is already used as the theoretical foundation for all kinds of different type system techniques and, built upon that, real-world programming languages. Therefore research has been going on related to it, leading to various publications. Using a variant of the lambda calculus to describe effects of metakeys as opposed to hard-coding a few typing rules for the existing metakeys gives us the big advantage that the type system is extensible. If we would choose to write typing rules for the existing metakeys in a direct, static fashion we would have to extend the type system each time a new relevant metakey gets introduced with rules for the new metakey.

We have already examined a few different variants of the lambda calculus with varying power, shown through the lambda cube explained in Section 2.2.3. We conclude that the polymorphic lambda calculus, System F, is the best compromise between expressiveness, complexity and ease-of-use. The simply typed lambda calculus is unable to express polymorphic functions, while polymorphic functions appear to be a suitable way to describe the effects of links between keys and transformations and form the basis for type inference [Pie02]. This leads to the decision of using it, more specifically, a subset of it known as a Hindley-Milner style type system, as a formal basis.

Following another axis of the lambda cube shown in Figure 2.2, we next take a look at $\lambda\underline{\omega}$. In order to keep the type system easy to use we conclude that the possibilities offered by type-level functions are already too advanced and complicated. As not many programming languages offer type-level functions most plugin developers are expected to be unfamiliar with that topic. The benefits are insignificant for typing configuration specifications, as the different needs of the various categories of metakeys according to Section 3.1.1 can be covered using polymorphism alone. It is however possible to include a few built-in type-level functions. According to [Pie02], by defining their meaning directly in the language specification we can avoid formulating a complete $\lambda\underline{\omega}$-style type system.

Last, we evaluate whether using the concepts of $\lambda\Pi$, dependent typing, would be useful for our type system. Indeed it would be interesting to express the effects of certain checker-plugins with the help of dependent typing. For instance, the implemented checker plugin *check/range*, tests if the checked key's value corresponds to a given range of

numbers. The type of the plugin would depend on a value, in that case, the range given to it. We could conclude facts like a key falling back to either a number in the range 1 to 3 or to a number in the range 2 to 7 via a link is a number in the range 1 to 7 from a type perspective. Another conclusion would be that different syntactical representations for ranges like `1-3` and `1,2,3` are semantically equivalent. It is necessary to decide the equality of dependent types in a configuration specification. As the effects of metakeys may dependent on arbitrary additional information, deciding such equalities is undecidable in general and often needs manual proofs by the implementor [XP99]. Therefore we conclude not to incorporate concepts of $\lambda\Pi$, though it would be interesting to explore such techniques in future works.

As discussed at the end of Section 2.2.3, it is possible to use a language with a richer type system to prototype our own type system. Wwe can implement it without having to write a complete type checker from scratch. We can experiment with all the type features prevalent in the host language. Future developments in the host language expand our own possibilities. As many functional programming languages support at least a variant of System F, often extended with various ideas from System F$\omega$ and/or System F$\Pi$, choosing System F seems appropriate.

We want to emphasize the fact that we do not aim to build an actual general-purpose programming language even though we use System F as the foundation. In Elektra plugins, defining the actual effects of metakeys, can be written in basically any programming language. It would be useful to express transformations or checks directly in our own language specification, already enforcing our proposed static typing. This would make the language design more complicated due to various additional language constructs having to be added to make it easy to program in it. Therefore we decide not to do this for now. We simply develop a type system for a small domain-specific metalanguage to check whether a typed configuration specification is sound.

#### Checks

After having decided the formal foundation of our type system, we now examine which typing techniques can be utilized to express effects of the current checker plugins. We will call those effects *checks*. As discussed in Section 2.1.1 a check ensures that a key's value will always correspond to some condition. This condition is enforced mainly upon writing a key to the KDB. A plugin corresponding to the given metakey will evaluate the value to be written, rejecting it upon violating the conditions expressed by the plugin. It is perfectly possible that a key fulfills various conditions at once, for instance, a key could be both a number in the range 1 to 3 and a valid ASCII string at the same time.

The first idea is to define a set of standard types that can be used to express the effects of various metakeys. As Elektra already provides a checker plugin implementing the specifications of the different CORBA types. CORBA is a standard for the communication between software independent of their implementation language and their hardware platform, covering a range of numerical and general-purpose types [Vin97]. A downside

of this approach is that we would need additional language constructs to build up more complex data types based on those basic data types, for instance records, to model more sophisticated data types such as IPv4 addresses. This also raises the issue of ambiguity. For instance an IPv4 address could be represented as a set of four numbers, each limited to the range from 0 to 255. It could also be represented using a plain string, a binary form, an array of digits or similar. Users would have to make a decision which representation to use, and once this representation has been chosen and is relied upon in our type system it is hard to change afterwards. It is not possible to express range checks in such a type system without having to include the definition of our ranges directly in the type system. Similar to range checks, it is not possible to include additional basic types that go beyond the scope of CORBA types into the type system without altering it. For instance, the C programming language already supports more data types natively than those provided by CORBA. It would be interesting to make use of more fine-grained types in certain scenarios. The non-modularity of CORBA conflicts with the modular nature of Elektra.

Building upon the idea of using CORBA types, we also had the idea to use concepts of subtyping. By declaring a hierarchy that defines that numbers are a subtype of ASCII strings and the range 1 to 3 is a subtype of a number, this would be easily possible. Usually in practical programming languages, nominal subtyping is used [MA09]. This goes against the modular nature of Elektra. A subtyping hierarchy has to be carefully defined in advance as stated in [MA08] when using nominal subtyping. Later changes will be hard to achieve. Structural subtyping on the other hand would enable more flexibility as it only considers whether two elements share the same traits. There still needs to be some kind of hierarchy defined [MA08]. We intended to express our type system in the functional programming language Haskell as discussed at the end of Section 2.2.3. This is a further argument against subtyping as that technique is not directly supported by Haskell.

Next we came to the idea of using a variant of *intersection types* in order to express the general idea of checks. According to [Pie02] an intersection type can be seen as a type that represents the intersection of two types $\tau_1 \wedge \tau_2$, so terms of type $\tau_1 \wedge \tau_2$ belong to both the types $\tau_1$ and $\tau_2$. Intersection types are very powerful and make the language design more complex. A key has no type by default, which is represented via the most general type *any*. Each check further restricts the type of a key. By adding a check that checks for the key containing only ASCII characters, we would effectively get the refined type $any \wedge ascii$, loosely following the general idea of intersection types. This key fulfills every constraint implied by the type *any*, but due to the check it is ensured that it will also fulfill the constraint *ascii*. By adding another range check, for instance a check for the range `1-3`, we get a new type that resembles the intersection of the previous types with the range check, which could look like $any \wedge ascii \wedge range\texttt{1-3}$. While this approach seems quite promising on the first glance, it fails to abstract any further information about the semantics of different metakeys in its raw form. For instance, the intersection type $any \wedge range\texttt{4-6} \wedge range\texttt{1-3}$ is valid considering the basic idea of intersection types. We see that there is no actual intersection between the ranges `4-6` and `1-3`

considering the semantics of a range check. We would have to include the semantics of various metakeys directly in our language. Again this conflicts with the modular nature of Elektra.

Last, we thought about using *regular expressions* (*regexes*) as a basic type in our proposed type system. Regular expressions are well-known in computer science and have a wide range of applications. A regex is a string describing a regular language. This basically means that there is no context involved in the language. In our use case we intend to use regexes to describe the content of our keys. As each key in Elektra represents a string in its untyped form, with the exception of binary data, this already seems promising.

A work that uses a type system based on regular expressions is called *Boomerang*. According to [BFP+08], Boomerang is a programming language that is specialized for the bidirectional transformation of string data, so called lenses. It defines a few general-purpose combinators for string transformations that are type checked using regular expressions to detect whether two different transformations are compatible with each other.

The configuration management tool Augeas uses regular expressions in order to validate configurations. As described in [Lut08], Augeas uses the type checker to ensure that modified configuration files are valid before transforming them. A downside is that regular expressions are not able to handle arbitrarily nested constructs, using the example of the *IfModule* construct of the *httpd.conf* of the well-known Apache Webserver. This is not directly an issue for the use with Elektra. When a configuration like that gets mounted into the KDB, storage plugins split it into different parts that are then validated key-per-key, avoiding the problem of nested constructs.

During our metakey categorization we noticed that many of them seem to be easily expressible as regular expressions. We can directly express the metakey *check/validation* as it resembles a regular expression check. We can describe all the basic CORBA types directly as numbers with limited ranges as regular expressions. We can also identify enumerations, range checks (basically a dynamic variant of numerical data types) and IP addresses to be expressible as regular expressions. Regular expressions already carry some semantics of the content they describe. We have already explained the issue of intersection types not being able to detect the incompatibility of the two range checks `4-6` and `1-3` in the above paragraph. With regular expressions it is possible to determine that there is no intersection between those two checks. The first check could be expressed as the regular expression `4|5|6` and the second as `1|2|3` . As they refer to different characters there is no regular expression that resembles the unification of both ranges. As stated in [BK93], a regular expression can be converted into the representation of a finite automaton and vice-versa. When modeled as a finite automaton, we can perform various operations such as the intersection between two regular expressions, the containment of a regular expression in another, and the union of two regular expressions [BL80].

Obviously we can only express regular languages with regular expressions. There are also irregular languages, such a dates, depending on a certain context. For instance the

number of days in the month February depends on whether the given year is a leap year or not, thus it is irregular. We can still capture the general form of a date using a regular expression ignoring such details.

We conclude that types based on regular expressions are a good compromise between real-world usability, expressibility and developer-familiarity for our type system when keeping our target domain in mind. As the theoretical foundation of our type system will be based on a variant of the lambda calculus, it can be extended with further typing techniques in the future.

#### Transformations

The next category of configuration metakeys, as categorized in Section 3.1.1, are *transformations*. Transformation metakeys are used to convert a key's value in a specific way depending on the metakey. They are usually applied on keys to transform the values they get assigned to different ones. They can be used to specify keys that are basically different views on existing keys under new key names.

Transformations can be handled using our existing type system ideas. A transformation basically takes a key of a given type and transforms it to a key of a different type. Such a transformation can easily be modeled using polymorphic functions that we use as a foundation for our type system. Transformations can make use of our regex types. We use the example of a hypothetical metakey that expects a key to be an ASCII string and hashes it, storing the result encoded as a hexadecimal number. Its function signature could look like `[\x00-\x7F]+` → `0[xX][0-9a-fA-F]+` , where the first regex represents the range of ASCII characters and the second regex represents a hexadecimal number prefixed with `0x` (case insensitive). The arrow in the middle acts as a separator for the two types. Suppose a key representing an integer number gets passed as the first parameter, for instance `[0-9]+` . As digits are a subset of the ASCII characters, this is perfectly valid.

#### Links

The third category of configuration metakeys as categorized in Section 3.1.1 are *links*. Links are used to specify certain relationships between keys. At the time of writing there are four link-related metakeys implemented. *Fallback* expresses that the value of the referenced key will be used in case the current key has no value. *Override* on the other hand is the complement of *fallback*. It states that the referenced key will be used instead of the referencing key in case the referenced key has a value. The last of the three metakeys is *default*, specifying a default value to use in case another key overrides the referencing key, the referenced key has no value and there is no *fallback* key with a value to use either. It is debatable whether a default value can be classified as a link. We concluded that a default value can be seen as a special version of *fallback* that does not refer to another key but to a given input value. The remaining two link-related metakeys

are *namespace* and *override*, but as they do not really alter the values a key can take they are irrelevant to the type system.

Links can be expressed via polymorphic functions. Following the example of *fallback*, a possible type signature could be expressed as $b \to a \to a$, where the type variable $a$ denotes the type of the referencing key and the type variable $b$ denotes the type of the referenced key. As we use type variables here instead of actual types, this type scheme can be used for any kind of fallbacks. The effect on the type system will be the same. The resulting type has to be compatible to the referencing key. Compatibility means that it represents the result of the override, which is either the key of type $a$ itself, or a compatible key it falls back to. Thus we naively assumed type $a$ to be the result type. However, there is more complexity involved into expressing links.

First of all, the compatibility between the two type parameters $a$ and $b$ has to be ensured. Therefore we treat such link-related functions differently compared to transformation functions. We propose to add an additional built-in compatibility check between regex types, expressed as a constraint for link functions. Constraints are used to restrict the input domain of a function. This check could be expressed using the symbol $\subseteq$, inspired by the mathematical subset notation. This then leads to the refined type signature of $b \subseteq a \Rightarrow b \to a \to a$. We use Haskell's notation of specifying constraints for functions using the $\Rightarrow$ symbol. The semantic of such a compatibility check is that the regex represented by the type variable $b$ either has to be the same regex as $a$, or is a subset of it. This ensures that the link is always safe in terms of typing rules.

We demonstrate these thoughts with a short example. Consider a key with the ASCII regex and a key with a hexadecimal number regex, both mentioned in Section 3.1.2. Now we link the ASCII key to the hexadecimal number key. The resulting function signature of the *fallback* with the types already applied would look like `0[xX][0-9a-fA-F]+` $\subseteq$ `[\x00-\x7F]+` $\Rightarrow$ `0[xX][0-9a-fA-F]+` $\to$ `[\x00-\x7F]+` $\to$ `[\x00-\x7F]+` . Note that the regex representing a hexadecimal number is a subset of the regex representing ASCII characters, making this link valid.

### Structural Types

The last category of configuration metakeys as categorized in Section 3.1.1 are *structural types*. Structural types are used to impose some kind of structure between keys, for instance that keys form an array of a specific size or that a hierarchy in the KDB contains keys of certain types. We also include structural transformations in this category, i.e., the renaming of keys.

There are various approaches of how to introduce structural types in a language. Many languages based on the lambda calculus include some kind of list, tuple and record notation to represent basic structures. Including such specific language constructs directly again interferes with the modular approach of Elektra. A more general approach to structural types is presented in [HVP05]. The authors try to express the structure of hierarchical tree-based data using operators commonly associated with regular expressions to express

repetition, optional occurrences and alterations. This approach is quite different from our idea of using regular expressions to describe the contents of a key as presented first in Section 3.1.2. In essence this corresponds to the concept of a *regular tree grammar* instead of regexes that resemble a *regular word grammar*. However, including structural types based on a regular tree grammar would introduce more complexity to our type system. The question remains how structural transformations can be treated in a language. Thus we have decided to skip structural types in our type system. They may be added in future extensions though.

## 3.2 Formal Definition

After having discussed the design space for our type system in Section 3.1.1, we now formalize our type system. We describe our type system based on a generic framework for Hindley-Milner style type systems called HM(X). This framework was introduced in Section 2.2.4. Afterwards we add the extensions that we need for our regex types in Section 3.2.1. Regex types act as our sole type for describing keys. Further, we explain the link between regular expressions and finite automata. We elaborate how various operations on finite automata, such as intersection and containment, can be expressed. We prove the necessary obligations as imposed by the HM(X) framework along the way. In Section 3.2.3 we describe how type inference is achieved and prove it.

In Table 3.2 we can see the metavariables that we are going to use in our formalizations. To keep our typing rules symbolic and concise, we will mostly use 1- or 2-letter variables to refer to different syntactical and semantical building blocks of our formal language.

| **Metavariable** | **Usage** | **Description** |
|---|---|---|
| $e$ | Expression | Expressions, can be reduced to values |
| $v$ | Value | Values, can not be reduced further |
| $\tau, \mu$ | Type | General and function types / Monotypes |
| $\alpha, \beta, \gamma$ | Type Variables | Placeholders for concrete types |
| $\sigma$ | Type scheme | Constrained types / Polytypes |
| $\tau_r$ | Regex Type | Regex types |
| $C, D$ | Constraint | A constraint that further restricts types |
| $\Gamma$ | Type Context | Context with current type assumptions |
| $r$ | Arbitrary Regex | |
| $s$ | Arbitrary String | |
| $p$ | Elektra Key Name | A valid key name |

Table 3.2: Metavariables Overview

### 3.2.1 Defining RGX

In this section we are going to gradually introduce our *regex types*. A regex type describes the type of a key in the global KDB as explained in Section 2.1. We have already discussed the basic idea of our regex types in Section 3.1.2. Typically a type system gets introduced along with some kind of programming language. A common way of defining a typed programming language is to specify its grammar, evaluation rules that describe how the different constructs of a programming language get evaluated at the time of execution, and typing rules specifying further rules of how the programming language constructs can be used with each other [Pie02]. Using regex types we can statically check a configuration specification according to the rules of our type system without having to deal with the actual implementation. This is due to the fact that the implementation of checks or transformations is done using plugins in various programming languages. Consequently, there are no evaluation rules required for our use case. We define a type system for configuration specifications but not a complete programming language. This is supported by the fact that the HM(X) framework, introduced in Section 2.2.4, is only a type system framework. It only specifies basic grammar and typing rules along with a type inference algorithm. We continue by defining the five-tuple for our domain-specific extensions that parameterizes the HM(X) framework. We will refer to this tuple as *RGX*. Therefore we call our instance of the HM(X) framework the HM(RGX) type system.

First we define our term algebra $T$, the third parameter of the five-tupel RGX. A term algebra describes valid type terms and can be seen as some kind of syntax definition. It is similar to the well-known *Backus-Naur form*, a general notation technique for writing context-free grammars. We follow the same concept of how a term algebra for representing physical dimensions is described in [Sul00]. The term algebra describing physical dimension forms a two-sorted term algebra. This means that it distinguishes between ordinary types, like type variables and function types, and dimension types. The main consideration is that dimension types are a special kind of ordinary types tagged with a certain type constructor. One cannot use ordinary types to build up more complex dimension types. These two domains are separated so the number of cases that have to be proofed are limited when reasoning about the type system's soundness. We define $T$ to be a two-sorted term algebra consisting of *regex types* and ordinary types in Figure 3.1. Both $\tau$ and $\tau_r$ have the same notion of type variables that are called $\alpha$. This is modeled the same way as in the dimension type system, so we can distinguish between regex type variables and ordinary type variables syntactically. Regex intersection types can contain type variables. The type inference process itself does not distinguish between those different kinds of type variables thus they have the same notion. The remaining terms are equal to the generic ones presented in Figure 2.3 that already exist in the HM(X) framework, except for *T-Const*. *T-Const* simply defines that the only constants/values we have in our type system are key constants that need an explicit type assignment using the :: notation. This is the only place where explicit types have to be specified in HM(RGX). The remaining constraints and types get inferred from the context.

We define our constraints $C$ that can be used inside our adjusted term constraint system

| | | | |
|---|---|---|---|
| $\langle v \rangle$ | $::=$ | ... | |
| | $\mid$ | Key $::$ $r$ | typed key constant |
| $\langle \tau \rangle$ | $::=$ | $\alpha$ | type variable |
| | $\mid$ | $\tau \rightarrow \tau$ | function type |
| | $\mid$ | Rgx $\tau_r$ | regex type |
| $\langle \tau_r \rangle$ | $::=$ | $\alpha$ | type variable for regex types |
| | $\mid$ | $r$ | string specifying a regex |
| | $\mid$ | $\tau_r \cap \tau_r$ | regex intersection type |
| $\langle C \rangle$ | $::=$ | $\{\}$ | empty constraint set, always satisfiable |
| | $\mid$ | $C \wedge C$ | logical conjunction |
| | $\mid$ | $\exists \alpha . C$ | type variable quantification |
| | $\mid$ | $\tau = \tau$ | type equality predicate |
| | $\mid$ | $\tau_r \subseteq \tau_r$ | regex containment predicate |
| | $\mid$ | $\mathit{Intersectable}\ \tau_r$ | regex intersection predicate |
| $\langle S \rangle$ | $::=$ | $\{\}$ | empty constraint set, always satisfiable |
| | $\mid$ | $\exists \alpha . C$ | type variable quantification |
| | $\mid$ | $C \wedge C$ | logical conjunction |
| | $\mid$ | $\tau_r \subseteq \tau_r$ | regex containment predicate |
| | $\mid$ | $\mathit{Intersectable}\ \tau_r$ | regex intersection predicate |

$\langle v \rangle$ ... values $\langle \tau \rangle$ ... types $\langle \tau_r \rangle$ ... regex types $\langle C \rangle$ ... constraints $\langle S \rangle$ ... solved forms

**Typing Rules**

$$\text{T-Const}\ \frac{}{C, \Gamma \vdash (\mathit{Key} :: r) : \tau_r}$$

Figure 3.1: The term language and typing rules of our type system HM(RGX), extending the generic basis specified in Figure 2.3

$\mathrm{TCS}_{\mathrm{RGX}}$ in Figure 3.1. Besides the standard constraints for conjunction and variable binding we include two regex-related predicates. Before we can define the semantics of those two regex-related predicates, we introduce a few auxiliary functions and predicates that we are going to use in further definitions in Definition 1. We need additional predicates and functions so that we can abstract over the fact that regexes may have different syntactical representations for the same semantical meaning. This is relevant when using regex intersection types as defined in the term language in Figure 3.1 in definitions.

**Definition 1.** *We assume the availability of the following predicates and functions for*

*the use in further definitions of HM(RGX). In general we will use the lambda-like function application notion functionName parameter$_1$ parameter$_2$ :*

1. ***isRegexEqual*** *is a predicate taking two regexes $r_1$ and $r_2$ as parameters, deciding their equality via finite automata*
2. ***intersect*** *is a function taking two regexes $r_1$ and $r_2$ as parameters, returning a regex representing the intersection of the two regexes*
3. ***foldIntersect*** *is a function that takes a non-empty set $R$ of regexes $r$ as its parameter. It then folds the function intersect over the set $R$. This means it takes two arbitrary regex $r_1$ and $r_2$ from the set, removing them from the set. It calculates their intersection $r_{12}$. Then the function takes the next regex $r_3$ from the set and intersects this with $r_{12}$, and so on. This is done until there are no more regexes left in the set and then it returns the resulting regex. In case there is only one regex in the set, this regex is returned immediately. This function may return the empty regex as a result if no intersection is possible.*
4. ***collect*** *is a function that takes a regex type $\tau_r$ as its parameter.*
   - *In case $\tau_r$ is a regex type variable $\alpha$, it returns a tuple $(\{\alpha\}, \emptyset)$.*
   - *In case $\tau_r$ is a regex $r$, it returns a tuple $(\emptyset, \{r\})$.*
   - *If $\tau_r$ is an regex intersection type $\tau_r \cap \tau_r$ it traverses the intersection tree recursively and and returns a tuple consisting of two sets. The first set contains all regex type variables $\alpha$ encountered while traversing the intersection tree, the second set contains all regexes $r$.*
5. ***isIntersectable*** *is a predicate taking a set $R$ of regexes $r$ as its parameter and checks whether foldIntersect $R$ does not equal to the empty regex. If the given set is empty, this predicate is satisfied.*
6. ***contains*** *is a predicate taking two regexes $r_1$ and $r_2$ as parameters and decides whether the alphabet represented by the first regex $r_1$ can express at least the alphabet represented by the second regex $r_2$.*

After having introduced the functions shown in Definition 1, we now define the semantics of the constraints specified in our language definition in Figure 3.1.

**Definition 2.** *The predicate Intersectable is defined by the following rule, making use of predicates and functions defined in Definition 1:*

$$\textit{Pred-Intersectable}\;\frac{\textit{collect}\ \tau_r = (V, R) \qquad \textit{isIntersectable}\ R}{\textit{Intersectable}\ \tau_r}$$

The predicate *Intersectable*, defined in Definition 2, is satisfied if its parameter $\tau_r$ is a primitive regex or a possible intersection of regexes. Intersection between the two regexes is possible in the sense that it does not yield the empty language that accepts no input at all. Note that type variables have no influence on the outcome of this predicate as it can be seen in its definition, it is solely determined by regexes. As long as there are still any type variables involved it cannot be decided yet. Thus it is also a valid term in our term constraint system.

**Definition 3.** *The predicate $\subseteq$ is defined by the following rules, making use of some predicates and functions defined in Definition 1*

$$\textit{Pred-Contains-Left-empty}\ \frac{\textit{collect}\ \tau_{r_1} = (V_1, \emptyset)}{\tau_{r_1} \subseteq \tau_{r_2}}$$

$$\textit{Pred-Contains-Right-empty}\ \frac{\textit{collect}\ \tau_{r_2} = (V_2, \emptyset)}{\tau_{r_1} \subseteq \tau_{r_2}}$$

$$\textit{Pred-Contains}\ \frac{\begin{array}{cc} \textit{collect}\ \tau_{r_1} = (V_1, R_1) & \textit{collect}\ \tau_{r_2} = (V_2, R_2) \\ \textit{isIntersectable}\ R_1 & \textit{isIntersectable}\ R_2 \\ \multicolumn{2}{c}{\textit{contains}\ (\textit{foldIntersect}\ R_1)\ (\textit{foldIntersect}\ R_2)} \end{array}}{\tau_{r_1} \subseteq \tau_{r_2}}$$

The predicate $\tau_r \subseteq \tau_r$ indicates that the regex described by the second parameter accepts everything that the regex described by the first parameter would accept as defined in Definition 3. As long as there are still any type variables involved, it cannot be decided yet thus it is also a valid term in our term constraint system. However there are two exceptions to this rule that can make a term in our term constraint system involving $\subseteq$ trivially valid. *Pred-Contains-Left-empty* states that if the left hand side of this binary predicate involves no regex $r$ we cannot decide the predicate yet. We have no pair of regexes $r$ to compare, thus we leave it as a valid term. *Pred-Contains-Right-empty* is the equivalent of this rule for the right side of the binary predicate. We proceed by defining some axioms about our regex related predicates and types.

**Definition 4.** *The following axioms regarding our regex related predicates and types apply to our term constraint system $TCS_{RGX}$. The axioms* RGX1*,* RGX2 *and* RGX3 *deal with additional properties of regex intersection types as defined in Figure 3.1. They are used to justify certain steps in further definitions. Axioms* RGX4*,* RGX5*,* RGX6 *and* RGX7 *deal with properties of the containment predicate from Definition 3. Note that we are defining axioms here, so they do not define the exact meaning of the operators but instead additional higher-level properties about them that are guaranteed to hold in the context of our term constraint system. Thus the axioms are notated as entailment rules in the context of our term system, written as $\vdash^e$:*

$$\textit{RGX1}\ \frac{}{\vdash^e \tau_{r_1} \cap \tau_{r_2} = \tau_{r_2} \cap \tau_{r_1}}$$

$$\textit{RGX2}\,\frac{}{\vdash^e \tau_{r_1} \cap (\tau_{r_2} \cap \tau_{r_3}) = (\tau_{r_1} \cap \tau_{r_2}) \cap \tau_{r_3}}$$

$$\textit{RGX3}\,\frac{}{\vdash^e \tau_r \cap \tau_r = \tau_r}$$

$$\textit{RGX4}\,\frac{D \vdash^e \tau_{r_1} \subseteq \tau_{r_2} \qquad D \vdash^e \tau_{r_2} \subseteq \tau_{r_1}}{D \vdash^e \tau_{r_1} = \tau_{r_2}}$$

$$\textit{RGX5}\,\frac{D \vdash^e \tau_{r_1} \subseteq \tau_{r_2} \qquad D \vdash^e \tau_{r_2} \subseteq \tau_{r_3}}{D \vdash^e \tau_{r_1} \subseteq \tau_{r_3}}$$

$$\textit{RGX6}\,\frac{}{\vdash^e \tau_{r_1} \subseteq \texttt{.*}}$$

$$\textit{RGX7}\,\frac{}{\vdash^e \tau_r \subseteq \tau_r}$$

The axioms *RGX1* to *RGX3*, defined in Definition 4, rely on the fact that the intersection of finite automata has the same properties as the operation of intersection on sets. Thus *RGX1* describes commutativity of intersection. *RGX2* describes associativity of intersection, and *RGX3* describes that the intersection of a regex with itself results in the same regex. The remaining axioms deal with properties of the predicate $\subseteq$ as defined in Definition 3. These also arise from the similarity between finite automata and set operations. *RGX4* describes equality in terms of containment. *RGX5* expresses the transitivity of containment. *RGX6* describes that everything can be contained in the regex `.*` . This arises from the fact that everything is trivially contained in the regex `.*` , as it allows an arbitrary sequence of arbitrary characters, and that the predicate only compares regexes as stated in its definition. *RGX7* describes that everything is trivially contained in itself.

As elaborated in Section 2.2.4 we have to define a type equality predicate $=^{\text{RGX}}$, shown in Definition 5. Our predicate does not refer to syntactic equality alone. Instead we define our own equality predicate that supports non-syntatic *regex equality*. Regex equality means that the regular language described by the first regex accepts exactly the same input as the regular language described by the second regex in case regex types get compared. This is expressed in our regex equality predicate $=^{\tau_r}$ by rule *Pred-Eq*$^{\tau_r}$*-r*. In case regex intersection types get compared, it calculates the actual intersections recursively as defined by the rules *Pred-Eq*$^{\tau_r}$-$\cap$-*r*-*Right* and *Pred-Eq*$^{\tau_r}$-$\cap$-*r*-*Left*. It handles cases where intersections do not have regex type variables on one side. *Pred-Eq*$^{\tau_r}$-$\cap$-$\alpha$-*Right* and *Pred-Eq*$^{\tau_r}$-$\cap$-$\alpha$-*Left* handle cases where intersections do not have regexes on one side. *Pred-Eq*$^{\tau_r}$-$\cap$ handles the default case when there are regex and regex type variables on both sides involved. *Pred-Eq*$^{\tau_r}$-$\alpha$ deals with the equality of regex type variables using syntactic equality. Based on $=^{\tau_r}$ we then define $=^{\text{RGX}}$. The rules *Pred-Eq*$^{RGX}$-$\alpha$ and *Pred-Eq*$^{RGX}$-$\tau$ define type equality for non-regex types. This arises from the basic Hindley-Milner instance HM(HERBRANDT), introduced in Section 2.2.4. *Pred-Eq*$^{RGX}$-*RGX* compares the regex types using $=^{\tau_r}$.

The equality of two regular expressions can be decided by using the fact that regular expressions can be converted to a different representation, called *finite automata* (FA).

As Theorem 2.1, Theorem 2.2, Theorem 2.3 and Theorem 2.4 in [HMU01] state this is possible because regexes and finite automata have been proven to be equivalent. They are different representations for the same underlying problem. This theorem is known as *Kleene's theorem*. According to [NR05] well-known algorithms for this task are *Thompson's Construction* or *Gluskov's Construction*, converting a regular expression into a *nondeterministic finite automaton* (NFA). An NFA then gets converted to its deterministic form, a *deterministic finite automaton* (DFA), for instance using the *powerset construction*. It is then possible to decide whether the two resulting DFAs are equivalent as stated in Theorem 3.8 in [HMU01]. There are different ways to test this, for instance by constructing their symmetrical difference and testing whether that result equals to the empty set. In that case they are indeed equal. We can also decide the other two constraints by using finite automata. It is possible to calculate the intersection of the two generated DFAs as elaborated in Theorem 3.3 in [HMU01], stating that the intersection of two regular sets is closed under intersection. A regular set is just another representation of a regex that is also equivalent according to [HMU01]. An operation on regular sets is closed if the result is guaranteed to be a regular set, thus it can always be converted back to a regex. In case an intersection does not equal to the empty regular set our regex intersection predicate is satisfiable. Last, we can check for containment, i.e., the second regex type accepts everything the first regex type would accept and possibly more. We can calculate the intersection between the two regexes and check whether it is equal to the first regex. It is also possible to map the resulting DFA back to a regular expression, for instance using *Kleene's algorithm* presented in [HMU01]. This is useful to normalize the type representations.

**Definition 5.** *We split the definition of the equality predicate for our term constraint system $TCS_{RGX}$ into two parts. We cannot define a regex type equality predicate by using syntactic equality alone. We refer to the predicates and functions in Definition 1 and the axioms defined in Definition 4. We define a regex type equality predicate $=^{\tau_r}$ as the reflexive transitive closure over regex types $\tau_r$ with the following properties:*

$$\textit{emptyOrAny–empty}\;\frac{}{\textit{emptyOrAny}\ \emptyset}$$

$$\textit{emptyOrAny–any}\;\frac{\textit{foldIntersect}\ R =^{\tau_r} \texttt{.*}}{\textit{emptyOrAny}\ R}$$

$$\textit{Pred-Eq}^{\tau_r}\textit{-r}\;\frac{\textit{isRegexEqual}\ r_1\ r_2}{r_1 =^{\tau_r} r_2}$$

$$\textit{Pred-Eq}^{\tau_r}\textit{-}\alpha\;\frac{}{\alpha =^{\tau_r} \alpha}$$

$$\textit{Pred-Eq}^{\tau_r}\textit{-}\cap\textit{-}\alpha\textit{-Right}\;\frac{\textit{collect}(\tau_{r_1} \cap \tau_{r_2}) = (V, R) \quad \textit{emptyOrAny}\ R \quad V = \{\alpha\}}{\tau_{r_1} \cap \tau_{r_2} =^{\tau_r} \alpha}$$

$$\textit{Pred-Eq}^{\tau_r}\textit{-}\cap\textit{-}\alpha\textit{-Left}\;\frac{\textit{collect}(\tau_{r_3} \cap \tau_{r_4}) = (V, R) \quad \textit{emptyOrAny}\ R \quad V = \{\alpha\}}{\alpha =^{\tau_r} \tau_{r_3} \cap \tau_{r_4}}$$

$$\textit{Pred-Eq}^{\tau_r}\textit{-}\cap\textit{-r-Right}\frac{\textit{collect}(\tau_{r_3} \cap \tau_{r_4}) = (\emptyset, R) \quad \textit{foldIntersect } R =^{\tau_r} r}{\tau_{r_1} \cap \tau_{r_2} =^{\tau_r} r}$$

$$\textit{Pred-Eq}^{\tau_r}\textit{-}\cap\textit{-r-Left}\frac{\textit{collect }(\tau_{r_3} \cap \tau_{r_4}) = (\emptyset, R) \quad \textit{foldIntersect } R =^{\tau_r} r}{r =^{\tau_r} \tau_{r_3} \cap \tau_{r_4}}$$

$$\textit{Pred-Eq}^{\tau_r}\textit{-}\cap\frac{\begin{array}{cc}\textit{collect }(\tau_{r_1} \cap \tau_{r_2}) = (V_1, R_1) & \textit{collect }(\tau_{r_3} \cap \tau_{r_4}) = (V_2, R_2) \\ V_1 = V_2 & \textit{foldIntersect } R_1 =^{\tau_r} \textit{foldIntersect } R_2\end{array}}{\tau_{r_1} \cap \tau_{r_2} =^{\tau_r} \tau_{r_3} \cap \tau_{r_4}}$$

*We define the equality predicate $=^{RGX}$ as the reflexive transitive closure over types $\tau$ that makes use of $=^{\tau_r}$ with the following properties:*

$$\textit{Pred-Eq}^{RGX}\textit{-}\alpha\frac{}{\alpha =^{RGX} \alpha}$$

$$\textit{Pred-Eq}^{RGX}\textit{-}\tau\frac{\tau_1 =^{RGX} \tau_3 \quad \tau_2 =^{RGX} \tau_4}{\tau_1 \rightarrow \tau_2 =^{RGX} \tau_3 \rightarrow \tau_4}$$

$$\textit{Pred-Eq}^{RGX}\textit{-RGX}\frac{\tau_{r_1} =^{\tau_r} \tau_{r_2}}{\textit{RGX } \tau_{r_1} =^{RGX} \textit{RGX } \tau_{r_2}}$$

We now define our term constraint system $\mathrm{TCS_{RGX}}$ to be a constraint system like HM(HERBRAND) over the above-mentioned term algebra $T$, where we have the type equality predicate $=^{\mathrm{RGX}}$ from Definition 5 and the two predicates *Intersectable* and $\subseteq$ from definitions 2 and 3. Furthermore the axioms stated in Definition 4 hold.

### 3.2.2 Proofing the Soundness of RGX

As mentioned in Section 2.2.4, we have to show that the conditions imposed by the HM(X) framework hold for our customized term constraint system $\mathrm{TCS_{RGX}}$. This is required so that the general soundness proof given in [Sul00] holds for our instance. We start by showing the required conditions on a term constraint system as given in Definition 7 in [Sul00] hold for $\mathrm{TCS_{RGX}}$.

**Theorem 1.** *The term constraint system $TCS_{RGX}$ for our type system HM(RGX) fulfills the conditions on a term constraint system. According to Definition 7 in [Sul00] the following conditions have to be met:*

*A term constraint system $TCS_\tau$ over the term algebra $T$ is a cylindrical constraint system with predicates of the form $p(\tau_1, ..., \tau_n)(\tau_i \in T)$ such that the following holds:*

*For each pair of types $\tau$, $\tau'$ there is an equality predicate $(\tau = \tau')$ in the term constraint system satisfying:*

$$D1\frac{}{\vdash^e (\alpha = \alpha)}$$

$$D2\frac{}{(\alpha_1 = \alpha_2) \vdash^e (\alpha_2 = \alpha_1)}$$

$$\frac{}{(\alpha_1 = \alpha_2) \wedge (\alpha_2 = \alpha_3) \vdash^e (\alpha_1 = \alpha_3)}\ D3$$

$$\frac{}{(\alpha_1 = \alpha_2) \wedge \exists\alpha.(C \wedge (\alpha = \alpha_2)) \vdash^e C)}\ D4$$

$$\frac{}{(\tau_1 = \tau_2) \vdash^e (T[\tau_1] = T[\tau_2])}\ D5$$

*where $T[]$ is an arbitrary term context*

*For each predicate $p(\tau_1, ..., \tau_n)$,*

$$\frac{}{[\tau/\alpha]p(\tau_1, ..., \tau_n) =^e \exists\alpha.(p(\tau_1, ..., \tau_n) \wedge (\alpha = \tau))}\ D6$$

*where $\alpha \notin fv(\tau)$*

*Proof.* We build upon our definition of our type equality predicate $=^{\text{RGX}}$ in Definition 5. *D1* is true according to our definition, meaning that each type $\tau$ is equal to itself. We can proof this via structural induction on our types $\tau$. Type variables and regex type variables $\alpha$ are equal to themselves according to rules *Pred-Eq$^{RGX}$-$\alpha$* and *Pred-Eq$^{\tau_r}$-$\alpha$*. Function types $\tau_1 \to \tau_2$ are equal to themselves according to rule *Pred-Eq$^{RGX}$-$\tau$*. Regex types *RGX* $\tau_r$ are equal to themselves due to rule *Pred-Eq$^{RGX}$-RGX* that delegates to the regex equality predicate $=^{\tau_r}$. *Pred-Eq$^{\tau_r}$-r* ensures that regexes $r$ are equal to themselves due to our definition of the auxiliary function *isRegexEqual*. The rule *Pred-Eq$^{\tau_r}$-$\cap$* ensures that regex intersection types $\tau_1 \cap \tau_2$ are equal to themselves. The auxiliary function *collect* called on both $\tau_1$ and $\tau_2$ will result in the same pair of sets $(V, R)$ on both sides according to its definition. Thus any regex type variables in the set $V$ are equal. *foldIntersect* will also result in the same regexes $r$ that are then being compared with our regex equality predicate $=^{\tau_r}$. As *foldIntersect* results in a regex $r$, the rule *Pred-Eq$^{\tau_r}$-r* will then apply as we have already shown. Thus we can conclude *D1* holds. *D2* holds as the equality of two finite automata is commutative due to the fact that it is an equivalence relation and thus symmetric. It can also be derived through our axioms *RGX4* and *RGX5* from Definition 4. *D3* holds as finite automata equality is also transitive, another property of an equivalence relation. This can also be derived with the axioms *RGX4* and *RGX5*. *D4* states that type equality can be substituted by type equality in a constraint. It means that if a type $\alpha_1$ is equal to another type $\alpha_2$, then we can satisfy a quantified constraint $\exists\alpha.(C \wedge (\alpha = \alpha_2))$ by instantiating $\alpha$ with $\alpha_1$. Our type equality predicate $=^{\text{RGX}}$ does not depend on any context that could invalidate this rule according to its definition. *D5* describes that type equality is a congruence relation, i.e., it is independent of the context. In case two types $\tau_1$ and $\tau_2$ are equal, they have to be equal in each context. As already mentioned when proofing *D4*, our types do not depend on any kind of context, hence this also holds. Last, *D6* simply connects substitution over predicates using projection and equality. We see that the replacement of type variables into predicates still holds given our two predicates $\subseteq$ and *Intersectable*:

$$[\tau/\alpha](\alpha \subseteq \beta \wedge ... \wedge \alpha \subseteq \gamma) = [\tau/\alpha](\alpha \subseteq \beta) \wedge ... \wedge [\tau/\alpha](\alpha \subseteq \gamma)$$
$$[\tau/\alpha](\mathit{Intersectable}\ \alpha \wedge ... \wedge \mathit{Intersectable}\ \gamma) =$$
$$[\tau/\alpha](\mathit{Intersectable}\ \alpha) \wedge ... \wedge [\tau/\alpha](\mathit{Intersectable}\ \gamma)$$

□

We have now defined the term constraint system $\mathrm{TCS}_{\mathrm{RGX}}$, the first element of the RGX five-tupel and shown the conditions a term constraint system for a valid HM(X) instance imposes. Next we are going to define the second element of the five-tupel RGX, the subsumption relation $\preceq$. Regardless of the symbol, it does not necessarily have to be a subsumption relation. We use the equivalence relation defined by our equality predicate $=^{\mathrm{RGX}}$ as specified in Definition 5 for $\preceq$. Our type system does not specify a subsumption relation.

**Theorem 2.** *Our subsumption relation $\preceq$ for HM(RGX) that is defined in Definition 5 fulfills the conditions regarding a partial ordering plus a contra-variance rule. According to [Sul00] the following conditions have to be met:*

$$\textit{Subsumption-REFL}\frac{}{(\alpha_1 = \alpha_2) \vdash^e (\alpha_1 \preceq \alpha_2) \wedge (\alpha_2 \preceq \alpha_1)}$$

$$\textit{Subsumption-ANTISYM}\frac{}{(\alpha_1 \preceq \alpha_2) \wedge (\alpha_2 \preceq \alpha_1) \vdash^e (\alpha_1 = \alpha_2)}$$

$$\textit{Subsumption-TRANS}\frac{D \vdash^e (\alpha_1 \preceq \alpha_2) \qquad D \vdash^e (\alpha_2 \preceq \alpha_3)}{D \vdash^e (\alpha_1 \preceq \alpha_3)}$$

$$\textit{Subsumption-CONTRA}\frac{D \vdash^e (\alpha_1' \preceq \alpha_1) \qquad D \vdash^e (\alpha_2 \preceq \alpha_2')}{D \vdash^e (\alpha_1 \to \alpha_2 \preceq \alpha_1' \to \alpha_2')}$$

*Proof.* Our type equality predicate $=^{\mathrm{RGX}}$ defined in Definition 5 defines an equivalence relation and we do not use subsumption in our type system. We have already shown that $=^{\mathrm{RGX}}$ is reflective when proofing *D1* of Theorem 1, so *Subsumption-REFL* holds. Transitivity of $=^{\mathrm{RGX}}$ has also been shown when proofing *D3* from Theorem 1, thus *Subsumption-TRANS* holds. *Subsumption-ANTISYM* is fulfilled as $=^{\mathrm{RGX}}$ is symmetric as shown when proofing *D2* from Theorem 1. Antisymmetry $\alpha_1 \preceq \alpha_2 \wedge \alpha_2 \preceq \alpha_1 \to \alpha_1 = \alpha_2$ can be derived from symmetry $\alpha_1 \preceq \alpha_2 \to \alpha_2 \preceq \alpha_1$. *Subsumption-CONTRA* is true for our type system as types get compared via rule *Pred-Eq*$^{RGX}$-$\tau$ of Definition 5 in our equality predicate. □

We have also stated our definition of the fourth element of the five-tupel, the set of solved forms $S$, in Figure 3.1. In solved forms we allow our two regex predicates $\subseteq$ and *Intersectable* to appear. We also have conjunction and projection as we need a way to have both predicates in a single type scheme while still allowing for quantification to support type variables. The empty constraint set $\{\}$ is contained for terms not imposing any further constraints. The equality predicate defined in $C$ only gets used inside the term constraint system to avoid equations in constraints.

We define a set of primitive functions in the initial type environment $\Gamma_0$. We add a function *intersect* that calculates a regex intersection defined as $\forall\alpha_1\forall\alpha_2.\textit{Intersectable}\ (\alpha_1 \cap \alpha_2) \Rightarrow$

$\alpha_1 \to \alpha_2 \to \alpha_1 \cap \alpha_2$ to the initial type environment. Intersection is primarily related to checks. We also need a way to express links. We add another function *link* that checks a link defined as $\forall\alpha_1\forall\alpha_2.\alpha_2 \subseteq \alpha_1 \Rightarrow \alpha_2 \to \alpha_1 \to \alpha_1$. Concrete checks and links are then expressed using lambda abstraction combined with those two primitive functions. Therefore it is usually not necessary to add explicit type signatures stating constraints for a term as they are inferred from those two primitive functions. This completes the definition of the five-tupel RGX parameterizing the generic HM(X) framework.

It remains to show that HM(RGX) is indeed sound. According to Theorem 5 of [Sul00], an instance of HM(X) is sound if the underlying constraint system is sound and the subsumption predicate $\preceq$ is coherent.

**Theorem 3.** *The term constraint system $TCS_{RGX}$ for our type system HM(RGX) is sound as specified in Definition 9 of [Sul00], i.e., every satisfiable constraint has a monotype solution, which is a type $\tau$ where no free variables appear.*

*Proof.* Referring to Definition 9 of [Sul00], it means that for all type variables $\overline{\alpha}$ and constraints $C \in S$, if $\vdash^e \exists\overline{\alpha}.C$ then there are monotypes $\overline{\mu}$ such that $\vdash^e \exists\overline{\alpha}.((\overline{\alpha} = \overline{\mu} \wedge C))$. This is the case for our constraints $\subseteq$, *Intersectable* and our adjusted type equality constraint $=^{\text{RGX}}$. As defined in our term algebra in Figure 3.1 our regex types $\tau_r$ are all monotypes as they are a sort of ordinary types $\tau$ using a special type constructor Rgx. All three of our predicates are satisfiable. Assume an arbitrary regex type $\tau_r$ using its representation in $\tau$ as Rgx $\tau_r$. Now we can always use Rgx $\tau_r$ as monotype substitutions for $\overline{\mu}$ due to the fact that $\vdash^e \exists\alpha_1\exists\alpha_2.(\alpha_1 = \text{Rgx } \tau_r \wedge \alpha_2 = \text{Rgx } \tau_r \wedge \mathit{Intersectable}\ (\alpha_1 \cap \alpha_2))$ always holds in case a constraint is satisfiable. This follows from Axiom *RGX3* in Definition 4. The argument works the same with $\subseteq$ where it follows from Axiom *RGX4*. It also holds with $=^{\text{RGX}}$ where we have already proven reflectivity when proofing *D1* of Theorem 1, so Rgx $\tau_r$ is indeed a satisfiable monotype solution. □

**Theorem 4.** *The subsumption relation $\preceq$ for our type system HM(RGX) is coherent, i.e., if an arbitrary type $\tau$ subsumes a type $\tau'$, $\tau$ is a subset of $\tau'$ as specified in Definition 11 of [Sul00].*

*Proof.* This is true as our subsumption relation is an equality relation as outlined when proofing Theorem 2. □

Summing up we can now conclude that we have defined a sound instance of HM(X):

**Theorem 5.** *Our type system HM(RGX) is a valid sound instance of HM(X) as specified in Theorem 5 of [Sul00], i.e., our term constraint system is sound and coherent.*

*Proof.* This immediately follows from Theorem 1 and Theorem 2, showing that our term constraint system is valid, Theorem 3, showing that our term constraint system is sound, and Theorem 4, showing that our subsumption relation is coherent. □

### 3.2.3 Type Inference for HM(RGX)

We have defined a sound type system.It needs to be shown that this type system satisfies the *lifting property*. In that case the *principal constraint property* also holds. This allows us to use a sound and complete type inference algorithm as outlined and proven in [Sul00]. As stated in Definition 25 [SMZ99] our type system HM(RGX) can make use of this algorithm, if it fulfills the principal constraint property. As outlined by Theorem 10 in [Sul00] this is the case if the lifting property specified in Definition 28 of [Sul00] holds. We are going to recite it in Definition 6:

**Definition 6.** *Given a term constraint system $TCS_\tau$ over the term algebra $T$ and a set $S$ of solved forms, we say that it satisfies the lifting property if the following conditions hold:*

- ***LC1*** *There exists a computable procedure normalize such that for each constraint problem $(D, \phi)$ where $D$ is projection free, normalize$(D, \phi) = (C, \psi)$ and $(C, \psi)$ is the principal normal form of $(D, \phi)$, or normalize$(D, \phi)$ reports failure if $(D, \phi)$ does not have a normal form at all.*
- ***LC2*** *$S$ is equation-free. This is the case if given a constraint system $X$ over a term algebra $T$ and a set $S$ of solved forms, for each $C \in S$ if $C \vdash^e (\tau = \tau')$ then $\vdash^e (\tau = \tau')$.*
- ***LC3*** *Given a constraint problem $(D, \phi)$ and a normal form $(C, \psi)$ then $C = \psi D$.*
- ***LC4*** *For each $\exists \overline{\alpha}.C \in S$ there exists $\overline{\tau}$ and $D \in S$ such that $[\overline{\tau}/\overline{\alpha}]C = D \in S$.*

**Normalization Procedure** We define a *normalization procedure* that reduces arbitrary constraints $D$ to a normal form $C$ such that they cannot be reduced further and are unique up to semantic equivalence with their respective normal form $C$ [SMZ99].

In general regexes are ambiguous, there can be different representations for the same language. For instance `[a-b]` = `a|b` and `a|b` = `[a-b]` ∩ `[a-b]` are semantically equal, but not syntactically. This issue is resolved by using the equality of regexes and finite automata. As stated in Theorem 3.10 and Theorem 3.11 in [HMU01] an arbitrary DFA can be converted to an equivalent unique DFA that has the minimum number of states. A way of doing this is using *Hopcroft's algorithm.* This minimization allows us to compute a unique normal form for all possible constraints involving regex types by using the DFA representation of types throughout the constraint solving implementation. We define a function *normalizeRgx* that normalizes regex types $\tau_r$ as outlined in Definition 7. We can use *normalizeRgx* to convert any regex type $\tau_r$ appearing inside constraints to their normal form. In case *normalizeRgx* returns the empty regex, we emit no normal form as we treat this as a type error.

**Definition 7.** *The function normalizeRgx takes a regex type $\tau_r$ as its parameter and normalizes it as follows, using some auxiliary functions from Definition 1:*

1. *Split the regex type* $\tau_r$ *into type variables and regexes using the function collect* $\tau_r = (V, R)$*. Note that collect eliminates all duplicate type variables* $\alpha$ *and regexes* $r$ *applying set semantics. Note that equality is determined using our regex equality predicate* $=^{\tau_r}$ *to ignore syntactic differences for regexes* $r$*. This step is possible due to commutativity of intersection and the fact that a regex intersected with itself is exactly that regex, so* $\alpha \cap \alpha = \alpha$ *and likewise* $r \cap r = r$*. Furthermore according to our definition of collect we get* $(\{\alpha\}, \emptyset)$ *if* $\tau_r$ *is a regex type variable* $\alpha$ *and* $(\emptyset, \{r\})$ *if* $\tau_r$ *is a regex* $r$*.*

2. *Use foldIntersect* $R$ *to reduce* $R$ *to a single regex* $r$*. This is again possible due to associativity and commutativity. If* $R = \emptyset$*, skip this step.*

3. *Sort the type variables in* $V$ *by name and then fold regex intersection over the sorted set* $V$ *from the left to the right to get a single regex type* $v$*. For instance for three arbitrary type variables* $\alpha_1, \alpha_2, \alpha_3$ *we get the type* $\alpha_1 \cap (\alpha_2 \cap \alpha_3)$*. If* $V = \emptyset$*, skip this step.*

4. *Construct the final normalized type. If both* $R$ *and* $V$ *are not empty sets, intersect the folded regex type* $v$ *and the folded regex* $r$ *to* $v \cap r$*. Using the example for regex type variables* $\alpha$ *from the previous step, we get* $(\alpha_1 \cap (\alpha_2 \cap \alpha_3)) \cap r$*. If* $R = \emptyset$*, use the last type variable* $\alpha_3$ *appearing in the sorted* $V$ *for the right hand side instead of* $r$ *to get* $(\alpha_1 \cap \alpha_2) \cap \alpha_3$*. If* $V = \emptyset$*, return* $r$*.*

We can normalize a single *Intersectable* predicate by applying the function *normalizeRgx* to its argument. We also need a way to normalize multiple *Intersectable* predicates, as a constraint may involve several such predicates appended with our logical conjunction predicate, e.g. *Intersectable* $(\alpha_1 \cap r_1) \wedge$ *Intersectable* $(\alpha_1 \cap (\alpha_2 \cap r_2))$. We do this by combining them to a single *Intersectable* predicate, e.g. *Intersectable* $(\alpha_1 \cap \alpha_2) \cap r_3)$ where $r_3 = r_1 \cap r_2$. Again this is possible due to commutativity and associativity of intersection. If the first predicate of our example is satisfiable and the second predicate is satisfiable, we can intersect and test the predicate on the result of the intersection.

We currently have no way to apply a similar procedure to constraints with $\subseteq$ predicates as it is the case with *Intersectable* predicates, because we do not support regex union types. Assume two predicates $\alpha_1 \subseteq \alpha_2 \wedge \alpha_1 \subseteq (\alpha_3 \cap r_1)$. An equivalent representation would be $\alpha_1 \subseteq (\alpha_2 \cup (\alpha_3 \cap r_1))$ but we do not support a $\cup$ type. The main reason for this is that the predicate $\subseteq$ is not symmetric. Therefore we do not try to unify multiple $\subseteq$ predicates but instead leave them as they are and only normalize their two parameters using *normalizeRgx*. In case concrete regexes $r$ appear on both sides of the predicate, either inside an intersection type or directly, this predicate can already be decided while type variables may still appear. We can already decide a predicate like $(\alpha_1 \cap r_1) \subseteq (\alpha_2 \cap r_2)$ by checking whether $r_1 \cap r_2$ according to the definition of $\subseteq$ in Definition 3. If this is not the case, any intersection of $r_1$ and $r_2$ cannot allow more valid inputs than described by the two regexes $r_1$ and $r_2$. Thus we cannot yield a normal form for this constraint problem as it is unsatisfiable. We can furthermore apply the Axiom

*RGX6* of Definition 4 to eliminate $\alpha \subseteq$ `.*` . Using Axiom *RGX7* we can eliminate $\alpha \subseteq \alpha$ without altering the semantics. In case there are multiple $\subseteq$ predicates in a constraint we can merge them if they represent the same regex types after normalization.

**Theorem 6.** *Our type system HM(RGX) fulfills the lifting property as specified in Definition 6. Therefore it can make use of a generic sound and complete type inference algorithm as outlined in [Sul00].*

*Proof.* The lifting condition implies that for every possible constraint there is some unique, most general representation for it. We first show *LC1* by showing that our normalization procedure yields principal normal forms if possible. A constraint problem $(C, \psi)$ is a principal normal form if for all other normal forms $(C', \psi')$ of an arbitrary constraint problem $(D, \phi)$ it holds that $\psi \subseteq^{\phi'} \psi'$ and $C' \vdash^{e} \phi' C$ [SMZ99]. This property is fulfilled for our normalization procedure. We show this by structural induction on regex types $\tau_r$. As *Intersectable* and $\subseteq$ are independent from each other, we show it for each of them separately.

The base cases for *Intersectable* are that a regex type $\tau_r$ is either a concrete regex $r$ or a regex type variable $\alpha$. If it is a concrete regex $r$ it can be omitted as it is true according to the definition of *Intersectable* in Definition 2. We do not allow $r$ to be the empty regex in normalization procedures as outlined in our normalization procedure. *collect* $r = (\emptyset, \{r\})$ and *isIntersectable* $\{r\}$ can then be decided as *foldIntersect* $\{r\} = r$ will not result in the empty regex either. If $\tau_r$ is a regex type variable $\alpha$ it cannot be reduced further as it can be instantiated with an intersection type so the constraint has to be retained. The induction hypothesis is that we also cannot normalize regex intersection types further. Due to commutativity and associativity we can take a look at a whole intersection chain at once. Take an arbitrary intersection chain $\alpha_1 \cap \alpha_2 \cap r_1 \cap ... \cap \alpha_n \cap r_n$. There are two cases to distinguish. The first case is an intersection between concrete regex types $r$. Our normalization procedure intersects those to a single regex type $r'$ that cannot be reduced further. If this intersection would yield the empty regex, our normalization procedure does not emit a normal form thus this is not possible. The second case is that we have an intersection between the same type variables $\alpha_1 \cap \alpha_1$. This gets eliminated to yield simply $\alpha_1$ according to our normalization procedure that works with sets. Now we have a normalized intersection type where each regex type variable $\alpha$ appears exactly once and regexes $r$ got intersected already to yield a single new regex $r'$. We cannot normalize this further given our term constraint language. We can neither eliminate further regex type variables nor can we intersect regexes anymore.

Next we show that it also holds for $\subseteq$ according its definition in Definition 3. There are five base cases:

1. The containment of $\alpha \subseteq$ `.*` is true according to *RGX6* of Definition 4.
2. The containment of a type variable in itself $\alpha \subseteq \alpha$ is true according to *RGX7* of Definition 4.

3. The containment of two concrete regexes $r_1 \subseteq r_2$ can be decided and gets normalized to either true or no possible normal form.

4. The containment of type variables $\alpha_1 \subseteq \alpha_2$ can not be normalized any further as we cannot decide this predicate without substitutions for them.

5. The containment of a regex type variable in a concrete regex $\alpha \subseteq r$ or $r \subseteq \alpha$ cannot be normalized further using the same argument.

The induction hypothesis is that this also holds for regex intersection types. According to the definition of $\subseteq$ it calculates the tuple *collect* $\tau_r = (V, R)$ for each of the two parameters. If $R \neq \emptyset$ for both parameters, we emit no normal form if the first parameter $r_1$ is not contained in $r_2$. Otherwise we can only normalize the parameters using *normalizeRgx*. In case the normalization of the parameters resulted in a base case, it can be eliminated depending on the normalized result.

Thus we can conclude that *LC1* is indeed fulfilled given the normalization procedure we described in Paragraph 3.2.3. It indeed reduces arbitrary constraints involving our predicates *Intersectable* and $\subseteq$ to a principal normal form up to semantic equivalence.

Next we show *LC2*. As *Intersectable* is no equality predicate, we cannot use it to define equations. It is possible to define an equation using $\subseteq$ due to *RGX4*. An example for such constraint $C$ would be $(\alpha \subseteq \beta) \wedge (\beta \subseteq \alpha)$ as this leads to an equation with the solution $\alpha = \beta$. This still fulfills the condition that if $C \vdash^e \alpha = \beta$ then $\vdash^e \alpha = \beta$ according to our rule. As we do not support the equality predicate in the set of solved forms this case cannot happen otherwise.

The third condition *LC3* implies that for each constraint problem $(D, \phi)$ where a normal form $(C, \psi)$ exists, it must hold that the normal form $C$ is equal to $D$ when $C$'s substitution $\psi$ is applied to $D$. As stated in [SMZ99] equality in this context simply implies a semantic equality in terms of a congruence relation. Therefore *LC3* states that a normal form must have the same semantics as the problem it was derived from. This implies that the substitution $\psi$ has a use in $D$, or in case $D$ has a less general substitution $\phi$ it has no influence on the semantics. The property is not explicitly enforced by the principal normal form, which only requires entailment instead of semantic equivalence with regards to an arbitrary constraint problem $D$. Semantic equivalence is only required for a normal form and its substituted form $C = \psi C$ by the principal normal form. Our normalization procedure does indeed retain the original semantics. Reordering of intersections does not violate the semantics due to commutativity and associativity of $\cap$. The intersection of concrete regexes retains the semantics as they express exactly the same inputs as in their non-intersected form. The intersection of a type variable with itself $\alpha \cap \alpha$ yields $\alpha$ and thus this does not alter the semantics. The merging of *Intersection* constraints retains the semantics due to commutativity and associativity as explained in the normalization procedure. Eliminating $\alpha \subseteq \alpha$ is possible as it is trivially true and does not add any meaning. The containment of a type variable in `.*` can be eliminated as this is also

trivially true. Thus we can conclude that variables appearing in a substitution $\phi$ can only disappear when they have no influence on the semantics, otherwise they are retained.

The last property *LC4* means that for each quantified constraint $C$ in the set of solved forms there is some substitution so that we get a projection-free constraint $D$ in $S$. This is trivially true as we can always find some concrete regexes $\overline{\tau}$ for quantified variables so that we get a constraint consisting only of concrete regex types or intersections of such. To show this satisfiability we argue that we can always use an arbitrary regex $r$ for all type variables. Regex intersections then get reduced to exactly this $r$ which then satisfies the *Intersectable* predicate. $r \subseteq r$ is trivially true according to our definition of $\subseteq$. □

We have shown that all four lifting conditions hold. Therefore we can conclude that our type system HM(RGX) supports a sound and complete type inference algorithm. Now we have fulfilled all proof obligations as imposed by [Sul00]. This completes the formal definition of our type system.

## 3.3 Examples

We now give a small example of how our type system can be used to express configuration specifications in Listing 1. In Listing 2 we demonstrate what kinds of errors it can detect.

```
1  // define two keys
2  let exampleKey1  = Key :: ".*" in
3  let exampleKey2  = Key :: ".*" in
4  // define functions for checks and links
5  let lowerOrDigit = \key. intersect key (Key :: "[a-z0-9]*") in
6  let singleDigit  = \key. intersect key (Key :: "[0-9]") in
7  let fallback     = \key1. \key2. link key1 key2 in
8  let defaultVal   = \key1. \key2. fallback key1 key2 in
9  // express the configuration
10 fallback (singleDigit exampleKey1)
11         (defaultVal (lowerOrDigit exampleKey2) (Key :: "3"))
```

Listing 1: An example term in HM(RGX).
\ stands for $\lambda$, the syntax is according to HM(RGX) defined in Figure 3.1, extending HM(X) in Figure 2.3, *link* and *intersect* are primitive functions in HM(RGX). The only minor difference is that we've enclosed regexes with quotes to distinguish them properly. We assume that let expressions are inside parentheses from the left to the right for readability, i.e., let $x_1 = e_1$ in (let $x_2 = e_2$ in (let $x_3 = e_3$ in (..)))

Assume two arbitrary keys *exampleKey1* and *exampleKey2*, two checks *lowerOrDigit* and *singleDigit*, and a link *fallback*. *lowerOrDigit* checks whether a key consists of digits and lowercase letters in an arbitrary order, expressed by `[a-z0-9]*` . The expression *singleDigit* checks whether its a digit, thus resembles the regex `[0-9]` . The link *fallback* checks whether the regex of another key *key1* a key *key2* can be linked to has to be contained in the regex describing *key1*. Then the first key *exampleKey1* is restricted with

*singleDigit*, the second key *exampleKey2* is restricted with *lowerOrDigit* and uses the link *fallback* to *exampleKey1*. In our type system this can be expressed as follows. Note that there are no explicit type specifications necessary apart from the trivial specifications expressing the regex of keys. A key in Elektra resembles the regex `.*` initially if no further specifications are applied, but our type system does not necessarily enforce this so we could also express keys that are already initially restricted somehow. This is useful for expressing default values as we can consider default values as virtual keys that already have a more specific regex describing them, and then use *defaultVal*, an alias for *fallback* to use the default value if it is compatible with the applied checks.

The term specified in Listing 1 passes the type checking successfully. The types for *exampleKey2* and *exampleKey2* are explicitly specified and do not need further explanation. The expression *lowerOrDigit* yields the polytype $\forall\alpha_1.Intersectable\ (\alpha_1 \cap \texttt{[a-z0-9]*}) \Rightarrow \alpha_1 \rightarrow (\alpha_1 \cap \texttt{[a-z0-9]*})$. This follows immediately from the definition of the primitive function *intersect* along with the lambda abstraction providing $\alpha_1$. The type for *singleDigit* gets inferred analogously to $\forall\alpha_2.Intersectable\ (\alpha_2 \cap [0-9]) \Rightarrow \alpha_2 \rightarrow (\alpha_2 \cap \texttt{[0-9]})$. The type for *fallback* is $\forall\alpha_3\forall\alpha_4.\alpha_3 \subseteq \alpha_4 \Rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \alpha_4$ following from our definition of the primitive function *link* in the initial type environment. The type for *defaultVal* is the same, as it is an alias for *fallback*. The term then gets the type `[a-z0-9]*` . The application of *lowerOrDigit* to *exampleKey2* is possible as the constraint $Intersectable\ (\texttt{.*} \cap \texttt{[a-z0-9]*})$ is satisfied and the result of the intersection is `[a-z0-9]*` . The application of *defaultVal (Key :: 3)* on that term works, as the constraint arising from *defaultVal* would be $\texttt{3} \subseteq \texttt{[a-z0-9]*}$ which is true. Analogously the application of *singleDigit* to *exampleKey1* results in `[0-9]` . Last, *fallback* gets applied to those two applications. Its constraint $\texttt{[0-9]} \subseteq \texttt{[a-z0-9]*}$ is satisfied, thus this function returns the final type `[a-z0-9]*` .

```
1  // define three keys
2  let exampleKey1    = Key :: ".*" in
3  let exampleKey2    = Key :: ".*" in
4  let exampleKey4    = Key :: ".*" in
5  // define functions for checks and links
6  let regex          = \key1. \key2. intersect key1 key2 in
7  let singleDigit    = \key. regex key (Key :: "[0-9]") in
8  let fallback       = \key1. \key2. link key1 key2 in
9  // apply a regex check on exampleKey2
10 let exampleKey2'  = regex exampleKey2 (Key :: "[a-z]") in
11 // apply a singleDigit check on exampleKey2'
12 let exampleKey2'' = singleDigit exampleKey2' in
13 // express the remaining configuration
14 fallback (regex exampleKey3 (Key :: "[a-z0-9]"))
15          (singleDigit exampleKey1)
```

Listing 2: An erroneous example term in HM(RGX), the remarks regarding the syntax in Listing 1 apply as well

Next, we look at two different kinds of errors that our type system detects in Listing

2. We use three keys *exampleKey1*, *exampleKey2* and *exampleKey3* with two checks *singleDigit* from the previous example, and *regex* that serves for arbitrary regex checks. This time we express *singleDigit* not directly but by parameterizing *regex* accordingly. We also use the previous link *fallback*. We then apply two different checks on a single key that are not compatible with each other as there is no input that would be accepted. We use *fallback* to link a key that accepts single digits to another key that accepts single characters, thus they are not compatible with each other either.

The term in Listing 2 contains two different kinds of errors. The first issue arises when type checking *exampleKey2"*. The inferred type of *exampleKey2'* is `[a-z]` . Then *exampleKey2"* fails because the constraint in the type signature of *singleDigit* is $\forall \alpha_2.Intersectable\ (\alpha_2 \cap \texttt{[0-9]})$, but the type of *exampleKey2'* is `[a-z]` . The constraint *Intersectable* $(\texttt{[a-z]} \cap \texttt{[0-9]})$ is unsatisfied according to our definition of the predicate.

Similarly, when applying *fallback* on *exampleKey3* this would result in the constraint $\texttt{[a-z0-9]} \subseteq \texttt{[a-z]}$ . It is unsatisfied because the key it falls back to may contain digits, but the key *exampleKey1* does only accept characters. Closing this chapter we have now shown two examples demonstrating how our type system works. We also demonstrated how type inference works. We presented two erroneous situations HM(RGX) can detect and prevent.

CHAPTER 4

# Implementation

In this chapter we explain our general approach of how we implement the specification defined in Chapter 3. Following the KISS principle we separate our implementation into two parts, both being orchestrated by a plugin for Elektra. We also include an additional plugin that calculates regex representations for a subset of supported metakeys. The general implementation approach is visible in Figure 4.1 and consists of the following parts:

1. *specelektra* [1], a library containing the implementation of our type system specification from Chapter 3 as a typechecker plugin for GHC
2. *spectranslator* [2], a library that reads specifications from the KDB and translates them to specelektra
3. *typechecker* [3], a plugin for Elektra that orchestrates the whole process and interprets specelektra files generated by the spectranslator using the GHC API and provides feedback about type checking errors and inferred types
4. *regexdispatcher* [4], a plugin for Elektra that generates regular expressions for a subset of supported metakeys that cannot be expressed directly, for instance the regex for a range check varies depending on the allowed range

In Figure 4.2 we see an architectural overview of our implementation. Opposed to the general implementation approach it shows which libraries and programming languages are used in the implementation. We use Haskell bindings for *libelektra* in order to work

[1] `https://master.libelektra.org/src/libs/typesystem/specelektra`
[2] `https://master.libelektra.org/src/libs/typesystem/spectranslator`
[3] `https://master.libelektra.org/src/plugins/typechecker`
[4] `https://master.libelektra.org/src/plugins/regexdispatcher`

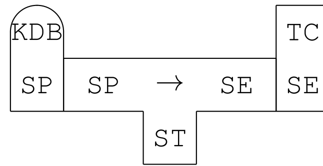



Figure 4.1: A T-Diagram of our Implementation Approach
SP = A Specification in the Elektra KDB in the /spec namespace
ST = spectranslator, translates the specification to specelektra
SE = specelektra, our type system implementation
TC = typechecker, type checks a configuration specification using specelektra

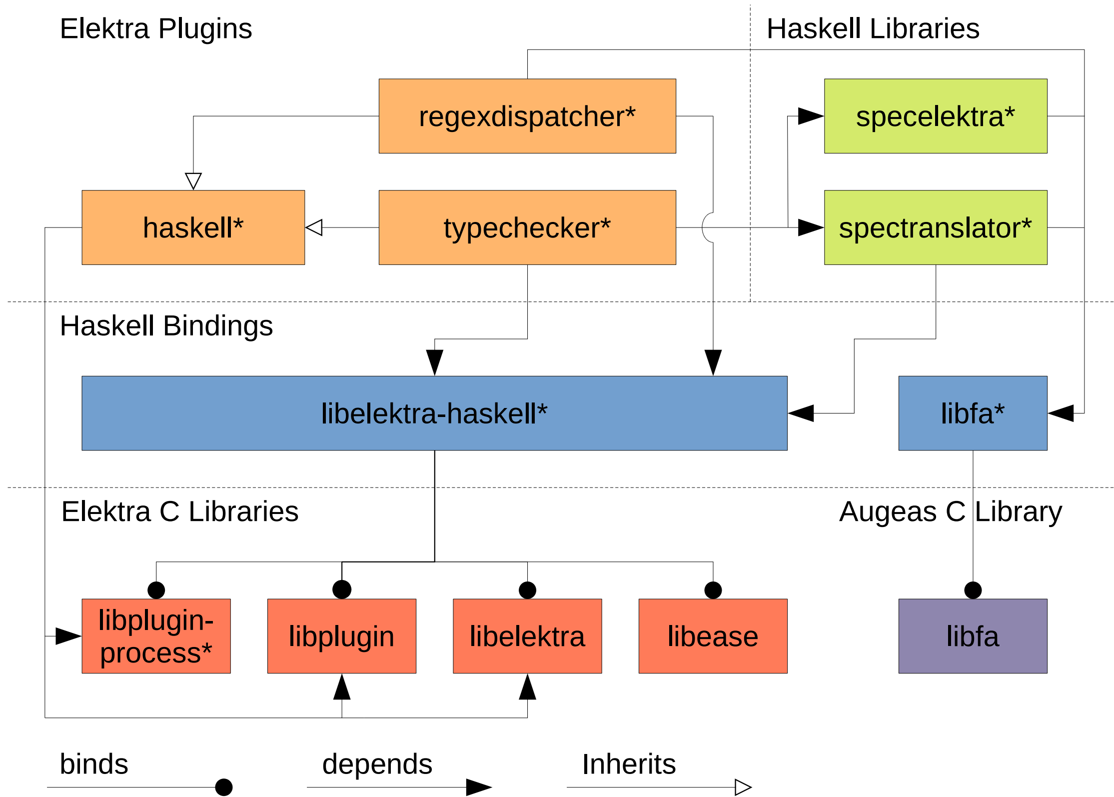



Figure 4.2: An architectural overview of the implementation approach. Components that are implemented in this thesis are marked with an asterisk.

with the library. Our bindings also wrap other libraries provided by the Elektra project, namely *libease*, *libinvoke* and *libplugin*. Haskell is already a very high level language, so we concluded creating separate bindings for the different libraries of Elektra yields little benefit. We assume that on platforms where we can effectively use Haskell, we also have all Elektra libraries available. Additionally we create bindings for *fa*, a C library for working with finite automata. The Haskell libraries specelektra and spectranslator make use of the two bindings in order to realize their functionality. We create the C library *pluginprocess* to execute Haskell plugins in a separate process. Furthermore we use a generic foundation to realize Haskell plugins, a template plugin called *haskell*. The Elektra plugins *typechecker* and *regexdispatcher*, both implemented in Haskell, make use of these building blocks to wrap everything together.

Users of Elektra write configuration specifications in an own namespace in Elektra's KDB, the spec namespace. We use spectranslator for mapping the keys of configuration specifications into our type system HM(RGX). We describe how keys and metakeys are translated to specelektra in Section 4.2. HM(RGX) is implemented in specelektra as a small custom EDSL along with a typechecker plugin for GHC [5] describing the semantics of our domain-specific type system as introduced in Chapter 3. Other information is ignored. spectranslator outputs a Haskell file using our term language that describe the specifications in the spec namespace. We also develop a plugin for Elektra that is called typechecker. This plugin is intended to be mounted along a configuration specification. It then generates a specelektra representation using spectranslator out of the mounted configuration specification, and interpretes it using the GHC API, presenting the results to users of Elektra. We also include a second Elektra plugin that generates regular expressions for dynamic metakeys that depend on some arguments, such as the *check/range* metakey.

As we implement our type system as an EDSL embedded in Haskell, the resulting implementation is more powerful than the type system defined in our specification. In fact we could make full use of Haskell's current type system for any experiments. We state that everything that can be typed in our specification can also be typed in our EDSL. We do not explicitly proof this statement in this work because the implementation is only a proof of concept.

## 4.1 Specelektra

Specelektra is the implementation of our type system specification from Chapter 3. We develop it as an *EDSL (Embedded Domain Specific Language)* in Haskell. We follow the approach suggested by [Gre31] of embedding our type system into an exiting one that is powerful enough to allow us to express our own needs, discussed at the end of Section 2.2.3. Therefore Haskell takes care of the whole type checking and type inference procedure guided by our custom typechecker plugin for the GHC to express the domain-specific semantics of HM(RGX). To work with regexes and finite automata we created Haskell

[5] `https://www.haskell.org/ghc/`

bindings [6] for the C library *libfa* [7]. At the time of writing there was no suitable native Haskell library that supported all the functionality we needed.

### 4.1.1 Haskell Extensions

Haskell is a functional programming language with its roots going back to 1987 [HHPJW07]. Since then there were several revisions of the language that mark the current process of development, called the *Haskell Report*. A notable milestone was Haskell98, which is still a relatively simple language based on a Hindley-Milner style type system. It has a few additions such as ad-hoc polymorphism using type classes. The main force behind Haskell's further development is the compiler GHC. GHC has evolved ever since and lately adds many extensions to the core language defined by the Haskell Report. Most importantly, it introduced features of System F$\omega$, such as type operators in the form of *type families* or the possibility to define custom kinds [WHE13]. Therefore we need to use some of those extensions to model our own type system using Haskell. We follow up with a short description of each necessary extension, taken from the Haskell User Manual [GHC15]. The minimum GHC version required to provide all the necessary Extensions is 8.0.1. This is also the current version of Haskell provided by the latest supported version of *Debian*, called *Debian Stretch*, a popular stable linux distribution with a rather conservative update policy [8].

**TypeFamilies** The extension *TypeFamilies* is one of the most important extensions for our implementation. It extends Haskell with both open- and closed type-level functions and equality constraints for the type checking [SPJCS08]. The difference is that open type families only define a type signature, which can then be implemented by various implementations, loosely resembling Haskell's *type class* mechanism. Closed type families are defined with all members in advance and thus the application of such function will pick the first matching candidate from top to bottom [EVPJW14]. According to [EVPJW14] this allows them to be non-injective, which is useful for instance to define equalities between types at compile time by allowing overlapping expressions. We use closed type families where their semantics are defined by our typechecker plugin.

**DataKinds** *DataKinds* adds kind polymorphism to Haskell, therefore we can use type variables referring to different kinds for type-level functions. This effectively makes types and kinds equal in Haskell's type system. Ultimately it allows the language to support type inference on a kind-level [WHE13]. It promotes algebraic data types to own kinds [YWC$^+$12]. This is required so we can lift the strings describing regexes onto the type level in Haskell and work with them in type families.

[6] https://master.libelektra.org/src/libs/typesystem/libfa
[7] http://augeas.net/libfa/
[8] https://packages.debian.org/stretch/ghc

**ConstraintKinds** This extensions adds a new kind called *Constraint*. It allows us to add type-level functions on custom types to be used in constraints for type signatures. This gives us an easy way to express constraints on the input arguments as specified in Section 3.2.1. [GHC15]

### 4.1.2 GHC Typechecker Plugins

Type checking in Haskell is realized as a constraint-generation and constraint-solving problem. There are several kinds of constraints, the most important being equality constraints that originate from typing rules. For instance, for a function application $f x$ the compiler has to check whether the function type of $f$ corresponds to the domain of the argument $x$. However, GHC only implements the constraint solving for type equality constraints up to type families, and type classes. To allow domain-specific constraint solving, the solver needs to be user-extensible. Support for compiler plugins has been added to GHC in version 7.2.1 to allow custom optimizations of GHC's underlying language System $F_C$. This way the compiler can be extended without having to be too familiar with its internal structure. [Gun15]

**Basic Procedure** An important building block for our regex-based types are so called type-level strings that have the kind *Symbol*. They are used to carry regular expressions describing the contents of a key. Our closed type families *RegexContains*, representing $\subseteq$ of our specification in Section 3.2.1, *Intersectable* and *RegexIntersection*, representing $\cap$ are then used to be processed by our typechecker plugin. This plugin deals with constraint solving and type inference for regex types.

A typechecker plugin gets called by GHC when the compiler's own constraint solver has finished and is left with some constraints that it was not able to solve, so called *wanted constraints*. The plugin can then either solve them directly, simplify them further (possibly generating more constraints), or reject some constraints as unsolvable. In case a constraint has been simplified or further constraints have been generated, GHC tries to use its main constraint solver again to resolve them. Then the plugin may be called again if there is still something left, and so on. There is a configurable limit on the number of times this procedure runs to avoid the computation getting into an endless loop.

The type definition of the constraint solving function that has to be implemented in a plugin is:

```
solveRegex :: RgxData
        -> [Ct] -> [Ct] -> [Ct]
        -> TcPluginM TcPluginResult
solveRegex customData given derived wanted = ...
```

- *RgxData* is an arbitrary data structure that gets initialized before the plugin begins its execution and can be used to initialize additional external dependencies

- *Ct* is the compiler's representation of constraints, according to [Gun15]:
  - *given* constraints are facts that the compiler has already inferred out of the context
  - *derived* constraints may arise from so called functional dependencies, but as we do not use this feature we will not describe them here
  - *wanted* constraints are those that the compiler's built-in solver was not able to handle. It is up to the plugin to treat them
- *TcPluginM* is a monad that abstracts internal functionality such as arbitrary IO, debug messages or the creation of type variables
- *TcPluginResult* is a type formed by two data constructors:
  - *TcPluginOk* takes a list of solved constraints along with their evidence, and a list of new constraints that have been generated
  - *TcPluginContradiction* includes a list of impossible constraints, resulting in a type checking failure

Though the *TcPluginM* monad allows for arbitrary IO, it is important that a typechecker plugin is pure in the sense that it creates the same output for the same inputs. It shall contain correct evidence to keep the type system sound. [Gun15]

**Soundness Evidence** As [Gun15] states it is very easy that plugins can claim that an arbitrary constraint is true without any proof by simply returning *TcPluginOk*. Therefore GHC uses a different approach for typechecking, by lifting the constraints into $SystemF_C$, its minimal core calculus. While this still cannot detect every possible compiler bug it helps detecting faulty behavior.

This is also the reason why *TcPluginOk* does not only deliver the solved constraints but also evidence for each of them. However it is not possible to proof every possible type checking plugin. Thus there exists a special evidence that simply causes the evidence to be true, regardless of any actual implementation. [Gun15] claims that this is a valid approach if a plugin builds upon axioms that have already been proven elsewhere. As we have already proven the soundness of our language in Sections 3.2.2 and 3.2.3 we are going to make use of this.

### 4.1.3 EDSL

The definition of the EDSL representing the term language of HM(RGX) is shown in Listing 3. First we define a new algebraic data type *Key* with a no-argument constructor as we only use this data type for explicit type definitions for regexes. It takes a type-level string containing its regex as a type parameter. We define the two constraints $\subseteq$ and *Intersectable* as closed type-level families. We haven chosen to use closed type families

```haskell
{-# LANGUAGE DataKinds, TypeFamilies, ConstraintKinds #-}

module Elektra.RegexType (RegexContains, RegexIntersection, Intersectable,
↪  Key (..),  Regex, intersect, link) where

import GHC.TypeLits
import GHC.Exts (Constraint)

-- Is the regex a contained in regex b?
type family RegexContains (a :: Symbol) (b :: Symbol) :: Constraint
  where
    -- trivial cases
    RegexContains a (".*") = a ~ a
    RegexContains a a = a ~ a
    -- otherwise interpreted by the typechecker plugin

link :: RegexContains b a => Key b -> Key a -> Key a
link = undefined

-- Is the given regex non empty?
type family Intersectable (a :: Symbol) :: Constraint
where -- interpreted by the typechecker plugin

intersect :: Intersectable (RegexIntersection a b) => Key a -> Key b -> Key
↪  (RegexIntersection a b)
intersect = undefined

-- Calculate the regex that represents the intersection of regexes a and b
type family RegexIntersection (a :: Symbol) (b :: Symbol) :: Symbol
  where
    -- trivial cases
    RegexIntersection a a       = a
    RegexIntersection a (".*")  = a
    RegexIntersection (".*") b  = b
    -- otherwise interpreted by the typechecker plugin

-- A key
data Key (a :: Symbol) = Key deriving Show
type Regex = Key
```

Listing 3: The definition of HM(RGX)'s term language as an EDSL for our proposed type system in Haskell. Trivial cases of our predicates are expressed directly in the EDSL, the remaining semantics are implemented in the GHC typechecker plugin.

because like this we can guarantee that there is no way to alter the behavior of our type families by adding additional instances. Their semantics are solely expressed by our typechecker plugin and their definition. We specify trivial cases we can decide right away directly in the definition of the type families. We use the trivially true constraint $a \sim a$ to express that those cases can be decided as true right away. As mentioned in Section 4.1.2 we rely on type-level strings to carry our regexes in Haskell. We represent regex intersection as a closed type family from two type-level strings to their intersected representation. The semantics of regex intersection are also realized by the typechecker plugin. We provide type definitions for the two primitive functions *link* and *intersect* as defined in the initial type environment of HM(RGX). The EDSL in Listing 3 along with the implementation of the typechecker plugin cannot be changed without invalidating our proof for HM(RGX).

### 4.1.4 GHC Typechecker Plugin

Our GHC typechecker plugin gets invoked by GHC during typechecking. GHC first tries to solve or simplify constraints as much as possible. It makes use of the trivial simplifications defined in the type family definitions of our EDSL as shown in Listing 3. Whenever there are unsolved constraints left after a run of the typechecker, it calls our GHC typechecker plugin. The plugin then solves or simplifies the constraints further. Since our two primitive functions *intersect* and *link* make use of those constraints in their type signature, they have to be solved for any custom functions involving those two primitive functions during typechecking.

```
1  {-# LANGUAGE DataKinds #-}
2  import Elektra.RegexType
3
4  exampleKey1 = Key :: Key ".*"
5  exampleKey2 = Key :: Key ".*"
6  lowerOrDigit key    = intersect key (Key :: Key "[a-z0-9]+")
7  singleDigit  key    = intersect key (Key :: Key "[0-9]")
8  fallback key1 key2 = link key1 key2
9  defaultValue key1 key2 = fallback key1 key2
10
11 spec = fallback (singleDigit exampleKey1) (defaultValue (Key :: Key "3")
   ↪  (lowerOrDigit exampleKey2))
```

Listing 4: The first example from Section 3.3, shown in Listing 1, formalized in our EDSL

In Listing 4 we have expressed the example shown in Listing 1 in our EDSL. We show how constraints are resolved by GHC when using the GHC typechecker plugin. The first notable difference compared to our term language is that we make use of top-level bindings offered by Haskell instead of declaring everything inside let expressions. We need the DataKinds language extension again because it is required to use type-level strings.

The first time our typechecker plugin is invoked with the following wanted constraints for our target expression. This means that GHC is unable to solve these constraints itself:

```
RegexContains "[0-9]" "[a-z0-9]+" (CNonCanonical)
Intersectable "[0-9]" (CNonCanonical)
RegexContains "3" "[a-z0-9]+" (CNonCanonical)
Intersectable "[a-z0-9]+" (CNonCanonical)
```

We then normalize the constraints as elaborated in our normalization procedure in Section 3.2.3. In this case GHC has already applied the intersections as far as it is possible, eliminating the intersections with `.*`. We end up with constraints that we can solve right away without further normalization, as all type variables have already been substituted with concrete regexes. In the next run, the type checker can successfully solve all three normalized constraints. Thus it concludes that this is a valid configuration specification. In case there are still type variables involved after normalization the typechecker cannot decide the constraint and will leave it unsolved. The constraint then gets added to the concerned expression's type signature as an unsolved constraint. This is used for typing functions involving regex types.

```
1  {-# LANGUAGE TypeInType #-}
2  import Elektra.RegexType
3
4  exampleKey1 = Key :: Key ".*"
5  exampleKey2 = Key :: Key ".*"
6  exampleKey3 = Key :: Key ".*"
7  regex key1 key2     = intersect key1 key2
8  singleDigit key     = regex key (Key :: Key "[0-9]")
9  fallback key1 key2 = link key1 key2
10 exampleKey2'  = regex exampleKey2 (Key :: Key "[a-z]")
11 exampleKey2'' = singleDigit exampleKey2'
12 spec = fallback (regex exampleKey3 (Key :: Key "[a-z0-9]")) (singleDigit
   ↪  exampleKey1)
```

Listing 5: The second example from Section 3.3, shown in Listing 2, formalized in our EDSL

We take a look at the second example from Section 3.3 to demonstrate how our implementation handles invalid configuration specifications, shown in Listing 5. As elaborated there are 2 errors in that configuration specification. The error messages show which constraints cannot be solved. The constraints appearing in error messages use the type family definitions from Listing 3. The first error message is:

```
Could not deduce: Intersectable
(RegexIntersection "[a-z]" "[0-9]")
arising from a use of ‘singleDigit’
In the expression: singleDigit exampleKey2'
```

It means that it is not possible to add the *singleDigit* check to the key *exampleKey2'*. The regexes are not intersectable. A key cannot be a character and a digit at the same time. The error messages are Haskell-focused and not postprocessed, thus they refer to keywords of our EDSL. The expression that is stated in the error message refers to the example shown in Listing 4. In general expressions appearing in error messages refer to a configuration specification written in our EDSL.

The second error message means that the fallback from *exampleKey1* to *exampleKey3* is not possible. Their regexes are not compatible with each other. *exampleKey1* only allows for single digits, while *exampleKey3* may also be a character.

```
Could not deduce: RegexContains "[a-z0-9]" [0-9]"
arising from a use of ‘fallback’
In the expression: fallback (regex exampleKey3
    (Key :: Key "[a-z0-9]")) (singleDigit exampleKey1)
```

## 4.2 Spectranslator

Spectranslator is a Haskell library that is used to read configuration specifications from the KDB, transform them into an intermediate representation and then translating them into our EDSL.

### 4.2.1 Libelektra Haskell Bindings and Plugins

To read configuration specifications from the KDB using Haskell we first had to implement bindings for Elektra. The core libraries of Elektra are implemented using the C programming language. Haskell bindings for C libraries are implemented using Haskell's *Foreign Function Interface (FFI)*. However, the FFI is quite verbose to use as developers have to take care of the marshaling between Haskell's and C's data types manually in bindings.

c2hs acts as a preprocessor for such bindings and allows developers to define them using *Hooks*. It then generates bindings that are using the FFI, but one does not have to take care of marshaling data types anymore. Hooks are used to specify type signatures and marshaling procedures for bindings that are then generated by c2hs. The tool analyzes the C header files and then replaces Hooks with plain FFI code in the resulting file. C2hs automatically takes care of the marshaling between the most common data types of the two languages so this does not have to be done manually. The bindings are not specific to spectranslator and can be used to develop arbitrary Haskell plugins and applications accessing Elektra. [Cha00]

Our Haskell binding is low-level, thus its usage is very close to the wrapped C library. Side effects of the underlying C library that works using mutable pointers are encapsulated in Haskell's *IO-Monad*. The binding does not make use of any advanced Haskell features.

Haskell offers many functions to work with monads in general, easing development with the binding nevertheless.

### 4.2.2 Parsing

The first functionality that spectranslator offers is reading and parsing a configuration specification into an intermediate representation. This intermediate representation contains all relevant information for the translation process. We distinguish between two different kinds of keys in a configuration specification. The first kind, called *KeySpecification* represents ordinary keys of configuration specifications. This is where check, link or transformation metakeys are specified.

The library reads all keys of a configuration specification, except those residing in a specific part of a configuration specification prefixed with */elektra/spec*. The keys in such parts are called *FunctionSpecification*. They contain meta-information about how different metakeys have to be translated into the EDSL and what effect they have in terms of the type system. This part is relative to the configuration specification's mountpoint. For instance, if a configuration specification gets mounted to *spec/example*, then the library interprets keys in *spec/example/elektra/spec* as *FunctionSpecification* keys. In case the typechecker is mounted as a global plugin, it interprets keys in *spec/elektra/spec* as *FunctionSpecification* keys. This library does not necessarily require configuration specifications to be mounted in the spec namespace, it simply works relative to a mountpoint. However configuration specifications should be mounted in the spec namespace in general so libelektra applies them as expected.

The following metakeys guide the parsing- and translation process:

**spec/type** specifies the type signature of a specification metakey. The syntax of type signatures is inspired from Haskell's type signatures. The first part specifies constraints on parameters, and the second part describes parameters and the resulting type, separated by ⇒. Parameters are separated using →.

To show this, we express the type signatures for the metakeys *fallback/#* and *check/validation* without the primitive functions *link* and *intersect*. The type signature for *fallback/#* is *RegexContains key2* :: **.** *key1* ⇒ *Key key1* → *Key key2* → *Key key1*. It denotes that this specification metakey resembles a function taking two keys as its parameters. The first key refers to another key, as specified by the separator :: **.** . The value after :: is the metakey on a *KeySpecification* that contains the path of the other key. In case it is **.** , it refers to the current specification metakey, i.e., the value of the metakey *fallback/#* that contains a path to a key. The second parameter *key2* has no path separator. This means it implicitly refers to the regex of the Key that is passed. It results in a Key described by the regex *key1*, if the constraints hold.

The metakey *check/validation* has the type signature *Intersectable* (*RegexIntersection key rgx*) ⇒ *Key key* → *Regex rgx* → *Key* (*RegexIntersection key rgx*). It denotes a

function taking a key and a regex as its parameters, and resulting in the intersection of the regex describing the key with the given regex, in case they can be intersected without resulting in the empty regex.

Using these basic building blocks more complicated type signatures for specification metakeys can be built by users if required. However most of the time specification metakeys can be described using the primitive functions *link* and *intersect*. In that case it is not necessary to explicitly specify a type signature, as it gets inferred from the context.

**spec/impl** specifies the implementation of a specification metakey. The implementation can be described either by using the primitive functions *intersect* and *link*, directly returning one of the parameters, or by specifying *undefined*. The syntax of implementations is that first the parameters have to be specified. Parameter names need to be valid Haskell identifiers. Note that no name for a function has to be specified, it gets inferred by spectranslator by pruning the specification metakey's name to a valid Haskell identifier. The implementation then can make use of the primitive functions *link* and *intersect* if the effect of the specification metakey can be described by them. In that case it is not necessary to explicitly specify a type signature using `/spec/impl`. One can specify *undefined*, but then an explicit type signature has to be given. As an example, the metakey *fallback/#* has the implementation `fallback key = link fallback key`. The metakey *check/validation* has the implementation `regex key = intersect key regex`. A metakey where the regex is known can be expressed directly, for instance, *check/singleDigit* would have the implementation `key = intersect key (Key :: Regex "[0-9]")`.

**spec/order** is used to specify the order in which metakeys are applied. It is supposed to be a number. Higher numbers get applied after lower numbers. By default links have the order 1000, checks have the order 500 and transformations have the order 0. The only restriction is that it fits into a 32-bit integer datatype. In case several keys share the same order, they are sorted by name.

**spec/rename** is used to rename the functions representing configuration metakeys. Usually their name is simply transformed by removing slashes from their key name. This could interfere with keywords that are reserved in Haskell like *default*. Hence they can be renamed to something else. For instance, the specification metakey *default* would have its translation process described by the key */elektra/spec/default*. To resolve the reserved keyword issue, we can use the *spec/rename* metakey on the key */elektra/spec/default* to rename the resulting function to something else.

### 4.2.3 Translation

The second functionality provided by the spectranslator library is the translation of the intermediate representation into Haskell source code that uses our EDSL described in

Section 4.1.3. For this purpose it utilizes the library *haskell-src-exts*, containing the official representation of Haskell's AST. During the translation it first generates functions that represent metakeys. It then generates a variable for each key of a configuration specification labeled with the keyname. These variables are the application of the functions representing metakeys in the given order on the unrestricted key, represented via the regex `.*`. The generated Haskell source code can then be type checked by GHC. In case any constraints of the functions are unsatisfiable during type checking, it gets reported as a type checking problem as seen in the second example in Section 4.1.4.

### 4.2.4 Example

To demonstrate the parsing- and translation process we show a configuration specification that is semantically equivalent to the example given in Listing 4. The configuration specification shown in Listing 6 is written in the INI format and gets translated by spectranslator to Haskell source code using our EDSL.

```
1  #@META spec/order = 1000
2  #@META spec/impl = fallbackKey key = link fallbackKey key
3  [/elektra/spec/fallback/#]
4
5  #@META spec/order = 1000
6  #@META spec/impl = overrideKey key = link overrideKey key
7  [/elektra/spec/override/#]
8
9  #@META spec/order = 1000
10 #@META spec/rename = defaultvalue
11 #@META spec/impl = value key = link value key
12 [/elektra/spec/default]
13
14 #@META spec/order = 500
15 #@META spec/impl = key = intersect key (Key :: Regex "[a-z0-9]+")
16 [/elektra/spec/check/lowerordigit]
17
18 #@META spec/order = 500
19 #@META spec/impl = key = intersect key (Key :: Regex "[0-9]")
20 [/elektra/spec/check/singledigit]
21
22 #@META check/singledigit =
23 [/examplekey1]
24
25 #@META default = 3
26 #@META check/lowerordigit =
27 #@META fallback/#1 = /examplekey1
28 [/examplekey2]
```

Listing 6: A semantical equivalent of the configuration specification shown in Listing 4.

Note that the type specifications are usually not contained in a configuration specification.

Instead they are loaded from a separate file called *prelude.ini* [9] containing default type definitions for commonly used metakeys. This gets translated to the Haskell source code shown in Listing 7 that uses our EDSL. The generated code has already been described in Section 4.1.4 and it is easy to see the semantic equivalence. The main difference is that we have assigned the two described keys to their own variable.

```
1  {-# LANGUAGE DataKinds, NoImplicitPrelude #-}
2  module TestSpecification where
3  import Elektra.RegexType
4  import GHC.TypeLits
5  checklowerordigit key  = intersect key (Key :: Regex "[a-z0-9]+")
6  checksingledigit key   = intersect key (Key :: Regex "[0-9]")
7  defaultvalue value key = link value key
8  fallback fallbackKey key = link fallbackKey key
9  override overrideKey key = link overrideKey key
10 examplekey1 = checksingledigit (Key :: Regex ".*")
11 examplekey2 = defaultvalue (Key :: Regex "3") (fallback examplekey1
   ↪  (checklowerordigit (Key :: Regex ".*")))
```

Listing 7: The configuration specification in Listing 6 after being translated to Haskell source code by spectranslator

## 4.3 Elektra Typechecker Plugin

Typechecker is a plugin for Elektra orchestrating the type checking of a configuration specification by combining the libraries specelektra and spectranslator with the GHC API that provides typechecking. It is intended to be mounted (see Section 2.1.1 for more details about mounting plugins) along with a configuration specification. It also supports loading a file that contains default definitions of translation metakeys for commonly used metakeys for spectranslator as outlined in Section 4.2, referred to as *prelude*. The plugin makes use of our Haskell libraries specelektra and spectranslator for the translation. It uses the GHC API to type check configuration specifications. It is written in Haskell itself as it is easier to use Haskell libraries and work with the GHC API from there.

### 4.3.1 Libelektra Haskell Plugins

Using the bindings we can use Elektra in Haskell applications and libraries. But we also need to be able to develop plugins for Elektra using Haskell, which is a different concern. As explained in Section 2.1.2, an Elektra plugin is usually written in C using a specific interface. This interface consists of functions to start and stop a plugin, and to handle key operations. We have written a plugin for Elektra called *haskell* that acts as a blueprint for developing Elektra Plugins in Haskell. The plugin haskell contains both C code and a Haskell library. The C code implements the plugin interface as usual and acts

[9] https://master.libelektra.org/src/plugins/typechecker/typechecker/prelude.ini

as a bridge between Elektra and Haskell. It first initializes the Haskell runtime, then forwards calls from Elektra to Haskell and last stops the Haskell runtime again. The Haskell library of a plugin has to export the same functions as specified by Elektra's plugin interface using the FFI. Inside the Haskell library we use our binding described in Section 4.2.1 to work with Elektra's Keys and KeySets. The C code of the haskell plugin then calls these exported methods after the runtime has been started, closing the gap between Elektra and Haskell.

We also provide macros [10] for *cmake* [11], the build system of Elektra. These macros can be used to create actual plugins in Haskell without having to implement the bridge in C again. A plugin developer does not need to take care of the tedious compilation- and linking procedure of Elektra and Haskell. The macros use the implementation of the C bridge of the blueprint plugin haskell [12] and links the new plugin's Haskell library instead of the one provided by the haskell plugin.

During the implementation of Haskell plugins the issue arose that once the Haskell runtime has been started and stopped in a single process, it cannot be started again. This interferes with Elektra's way of how plugins are handled. A single process using Elektra may start and stop plugins several times throughout its lifetime. Therefore we had to develop another C library for Elektra that is called *pluginprocess* [13]. This library is intended to be used when developing plugins. It provides functions to fork Elektra processes, launch parts of a plugin inside the child process and it provides a simple communication protocol based on KeySets serialized via Elektra itself over pipes for bidirectional communication between forked processes. It is then used for the C bridge in the haskell plugin. We use it to start and stop the Haskell runtime in the child process instead of the parent process. As a child process is closed again after a plugin gets stopped, the next time the main process starts a plugin using this library, it will be executed in a new child process. This solves the restart fallacy. The pluginprocess library is generic and not specific to Haskell plugins. For instance, it is used for Elektra's *python* plugin. The python plugin is used to execute scripts written in the programming language Python that are accessing the KDB using Elektra's bindings for Python. The runtime of Python also can only be initialized once, so pluginprocess helps here as well.

Now we have a way to implement the typechecker Elektra plugin directly in Haskell. We have decided to use this implementation approach because it is much easier to work with the spectranslator and specelektra libraries directly in Haskell than to access them from a C plugin. We make use of GHC to type check configuration specifications translated to our EDSL via spectranslator, guided by our typechecker plugin specelektra. There are already libraries that make it easy to work with GHC inside Haskell. We have created a foundation that developers can use to create further Haskell plugins for Elektra.

[10] `https://master.libelektra.org/cmake/Modules/LibAddHaskellPlugin.cmake`
[11] `https://cmake.org/`
[12] `https://master.libelektra.org/src/plugins/haskell`
[13] `https://master.libelektra.org/src/libs/pluginprocess`

### 4.3.2 Type Checking

The typechecker plugin gets mounted along with a configuration specification into the KDB. Elektra then calls the plugin whenever the KDB accesses the mounted configuration specification. An overview of the type checking process is shown in Figure 4.3.

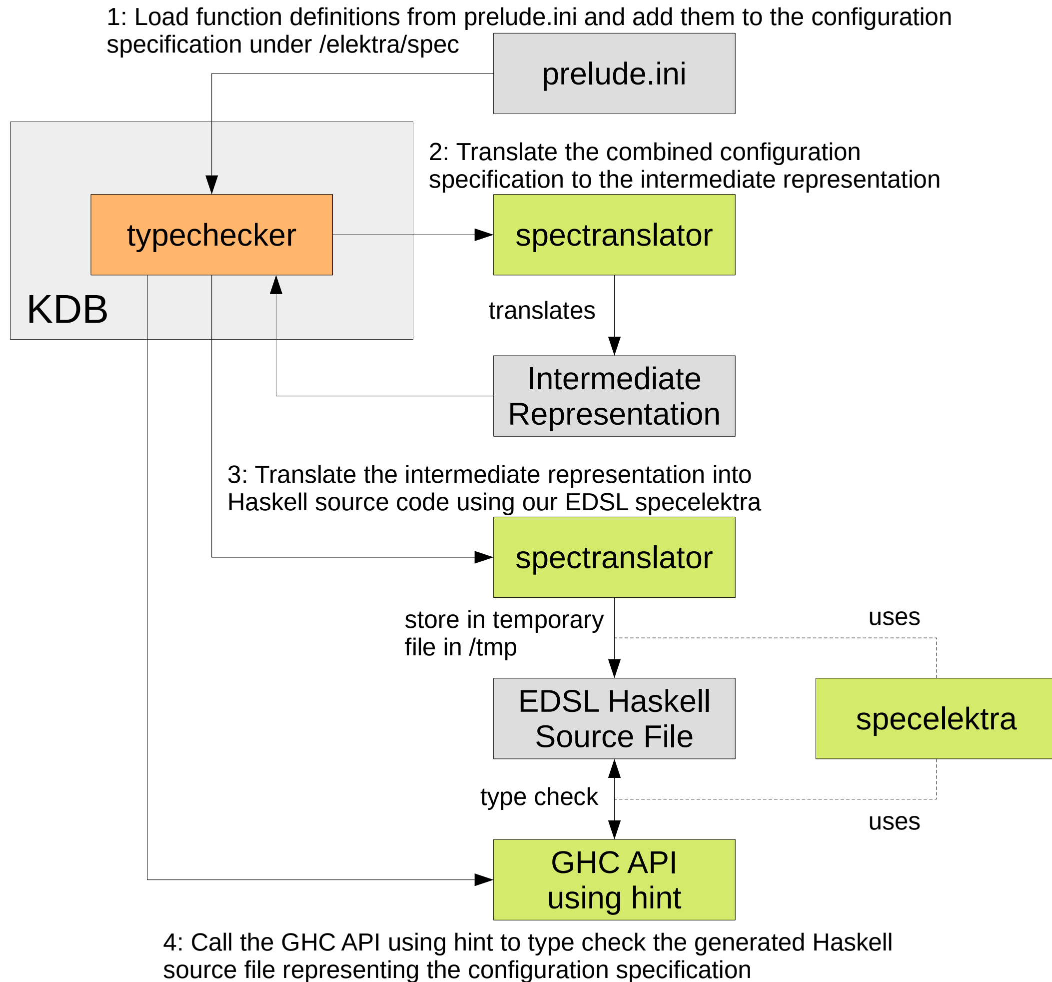



Figure 4.3: An overview of the type checking process

The first step is loading the default definitions from the file *prelude.ini*. The typechecker plugin adds these definitions to the mounted configuration specification. In the second step it uses the spectranslator library to translate the combined configuration specification into an intermediate representation that is easier to process in Haskell. In the third step it uses the spectranslator library to generate a Haskell file containing a representation of the given configruation specification in our Haskell EDSL, described in Section 4.1.3. To

generate a Haskell file we use the library *haskell-src-exts* [14]. It is the official representation of Haskell's abstract syntax tree (AST) and also supports generating a Haskell source code out of the AST. The generated source code gets stored in a temporary file. The fourth step is using the library *hint* [15], a wrapper for the GHC API that makes it easy to use, to type check the generated Haskell source file. In fact hint works in a similar way to Haskell's interpreter GHCi. We instruct hint to load our GHC typechecker plugin contained in the specelektra library.

In case the type checking succeeds, our typechecker plugin returns successfully and Elektra can continue with the remaining plugins of a given backend or store/retrieve the result. If the type checking fails, there are two cases. If a mounted configuration specification is being retrieved via the get function, the failure is being reported as a warning in Elektra, indicating that the mounted configuration specification is not sound according to the rules of the type system. Otherwise if a configuration specification is being modified and there is a type checking failure, it is being reported as an error instead so the modifications will not be persisted to the KDB to avoid unsound configuration specifications. Error messages are displayed as shown in Section 4.1.4. The author of this thesis believes that they indicate the point of failure in a precise enough way that users are able to figure out what is wrong with an unsound configuration specification. We have already shown and described error messages in Section 4.1.4 where we explained why they show the involved keys and issues. Future work on the typechecker could improve the error messages so they point out exactly which metakey causes an issue.

## 4.4 Regexdispatcher

Regexdispatcher is another plugin for Elektra. It preprocesses configuration specifications by generating appropriate regexes to represent metakeys that depend on parameters. Therefore it needs to be mounted along with the typechecker plugin so it can alter a configuration specification with additional unparameterized metakeys before passing it to the typechecker. This separation of concerns is important because we intend to keep the typechecker as modular as possible. It should not need to know how to handle parameterized metakeys. The regexdispatcher is also implemented in Haskell. Developers are free to implement other plugins that serve the same purpose of handling different parameterized metakeys in the language of their choice.

One example of parameterized metakeys is *check/range* that is parameterized with the numerical range that can be assigned to a key's value. Ranges can be expressed as regular expressions, though their representation as a regex is hard to read for humans because they get complex very quickly when representing large ranges. The calculated regex then gets added back to the configuration specification as the parameter of a *check/validation* metakey. As our type system natively handles regexes, it can then successfully interpret this metakey without having to know about the conversion of numeric ranges to regexes.

[14] `https://hackage.haskell.org/package/haskell-src-exts`
[15] `https://hackage.haskell.org/package/hint`

Furthermore it handles the metakey *check/enum*. The plugin does not aim to support every parameterized metakey that is expressible in our type system HM(RGX). Instead it acts as a proof of concept for our separation of concerns. The regexdispatcher plugin uses the *check/validation/message* metakey to store the original specification metakey that got preprocessed. In case there is both *check/range* and *check/enum* present on the same key, regexdispatcher will unify the two regexes by intersecting them. In case this intersection is not possible, it will emit the empty regex. The empty regex will then cause a type checking failure. This information could be used upon a type check error message to map the generated metakey back to the original metakey.

## 4.5 Case Study

To answer the second research question as stated in Section 1.2 we now finish the case study started in Section 3.1. The research question asks how many metakeys of *SpecElektra* that are currently known to be used by plugins, i.e., specified in the *METADATA.ini* file of Elektra with the status of being *implemented*, can have their behavior described by our type system. We analyze the metakeys per category as specified during the categorization in Section 3.1.1. We distinguish between three levels of support:

- **Full (F)** means that a given metakey can have its effects fully described by the type system
- **Partial (P)** means that a given metakey can have its effects partially described by the type system, i.e., the type system will allow more inputs than a plugin implementing the actual effect
- **None (N)** means that the effects of a given metakey cannot be expressed in our type system without further extensions

Furthermore we have already implemented some of the effects of metakeys by adding appropriate type signatures to the prelude.ini file as outlined in sections 4.2 and 4.3.2. In case a metakey cannot have its effects expressed by our type system, we do not categorize it, marked with a dash. Otherwise we distinguish between:

- **prelude.ini (P)** means that a given metakey has its effects described directly in *prelude.ini*, a file containing common type definitions as described in Section 4.3
- **regexdispatcher (R)** means that the plugin regexdispatcher as described in Section 4.4 adds more meta-information to keys describing its effect via a generated regex depending on a metakey's value
- **unimplemented (N)** means that the effect of the metakey has not been implemented yet

In the following tables the header S stands for the support categorization, and the header I stands for the implementation categorization.

**Checks** 21 metakeys, 17 fully supported, 3 partially supported, 1 unsupported, 11 implemented

| Metakey | S | I | Remarks and Regexes |
|---|---|---|---|
| *type* | F | P | by preprocessing the key and delegating to an appropriate regex check using the primitive function *intersect*. |
| *binary* | N | - | binary values are currently unsupported |
| *check/type* | F | P | same as *type* |
| *check/range* | F | R | by generating a regex describing the given range and then adding a *check/validation* metakey |
| *check/math* | P | N | in case the input ranges and the operators are known, a regex can be generated describing the output range so it can be checked if the current key is compatible with that range using the primitive function *intersect* |
| *check/ipaddr* | F | P | as the *ipaddr* plugin already validates keys using regexes, we can use the same regexes in our type system combined with the primitive function *intersect* |
| *check/path* | F | P | as this metakey checks for the existence of a path on the system it can only be decided at runtime. But we can use a regex to verify a path's format |
| *check/validation* | F | P | as this plugin checks whether a key matches a given regex, this regex can be directly lifted on the type system level |
| *check/validation/match* | F | R | by rewriting the regex so that it matches lines, words or everything |
| *check/validation/ignorecase* | F | R | by preprocessing the regex and replacing lower case characters with a group including their uppercase variant, e.g. `[aA]` for `a` |
| *check/validation/invert* | F | R | by inverting the finite automata representing the regex |
| *check/enum/#*<br>*check/enum* | F | R | by generating a regex describing the enum's values, separated via `\|` . Regex metacharacters appearing in enums have to be escaped |

**Check case study continued from previous page**

| Metakey | S | I | Remarks and Regexes |
|---|---|---|---|
| *check/enum/multi* | F | R | by generating a regex describing the different possibilities of a regex. This can be done by listing all enums in an array and generating all possible permutations, then combining them to a single regex |
| *check/calculate* | P | N | similarly to *check/math*, in case the calculation and input domains are known, a regex could be generated representing the allowed range |
| *check/condition* | F | N | by traversing the conditions and generating a regex representing the limitations imposed by the conditions, such as range or equality conditions. Conditions may refer to other keys, in that case these will have no effect on a key itself |
| *check/condition/any/#* | F | N | this metakey supplements *check/condition* and means the check has to match any of the conditions. This can be represented in regexes by forming the union of the finite automata representing conditions |
| *check/condition/all/#* | F | N | this metakey supplements *check/condition* and means the check has to match any of the conditions. This can be represented in regexes by forming the intersection of the finite automata representing conditions |
| *check/condition/none/#* | F | N | this metakey supplements *check/condition* and means the check has to match any of the conditions. This can be represented in regexes by forming the union of the finite automata representing conditions and then inverting the resulting finite automaton |
| *check/date* | P | N | as dates are an irregular languages, for example the number of days in a month depends on the year, they cannot be exactly represented in regexes. However the general syntax of different date formats can be checked up to such irregularities |

**Check case study continued from previous page**

| Metakey | S | I | Remarks and Regexes |
|---|---|---|---|
| *check/date/format* | F | N | this metakey supplements *check/date* and specifies a date format. Date formats describe the syntactic representation of dates. Semantical differences such as different numbers of days in a month depending on the year do not matter for that. Thus date formats can be represented as regexes |

**Transformations** 4 metakeys, 3 fully supported, 0 partially supported, 1 unsupported, 1 implemented

| Metakey | S | I | Remarks and Regexes |
|---|---|---|---|
| *assign/condition/#* *assign/condition* | F | N | by analyzing the conditionals and generating a regex out of the assigned values by escaping regex metacharacters appearing in them, or regexes in case keys are referenced |
| *crypto/encrypt* | N | - | this metakey encrypts a key's data so it is effectively a binary value. Binary values are currently unsupported |
| *unit/base* | F | P | this metakey transforms a key containing hexadecimal values to decimals, thus the effect can be described as a regex representing decimal values. |

**Links** 4 metakeys, 3 fully supported, 1 partially supported, 0 unsupported, 3 implemented

| Metakey | S | I | Remarks and Regexes |
|---|---|---|---|
| *fallback/#* | F | P | with the primitive function *link* |
| *override/#* | F | P | with the primitive function *link* |
| *default* | F | R | with the primtive function *link* after regexdispatcher generated a regex describing the default value by escaping all regex metacharacters in the default value |
| *context* | P | N | with the primitive function *link*, but as there is no general way to derive a regex describing the context it has to be done manually and some contexts can be too complex to describe with regexes |

**Structural Types** 5 metakeys, 0 fully supported, 0 partially supported, 5 unsupported, 0 implemented
As explained in Section 3.1.2 our type system does not support structural types, therefore no metakey of this category is supported.

Now we can answer our second research question:

**Research Question 2.** *How many metakeys of SpecElektra that are currently known to be used by plugins, i.e., specified in the* METADATA.ini *file of the Elektra project with the status of being implemented, can have their behavior described by our type system?*

We conclude that our type system HM(RGX) is currently able to describe the effects of 27 out of 34 metakeys that are listed in the *METADATA.ini* file of Elektra with the status of being *implemented* using regexes. Of those 27 metakeys, 23 can have their effects fully described by our type system while 4 can be partially described. We have implemented 15 of those metakeys for the use in our type system to show the feasibility and practicality of our general concept.

CHAPTER 5

# Related and Future Work

In Section 5.1 we present some other works that can supplement Elektra in useful ways. In Section 5.2 we discuss some ideas how HM(RGX) can be improved.

## 5.1 Related Work

There is a number of works that deal with configuration and their specifications, or ways to detect errors in configurations. In this section we present two papers that can supplement Elektra in useful ways.

### 5.1.1 Dhall

*Dhall* [G+18] is a total programming language specialized for configuration files. It is based on a variant of the lambda calculus to support functions used for abstraction, and includes a few built-in types as well. Furthermore it supports type checking for the types that are supported by the language.

Compilers can be implemented that take Dhall expressions as their input and generate configuration files out of them. These compilers vary depending on the given target domains. There can be also compilers that generate Dhall expressions out of configuration files. Thus Dhall can be seen as an alternative way of representing configuration to Elektra's key-value model. The concept of compilation in Dhall is similar to the way Elektra uses mountpoints to read and write configuration files.

Elektra is more general than Dhall and the concept of key-value pairs is easier to grasp for users that do not have a programming background. The functionality of Elektra can be greatly extended with plugins to not only support different configuration formats (provided by storage plugins in Elektra), but also to add all kinds of checks to a configuration or to handle other concerns such as notifications. Initially we took a

look at Dhall and considered using Dhall as the foundation of our type system instead of HM(RGX). However, Dhall aims to stay a simple language while HM(RGX) tries to model the effects of all kinds of metakeys. Thus it was decided not to use Dhall as the foundation. Adding support for reading Dhall expressions as a storage plugin is a good addition for Elektra so both projects can benefit from each other.

### 5.1.2 ConfigV

*ConfigV* [SZD+17] is a framework for the automated validation of configuration files. As there are hardly any specifications written for configuration files this is a complicated task. Thus ConfigV tries to generate a configuration specification of specific configurations by analyzing examples and inferring constraints from those examples. This approach has the advantage that the learning process is independent of a particular language or format, however enough examples have to be analyzed in order to infer meaningful semantics. ConfigV is able to learn relationships between configuration items in case a correlation seems to exist.

After generating a configuration specification, ConfigV can use this inferred specification to check configuration files. It not only aims to detect errors. It can also detect suboptimal configuration values that may impact application performance by comparing it to best-practices learned from analyzed files.

ConfigV differs from our type system HM(RGX) as we try to describe the effects of metakeys of Elektra while ConfigV tries to automatically infer configuration specifications from examples. It would be very interesting to be able to translate such inferred configuration specifications back to Elektra, describing the semantics via Elektra's metakeys.

## 5.2 Future Work

In this thesis we have developed the formal foundation of a type system for checking configuration specifications in Elektra statically. We developed an experimental implementation of this type system. There are several different areas where this foundation can be improved.

### 5.2.1 Error Messages

Our type checker implementation is realized as an EDSL in the programming language Haskell, along with a typechecker plugin for the Haskell compiler GHC to implement our regex semantics. Thus error messages during type checking are Haskell-focused. People familiar with this programming language should be able to correctly interpret error messages. For ordinary users of Elektra a system can be developed to customize the way Haskell generates error messages for this EDSL. Another idea is to postprocess error messages, for instance with information accumulated by spectranslator, in order to point the user to the cause.

### 5.2.2 Data Types

Currently our system only supports types based on regular expressions. However in order to describe more effects of metakeys with our type system, additional data types could be introduced. Furthermore a way to introduce user-defined types is useful. Possible use cases are the boxing of keys of a certain value inside a custom type, for instance to denote an encrypted key that cannot be assigned to other keys without proper decryption. Another use case could be the generation of structural types for keys. The support of binary data is also desirable.

### 5.2.3 Structural Types

Our type system currently offers no way to describe a structure between keys. Therefore it can be researched in which ways structures can be represented in a type system and to implement such an extension. We have already outlined some ideas in Section 3.1.2. One of the most promising ideas is expressed in [HVP05]. The authors try to model the structure of hierarchical tree-based data using operators commonly associated with regular expressions to express repetition, optional occurrences and alterations.

### 5.2.4 Dependent Types

Several metakeys have effects that depend on the value of keys. While in some use cases these effects can be approximated using regex, for instance for the *check/math* plugin, there is no way to represent such effects in a more precise way. As Haskell already supports some operations on natural numbers on a type-level this could be used to handle mathematical checks or similar use cases in a more precise way.

### 5.2.5 Implementation of Metakeys

We have already implemented the effect of some metakeys in our type system during the development of our prototype. However a great amount of effort went into the creation of the typechecker plugin, the Haskell bindings for Elektra, the Haskell plugin for Elektra and the formal definition of the type system. There are still some metakeys that can have their effects described by our type system but are not implemented yet.

### 5.2.6 Contextual and Circular Links

There is no way to express circular links in HM(RGX). Our prototype does support circular links as the type system of Haskell allows it. Therefore it would be interesting to add support for circular links directly to HM(RGX). Another interesting addition are contextual links, expressed by the *context* keyword. A plugin can be created that generates regexes describing contexts and dispatches the semantics to all involved keys matched by a context as far as it is possible.

CHAPTER 6

# Conclusion

We first introduced some general information about Elektra, the initiative providing the library libelektra. This library provides unified access to configuration via key-value pairs stored in a database in a hierarchical manner. We elaborate how Elektra uses configuration specifications to describe the semantics of configuration. We also argue why a type system for configuration specifications is important to prevent various kinds of mistakes in advance. There are two kinds of mistakes our type system detects. The first mistake is adding incompatible checks to keys, i.e., adding two range checks denoting different ranges. This is typically expressed using the primitive function *intersect* in our type system. The second mistake is linking between incompatible keys, i.e., from a key accepting a certain range to another key representing a different range, even if they partially overlap. This is typically expressed using the primitive function *link* in our type system. It is not intended to replace runtime checks, as users can set arbitrary values on keys that have to be checked each time.

We resumed with an introduction of type systems. We introduce some general information about type systems and argue why using a type system is beneficial. Afterwards we elaborated various variants of the lambda calculus, a notion of computation that is often used as the formal foundation for type systems. We closed our introduction by explaining HM(X), a generic framework for creating domain-specific type systems based on a restricted version of the lambda calculus $\lambda_2$ called a Hindley-Milner type system.

We analyzed the metakeys used in configuration specifications that are currently present in Elektra and implemented by plugins for this library. In this analysis we categorized the metakeys depending on the kind of effect they have on a key. Based on this analysis we researched various type system techniques and evaluated how suitable they are for the given target domain of configuration specifications. We came to the conclusion that a type system based on regular expressions to express the semantics of configuration specifications is a good compromise between ease of use, power and extensibility.

The type system has then been formally defined building upon the HM(X) type system framework. Along the way we fulfilled the necessary proof obligations as imposed by the framework in order to have a sound type system. Last, we also showed a normalization procedure for our type system in order to gain type inference as well. Our type system uses a common and well-researched formal basis as it is based on a Hindley-Milner-style type system. This basis can be used for further improvements of the type systems to incorporate more advanced typing techniques.

A prototype was developed for Elektra implementing the type system as a plugin. For the prototype we first created a Haskell binding for Elektra. Furthermore we created a Haskell binding for libfa, a C library to work with finite automata. Then we created a generic foundation for writing Elektra plugins in Haskell. We also created the pluginprocess library for Elektra to execute plugin logic in a separate process. This is required as the Haskell runtime cannot be restarted after it has been closed inside a process, but Elektra may open and close a plugin several times during runtime. This library now also is used to execute the python plugin in its own process that has the same issue. The generic foundation was used to create the typechecker plugin and the regexdispatcher plugin for Elektra. All these parts constitute the prototype that serves to demonstrate that our type system works as intended and has a practical use. These plugins can be used to validate configuration specifications in Elektra. We then evaluated how many metakeys can have their effects expressed by our type system. We described some similar approaches for typed configuration in related work and closed the thesis by outlining some fields where the type system can be improved to express the effect of additional metakeys.

Our type system HM(RGX) is suitable to describe many metakeys that are listed in the *METADATA.ini* file of Elektra with the status of being *implemented*. 27 out of 34 metakeys can have their effects expressed in our type system to aid in detecting errors in configuration specifications. We conclude that using regular expressions to describe values of configuration items is a good way to cover many cases. Further extensions to HM(RGX) are required to handle structural constraints on configuration items. Non-regular languages such as dates are not possible to express in the type system, however, their exact semantics can often be approximated.

# Bibliography


[Bar91] Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991.

[BFP+08] Aaron Bohannon, J Nathan Foster, Benjamin C Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *ACM SIGPLAN Notices*, volume 43, pages 407–419. ACM, 2008.

[BK93] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[BL80] Janusz A. Brzozowski and Ernst Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, 1980.

[Cha00] Manuel M. T. Chakravarty. C $\longrightarrow$ Haskell, or yet another interfacing tool. In Pieter Koopman and Chris Clack, editors, *Implementation of Functional Languages*, pages 131–148, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, pages 207–212. ACM, 1982.

[EVPJW14] Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *ACM SIGPLAN Notices*, volume 49, pages 671–683. ACM, 2014.

[G+18] Gabriel Gonzalez et al. The dhall configuration language. "`https://github.com/dhall-lang/dhall-lang`", 2018. [Online; accessed 13.08.2018].

[GHC15] GHC Team. GHC users guide. "`https://downloads.haskell.org/~ghc/8.0-latest/docs/html/users_guide/glasgow_exts.html`", 2015. [Online; accessed 08.03.2018].

[Gre31] Oleg Grenrus. Domain specific type systems. Master's thesis, Aalto University, 2016-10-31.

[Gun15] Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *ACM SIGPLAN Notices*, volume 50, pages 11–22. ACM, 2015.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN Conference on history of programming languages*, pages 12–1. ACM, 2007.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, March 2001.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, 2005.

[LP99] Leslie Lamport and Lawrence C Paulson. Should your specification language be typed. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):502–526, 1999.

[Lut08] David Lutterkort. Augeas - a configuration api. In *Linux Symposium, Ottawa, ON*, pages 47–56, 2008.

[MA08] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *European Conference on Object-Oriented Programming*, pages 260–284. Springer, 2008.

[MA09] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. In *European Symposium on Programming*, pages 95–111. Springer, 2009.

[NR05] Gonzalo Navarro and Mathieu Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2005.

[Pie02] B.C. Pierce. *Types and Programming Languages*. Types and Programming Languages. MIT Press, 2002.

[Raa10] Markus Raab. A modular approach to configuration storage. Master's thesis, Technical University of Vienna, 2010.

[Raa15] Markus Raab. Sharing software configuration via specified links and transformation rules. In *Kolloquium Programmiersprachen (KPS 2015)*, 2015.

[Raa16] Markus Raab. Improving system integration using a modular configuration specification language. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 152–157, New York, NY, USA, 2016. ACM.

[RB17] Markus. Raab and Gergö Barany. Introducing context awareness in unmodified, context-unaware software. In *ENASE 2017-12th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 1–8, February 2017.

[SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley-Milner style type systems in constraint form. *Res. Rep. ACRC-99-009, University of South Australia, School of Computer and Information Science*, 1999.

[SPJCS08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *ACM Sigplan Notices*, 43(9):51–62, 2008.

[Sul00] Martin Sulzmann. *A general framework for Hindley-Milner type systems with constraints*. PhD thesis, Yale University, 2000.

[SZD+17] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):64:1–64:20, October 2017.

[Vin97] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[WHE13] Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. System fc with explicit kind equality. In *ACM SIGPLAN Notices*, volume 48, pages 275–286. ACM, 2013.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 214–227. ACM, 1999.

[YWC+12] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on types in language design and implementation*, pages 53–66. ACM, 2012.

# Appendix I

| Metakey | Category | Status |
|---|---|---|
| order | U | implemented |
| comment | U | deprecated |
| comment/# | U | implemented |
| comment/#/start | U | implemented |
| comment/#/space | U | implemented |
| line | U | proposed |
| fallback/# | L | implemented |
| override/# | L | implemented |
| namespace/# | U | implemented |
| default | L | implemented |
| context | L | implemented |
| callback | U | reserved |
| type | C | implemented |
| binary | C | implemented |
| array | S | proposed |
| mountpoint | U | implemented |
| infos | U | implemented |
| config | U | proposed |
| opt | U | implemented |
| opt/long | U | implemented |
| env | C | proposed |
| see/# | U | implemented |
| rationale | U | idea |
| requirement | U | idea |
| description | U | reserved |
| example | U | reserved |
| rename/toupper | S | implemented |
| rename/tolower | S | implemented |
| rename/cut | S | implemented |
| rename/to | S | implemented |
| origname | S | implemented |

**Analyzation continued from previous page**

| Metakey | Category | Status |
|---|---|---|
| conflict/_ | U | proposed |
| array/range | C | proposed |
| require | S | unclear |
| mandatory | S | unclear |
| required | S | unclear |
| error | U | implemented |
| warnings/# | U | implemented |
| struct | S | proposed |
| check/type | C | implemented |
| check/type/min | C | deprecated |
| check/type/max | C | deprecated |
| check/range | C | implemented |
| check/math | C | implemented |
| check/ipaddr | C | implemented |
| check/format | C | idea |
| check/path | C | implemented |
| check/validation | C | implemented |
| check/validation/message | U | implemented |
| check/validation/match | C | implemented |
| check/validation/ignorecase | C | implemented |
| check/validation/invert | C | implemented |
| check/validation/type | U | deprecated |
| check/enum | C | implemented |
| check/enum/# | C | implemented |
| check/enum/multi | C | implemented |
| check/calculate | C | implemented |
| check/condition | C | implemented |
| condition/validsuffix | C | proposed |
| check/condition/any/# | C | implemented |
| check/condition/all/# | C | implemented |
| check/condition/none/# | C | implemented |
| assign/condition | T | implemented |
| assign/condition/# | T | implemented |
| check/date | C | implemented |
| check/date/format | C | implemented |
| trigger/warnings | U | implemented |
| trigger/error | U | implemented |
| trigger/error/nofail | U | proposed |
| deprecated | U | idea |
| internal/<plugin>/* | U | implemented |

**Analyzation continued from previous page**

| Metakey | Category | Status |
|---|---|---|
| source | U | implemented |
| dependency/control | C | idea |
| dependency/value | C | idea |
| application/name | U | idea |
| application/version | U | idea |
| fixed | T | idea |
| restrict/write | C | idea |
| restrict/null | C | idea |
| restrict/binary | C | idea |
| restrict/remove | C | idea |
| evaluate/<language> | T | idea |
| accessibility | U | idea |
| visibility | U | implemented |
| uid | C | obsolete |
| gid | C | obsolete |
| mode | C | obsolete |
| atime | C | obsolete |
| mtime | C | obsolete |
| ctime | C | obsolete |
| spec | U | reserved |
| proc | U | reserved |
| dir | U | reserved |
| user | U | reserved |
| system | U | reserved |
| csv/order | U | implemented |
| crypto/encrypt | T | implemented |
| crypto/salt | U | implemented |
| xerces/rootname | U | implemented |
| unit/base | T | implemented |

Table 1: A categorization of the metakeys currently present in the *METADATA.ini* file of the Elektra project.
U ... Unrelated to types
T ... Transformation between keys
S ... Structural constraints and key path transformations
C ... Checks of checker plugins
L ... Link between keys
O ... Other type-related metakey