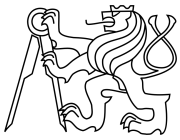
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



# ASSIGNMENT OF MASTER'S THESIS

| | |
|---|---|
| **Title:** | Advancement of the Manta Checker DFP System |
| **Student:** | Bc. Adam Kučera |
| **Supervisor:** | Ing. Robert Pergl, Ph.D. |
| **Study Programme:** | Informatics |
| **Study Branch:** | Web and Software Engineering |
| **Department:** | Department of Software Engineering |
| **Validity:** | Until the end of winter semester 2017/18 |

## Instructions

1. Describe briefly the Manta Checker system and its Data Flow Programming module (MCH-DFP).
2. Review the leading type systems used in contemporary programming languages.
3. Design a simple consistent type system for MCH-DFP.
4. Implement the type system and demonstrate it on examples.
5. Analyse the requirements on tracing DFP.
6. Implement a user-friendly system for viewing debugging logs.
7. Summarize your results and their benefits for the Manta Checker product.

## References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 26, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

# Advancement of the Manta Checker DFP System

***Bc. et Bc. Adam Kučera***

Supervisor: Ing. Robert Pergl, Ph. D.

29th April 2017

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 29th April 2017 .....

Czech Technical University in Prague
Faculty of Information Technology


### Citation of this thesis

Kučera, Adam. *Advancement of the Manta Checker DFP System.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

# Abstrakt

Manta Checker je nástroj pro automatickou kontrolu kódu ve velkých datových skladech. Modul datových toků této aplikace využívá paradigmatu programování pomocí datových toků (DFP), díky němuž mohou aplikaci využívat i uživatelé s menšími technickými znalostmi. Povaha datových toků uvnitř modulu umožňuje jednoduchou vizualizaci jednotlivých pravidel, podle nichž se dělá kontrola kódu, a také jejich vylepšování. Tato práce dále rozšiřuje zmíněný modul datových toků o vylepšený typový systém, dále implementuje logování do interpretu datových toků a umožňuje zobrazení těchto logů přímo na grafu těchto toků, což je nový přístup v debugování DFP aplikací.

**Klíčová slova** programování pomocí datových toků, vizuální programování, typové systémy, logování, trasování, debugování

# Abstract

Manta Checker is an automated code review tool for large data warehouses. Its dataflow module is leveraging the dataflow programming paradigm (DFP), thus even a users with lesser technical skills can use the application. The dataflow nature of the module allows simple visualization of code review rules and their straightforward improvement. This thesis further extends the

dataflow module with an enhanced type system, it implements means of logging into the DFP interpret and visualizes these logs directly on the DFP graph, which is a novel approach in debugging DFP applications.

**Keywords** dataflow programming, visual programming, type systems, logging, tracing, debugging

# Contents

# List of Figures

# List of Tables

# Introduction

Manta Checker (MCH) is an automated code review tool, which is mainly applied to large data repositories and warehouses and the database technologies they use. It can automatically locate problematic sequences in the database queries. [1] It uses user specified rules to find these issues, however these rules are defined in a technically complicated way and therefore a user of the application requires to have specific technical knowledge. To tackle this problem, MCH application was extended with a dataflow programming module, which enables designing these rules in a more intuitive way and requires lesser technical skills. [2]

But why is the dataflow programming approach a good fit for MCH-DFP application, which needs that even user with not that high technical knowledge can control it? Data flow programming is an alternative to classic control flow programming and one of its main advantages is that it can be easily visualized. Such visually constructed programs have lesser demands on perception to understand them and they are more comprehensible in relatively simple and pedagogical uses. [3]

Moreover, the human visual information processing is optimized for multi-dimensional data. A classical presentation of computer programs is an one-dimensional textual form and it is not utilizing the full power of human processing capabilities. However programs, such as flowcharts, are known to be helpful tools in application understanding. In addition, a visually represented program allows to present the user higher-level description of the actions the program undertakes, which can be especially useful when tracing the execution of program and visualizing the data it processes. [4]

The goal of this thesis is to advance the aforementioned MCH-DFP system with two new modules. Firstly, the existing type system in the application should be reviewed, its flaws should be identified and based on these flaws, the state-of-the-art research and existing type systems a new type system for MCH-DFP should be designed and implemented. Secondly, a logging and tracing module for the application should be designed and implemented, which

would allow the user to examine the flow of the data in the application and debug it. It is important that both of these new modules follow the same basic requirements for MCH-DFP, so that it can be easily used even by technical managers with lesser technical skills.

The thesis is structured as follows: In the theoretical part, the existing MCH-DFP system is described and also its underlying theory is presented. Then theoretical concepts about type systems in programming languages are reviewed and the examples of formal and informal type systems are presented. Afterwards the state-of-the-art research about logging and tracing DFP applications is presented. In the practical part, a type system module for MCH-DFP is designed, implemented and demonstrated on real world industrial example. Then requirements for logging and tracing system are set and a novel approach to tracing and debugging DFP applications is designed and implemented.

# Part I

# Theoretical part

The following theoretical part focuses on giving the reader a basic understanding of MCH-DFP system and to review the state-of-the-art research about two new modules that need to be implemented to MCH-DFP: type system and tracing and logging system.

Firstly, Manta Checker product will be presented with focus on its dataflow programming module. The application architecture will be described, including theoretical concepts behind the application. Then the terms *type* and *type systems* will be defined, several type system classifications will be presented and both formal and informal type system will be reviewed. Lastly, the theory about tracing and logging systems will be reviewed and related work will be presented.

Chapter 1

# Manta Checker and its Data Flow Programming module

In this section, the Data Flow Programming (DFP) module of Manta Checker (MCH) will be described. MCH is quite extensive application and DFP module is just a small part of it. The goal of this thesis is to implement and improve just a few parts of MCH-DFP, but it is essential to understand the application as a whole to comprehend the need for these improvements. In order to give the reader proper understanding, this section will focus on key parts of MCH and its DFP module relevant to the goals of this thesis.

## 1.1 Manta Checker

### 1.1.1 Basic information

Manta Checker is an automated code review tool focused mostly on large data repositories and warehouses. It enables its users to analyse database queries in their repositories and automatically locate errors and other issues. The application then offers the user thorough reports, which can be analysed and from which necessary steps to assure the proper quality can be taken.

The aim of the product is to reduce the costs of human resources maintaining the data repositories, increase the efficiency of such stores and also its security. [1]

The application is created by Czech company Manta Tools s.r.o. and parts of it were developed with financial support from Technology Agency of the Czech Republic (TACR). Researchers from Faculty of Information Technology at Czech Technical University in Prague (FIT CTU) were also cooperating on some parts of the application. The first version of product was published in 2013.

Right now, the application supports three major data governance solution providers: Teradata, Informatica and Oracle. [5]

### 1.1.2 Application functionality

The product contains large number of preset **rules**, against which a database code can be checked. These rules should enforce the absence of common errors, but also the fulfilment of different company policies and industrial best practices. An example of such a rule could be a checker, which ensures that every variable in a code has a proper length, so its name is clear and self explanatory. Sets of these rules are composed into **scenarios**.

MCH application allows a user to perform code reviews either directly on a data repository or on an uploaded file with a database code. The user can choose which scenario to run on the given code. Users can also specify their own rules and compose their own scenarios.

After running a scenario on a code, the user receives a detailed report about problems found in the code. Some of these issues can also be automatically repaired by the tool.

Typical use case of MCH product is on large data warehouses, which usually contain lots of code from huge number of different suppliers. Enforcing company policies and proper code quality in such environment is very problematic and costly and MCH can significantly simplify this process by automating most parts of it. [5]

## 1.2 Manta Checker Data Flow Programming module

In the previous subsection, it was described that Manta Checker product works with a set of rules, which check for common errors and other business requirements in the data repositories. A common way to design these rules is e.g. by complex XPath queries and regular expressions. This way can however be too complex for some users, as business requirements are often set by technical managers rather than engineers with proper technical knowledge. [6]

DFP module tackles this issue, as it offers more intuitive way of designing the rules. The module has a graphic interface, where the rule is visualised as a dataflow between separate functions and the user can directly edit this flow.

### 1.2.1 Underlying theory behind the DFP module

In this section, the underlying theory behind the DFP module will be briefly described, then the architecture of the module will be outlined and finally the user interface of DFP module will be characterized.

#### 1.2.1.1 DFP paradigma

Dataflow programming is a paradigm, in which the program is modelled as an oriented graph. [7]

**1.2.1.1.1 History** The paradigm firstly appeared in 1970s as an alternative to classic von Neumann architecture, which seemed to be unsuitable for massive parallelism. Programs with that kind of architecture was known to be directed with a global program counter and were accessing data in a global memory. On the other hand, the proposed dataflow architecture leveraged usage of local memory and executing program instructions as soon as their operands become available.

The paradigm got its name from the flow of data between instructions on the instruction oriented graph. At first, it was researched directly as an hardware alternative for von Neumann architecture, however later it also evolved into a paradigm for programming languages. [8]

**1.2.1.1.2 Dataflow execution model** Every dataflow program can be represented as an **oriented graph** (also called a **network**). **Nodes** of this graph are program instructions (also called **blocks**) and **edges** represent the dataflows between these instructions (ie. the input parameters of instructions and their return values). Every block can have **input and output ports** in which the data edges are connected.

Depending on number of ports, there are three types of blocks:

- **Source** - generates data to the network, has no input ports.
- **Sink** - dataflow ends in this block, has no output ports.
- **Processing block** - transforms the inputs to outputs, has both input and output ports.

Every block can also be a **composite block** (also called **component**). Such a block contains another subgraph of instructions and therefore a program can be decomposed into many composite parts.

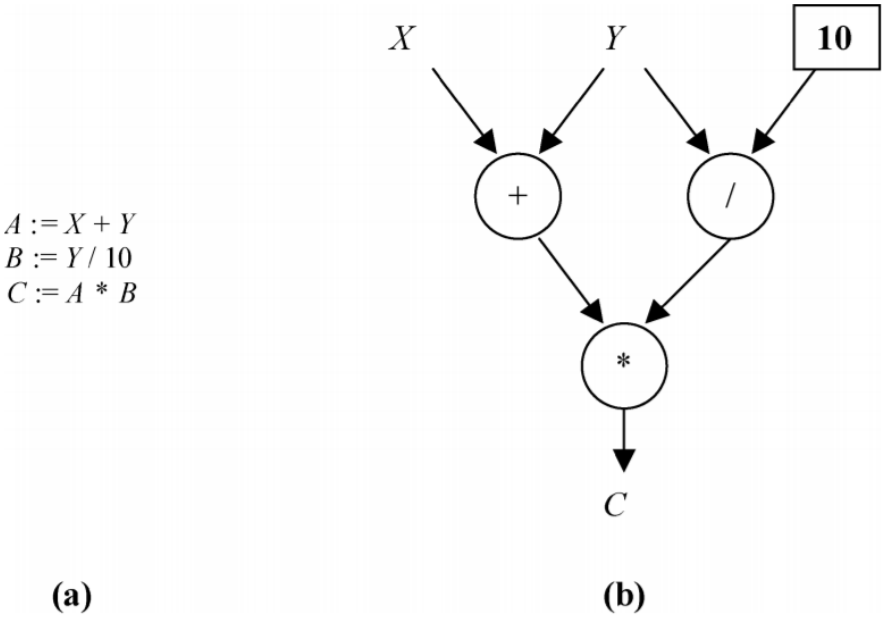
When a block is *evaluated* (also called *fired*), it performs the instruction it represents on the data from input ports and writes the results to the output ports. A block is *fireable*, if it has all its input ports ready (a value was written on them).

The execution of a dataflow program is simple. In the beginning, all the source blocks are fired, which means that the following blocks become fireable and therefore can also be executed. Every block is executed independently on each other and as soon as it gets fireable. That is quite different from von Neumann model, where an instruction is only executed when the program counter reaches it.

The most important advantage of this approach is that many blocks can be executed simultaneously and therefore can be easily run in parallel.

A simple example of the difference between classic and dataflow paradigm is shown on Figure 1.1. Part (a) shows familiar program instructions, part (b) depicts a dataflow graph of the same meaning. [8][7]

Figure 1.1: Difference between simple program (a) and a dataflow alternative (b)




Source: [8].

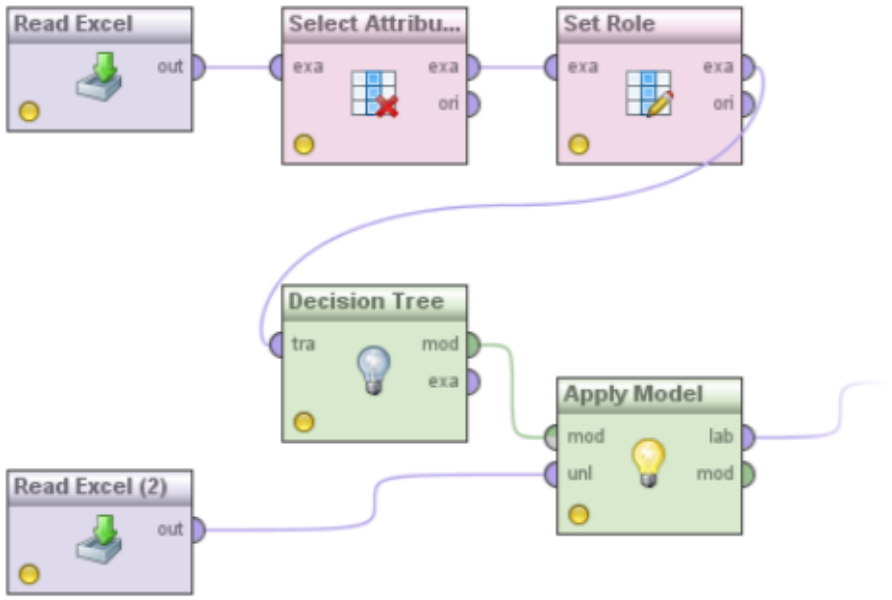
**1.2.1.1.3 Data flow programming use cases** An example of a dataflow programming in practise is a data mining tool Rapid Miner, where a user can perform different statistical and data mining operations on data. Blocks in Rapid Miner consist of such configurable operations. An example is shown on Figure 1.2.

However, a dataflow application does not always need to be a visualization of a graph. Even a spreadsheet processor, such as Microsoft Excel can fit into this category. Cells in each spreadsheet can either have a value or an expression with relation to other cells. The evaluation of a spreadsheet is the same as in the dataflow model: once a cell value is updated, the update is fired also on all the other cells, which depend on it. [7]

**1.2.1.1.4 Functional programming** Functional programming is another paradigm, which has many common features with dataflow programming. The computation of the program is carried out through the evaluation of expressions (functions). [9] Functional programming is based on a mathematical formalism known as $\lambda$-calculus and there are currently many different functional programming languages.

The main notion of the functional programming is that everything in a program is interpreted as a function, even all return values or function arguments

Figure 1.2: A dataflow operation – mining data from an Excel document.




Source: [8].

(concept known as *First-class* functions). According to this paradigm, there should not be any global values in the program and every function processes only its input parameters and output values (concept of *Pure functions*). Every function can also call itself and simulate a cycle instruction (concept of *Recursion*). [10]

Following the functional programming principles leads to better modularity of code and also easier testing and code readability. [2]

MCH-DFP module combines both dataflow and functional programming paradigma. On top level there is a dataflow programming model consisting of operations following the functional programming principles. Thanks to similarities between the two models is such composition straightforward.

**1.2.1.1.5 Domain Specific Language** Domain Specific Language (DSL) is a programming language which is focused on solving problems in a specifically defined domain. It is considered to be very high-level as it supports only limited set of operation. Unlike traditional programming languages, it does not need to be able to fulfil any programmatic problem, it suffices if it solves the problems from the target domain.

A high-level DSL is usually implemented via a middle-level programming language.

It brings several advantages to applications working in the given domain, as outlined by Ward in [11]:

- **Separation of concerns** - Operations in DSL are separated from its underlining implementation and a user of the DSL does not need to know anything about its specifics.
- **High development productivity** - Since the DSL is a high-level language, very complex functions can be implemented with just a few DSL lines of code.
- **Highly maintainable design** - According to several studies, the maintainability of a software system directly depends on its size. As the DSL is a high-level language, the DSL applications can be quite small.
- **Highly portable design** - Porting the DSL to another system requires no changes in DSL itself, but only rewriting the middle-level layer.
- **Opportunities for reuse** - DSL allows a user to use almost the same functions when dealing with similar problems.
- **User enhanceable system** - User experts from the given domain understand the language very well and can therefore contribute to its enhancement.

MCH-DFP module specifies its own DSL to allow the user define aforementioned rules, which the database repositories have to follow.

### 1.2.2 DFP interpret design and architecture

The back-end part of the DFP module, which is the DFP interpret of the aforementioned DSL, was implemented as a Master's thesis by Maksimau, so the complete documentation can be found in [2]. In this section, basic design and architecture of the DFP module will be summarized, including some implementation specifics important for this thesis.

#### 1.2.2.1 Basic components

DFP interpret uses the dataflow programming and therefore it uses all of its basic components: **nodes** (blocks) with **ports** which are connected by oriented **edges**.

**1.2.2.1.1 Nodes** Nodes represent operations. Every node has an arbitrary number of input and output ports. Nodes are connected through their ports with oriented edges. When a node is evaluated, the operation is applied to the data on input ports and the result is written to the output ports. There are several types of nodes in the interpret:

- **Atomic nodes**
  - **Literal nodes** - Nodes without input ports returning a specified value.
  - **Computational nodes** - Nodes transforming inputs to outputs.
  - **Modifying nodes** - A kind of computational nodes, which also have a side effect on the data flowing through the node (e.g. update of an attribute value).
  - **Side effect nodes** - Nodes without output ports performing only a side effect to outer environment (e.g. write to a report).
- **Composite nodes (components)** - Nodes encapsulating one or more other connected nodes.

**1.2.2.1.2 Ports** Ports are the means of a node to communicate with other nodes by allowing it to assign values to them. Every port can be either input or output. Every port also has a **type** of its value: input ports only accept data of the given type and output ports send the data of the given type.

Designing adequate type system for these ports is one of the objectives of this thesis.

**1.2.2.1.3 Oriented edges** Oriented edges simply connect an output port with an input port. Such a connection represents a data transfer, so whenever a value is written to an output port, this value is also copied to the given input port.

#### 1.2.2.2 Operations

The rules of MCH are applied to database repositories queries. In all technologies supported by MCH, these queries can be represented as a tree (XML, AST etc.). Therefore the interpret operations perform different actions on these trees. There are few main types of operations in MCH-DFP:

- **Filtration** - Filtering parts of tree based on some criteria. (e.g. getting all XML nodes with certain name)
- **Conversion** - Converting input data to output data of different type. (e.g. converting a XML node to set of its attributes)
- **Condition verification** - Evaluating if a certain condition is fulfilled. (e.g. all items in a collection are lesser than 5)
- **Modifying operations** - Manipulation with the tree. (e.g adding new nodes or removing them)

- **Reporting** - Writing to MCH reports which should be shown to the user as a result of the rule evaluation.

These operations are represented by nodes of DFP module and chained behind each other with edges.

#### 1.2.2.3 Technologies

MCH application is written in Java and DFP module had to consider this fact in order to be integrated with the product. The requirements given by the programming paradigm also had to be taken in account. From these two observations, JavaScript was chosen as a programming language and it is integrated into the Java application via Rhino JavaScript engine.

MCH-DFP works with custom rules, which are represented by DFP oriented graph. This graph needs to be persisted in some way and a technology for representing this graph had to be chosen. JSON format was picked for this as it is concise and it is easily processed by both machines and people. Representation of rules in JSON is more thoroughly described in the following section.

#### 1.2.2.4 Rules definition language

The language, in which the rules are described, is important for the second part of DFP module – its front-end – and also this thesis. Back-end part interprets the rules defined by it and the front-end part visualizes them as an oriented graph.

The rules are defined in JSON format with fixed structure:

```
{
  "components" : [...],
  "nodes" : [...],
  "edges" : [...]
}
```

where `components` is an array of component definitions, `nodes` is an array of nodes in the oriented graph and `edges` is an array of oriented edges between these nodes.

**1.2.2.4.1 Component definition** A component definition has the following fixed structure:

```
{
  "component" : "Component",
  "name" : "NewComponent1",
  "ID" : "new-component-1",
  "nprototype" : { ... },
```

```
  "structure" : {
    "nodes" : [...],
    "edges" : [...]
  }
}
```

Field `nprototype` specifies the node itself and has this structure:

```
{
  "input_ports": [
    {
      "id":"key",
      "type":"Symbol"
    },
    ...
  ],
  "output_ports": [...]
}
```

**1.2.2.4.2 Node definition** A node definition has the following fixed structure:

```
{
  "component" : "KnownComponent",
  "ID" : "component-1"
}
```

and it references an already known component.

**1.2.2.4.3 Edge definition** An edge definition has the following fixed structure:

```
{
  "component" : "Edge",
  "nprototype" : {
    "from" : {
      "node" : "from-node-id",
      "port" : "from-node-port-id"
    },
    "to" : {
      "node" : "to-node-id",
      "port" : "to-node-port-id"
    }
  }
}
```

where `component` specifies that it is an `Edge` and `nprototype` object defines the source and destination nodes and ports of the edge.

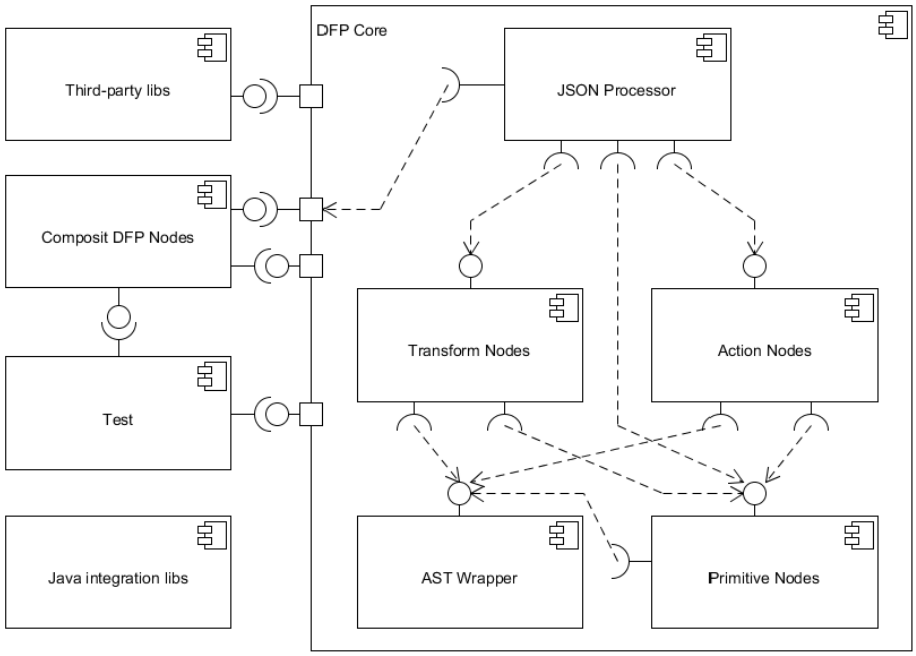
#### 1.2.2.5 Interpret architecture

Basic architecture of the DFP interpret is depicted on Figure 1.3. The main part of its core is JSON processor, which processes input JSON rule specifications and translates them into different types of nodes. After the translation, this representation of the rule is executed.

When the execution is triggered, all literal nodes are evaluated, which triggers the evaluation of following connected nodes and so the execution proceeds.

Other components of the interpret ensure integration with the Java based MCH and testing of the interpret itself.

Figure 1.3: Architecture of the DFP interpret




Source: [2].

### 1.2.3 DFP Editor

In the previous section, back-end part of DFP module, which interprets the rule specifications written in custom DSL, was described. Front-end part of

the module – DFP editor – is an graphical user interface (GUI, shortly UI), which visually represents the rules and allows the user to edit them or create new ones. This part will be described in this section.

The whole DFP editor was implemented by the author of this thesis and further extension of this user interface is part of the goals of this work.

#### 1.2.3.1 Basic functionality

As stated in previous sections, a dataflow program can be represented as an oriented graph and therefore the main functionality of the user interface is to clearly visualize it. However, users also wants to manipulate and edit the graphs representing rules and therefore there are several other main functional requirements[1]:

- Visualize the rule as a graph of operational nodes.
- Enable inserting of operational nodes from a palette.
- Manipulate the nodes in graph with drag&drop operation.
- Differentiate between different types of operational nodes.
- Enable editing of the oriented graph: deleting of nodes and edges, adding new edges.
- Allow cascade visualization of component nodes.
- Enable creation of new component nodes.

All of these functionalities were implemented and examples of the UI can be seen in the attachments A.

#### 1.2.3.2 Components of the user interface

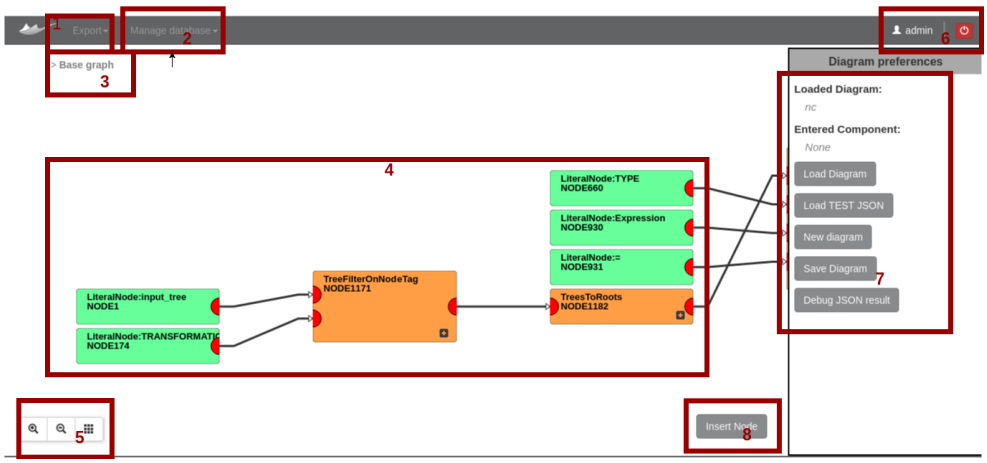
The Figure 1.4 depicts the user interface of the DFP module, where different components are labelled with a number:

1. **Exporting tools** - Menu enabling export of the rule to JSON or PNG.
2. **Tools for database management** - Administration tool for database saving the rules.
3. **Breadcrumb navigation** - Cascading viewing of components is possible, so this navigation shows the current level of the view and it is also possible to switch the levels using the navigation.

[1]The list is not complete, for full list of requirements see [7]

4. **Main graph window** - Canvas on which the rule is interactively visualized.
5. **Viewing tools buttons** - Buttons enabling zooming and layouting of the graph.
6. **User panel** - Logout of the current user.
7. **Diagram preferences** - Contextual menu showing the information about the rule or its building blocks. Contains also buttons for loading different rules.
8. **Palette button** - Button for adding new nodes from the palette.

Figure 1.4: Components of the DFP user interface




Source: [12], Author.

#### 1.2.3.3 Technologies

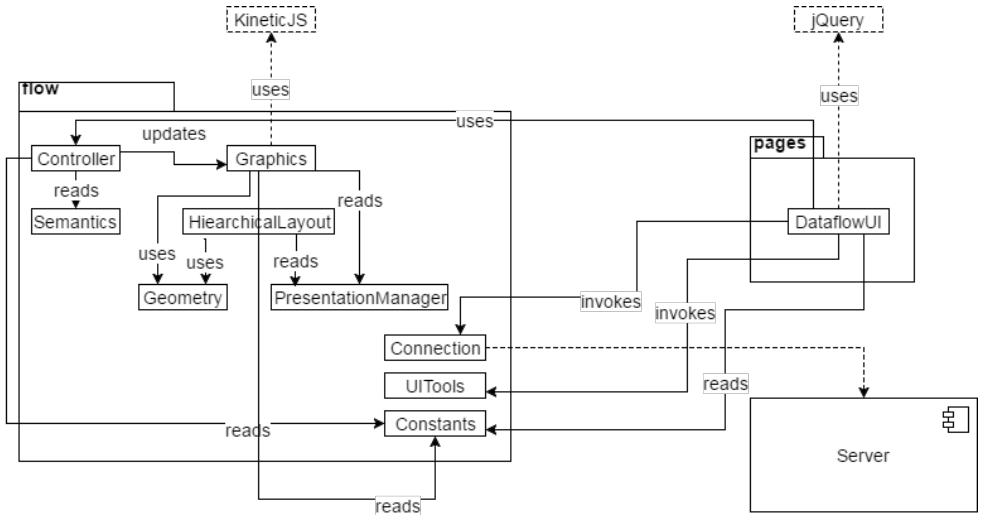
DFP editor is a web application run in a web browser and it is built using JavaScript programming language, Twitter Bootstrap framework for user interface components, jQuery library for user interface manipulation and KineticJS for graph representation. Minor part of the front-end uses Java Spring framework to store the rules, however it is not relevant for this thesis. [13][12]

#### 1.2.3.4 User interface architecture

DFP editor consists of several JavaScript namespaces, where each of them covers certain problem area in the application. This architecture is displayed on Figure 1.5. The following namespaces are relevant for this thesis: [13][12]

- **DataflowUI** - Contains functions controlling user interactions with the UI.
- **Semantics** - Defines the semantic model of the visualized graph and stores it.
- **Graphics** - Contains the graphic representation of the graph.
- **Controller** - Contains Controllers for the building blocks of the graph. These Controllers have a pointer to both semantic and graphic representation of blocks.
- **UITools** - Contains functions invoking different UI actions.

Figure 1.5: Architecture of the DFP editor.




Source: [12], Author.

### 1.2.4 Relevance to the goals of this thesis

As specified in the introduction, the goal of this thesis is to extend and upgrade the MCH-DFP module. Specifically, two new submodules should be implemented: a type system for MCH-DFP and a system for viewing and debugging logs from MCH-DFP.

#### 1.2.4.1 Type system

The type system of MCH-DFP spans through the whole application as it directly impacts the semantics of rules themselves, however there are more

places in the application where it can be checked whether the restrictions set by the rule system are followed:

- Checking at DFP interpret during the JSON processing or evaluation.
- Checking at DFP editor during the rule creation and design.

DFP interpret utilizes the functional programming and uses JavaScript language, which is not strongly typed, as a lower layer, which means that during evaluation there is no built-in type checking. However, DFP editor does not use types on the lower level at all. Therefore the fulfillment of the type system restrictions is checked in DFP editor part. For more information, see chapter 4.

#### 1.2.4.2 Log viewer

Log viewer and logging system influence both parts of MCH-DFP. DFP interpret creates logs during the execution and these logs are later viewed directly on graph in DFP editor. For more information, see chapter 5.

Chapter 2

# Type systems in programming languages

In this chapter, different type systems used in programming languages will be described. Firstly, the general meaning of a type system will be portrayed, secondly, the most important forms of type system will be enumerated and finally the type systems most relevant to this thesis will be characterized.

## 2.1 Type system

Modern software applications are growing more complex and therefore the popularity of different *formal checkers of correctness* rises. However, most of these formal methods are quite complicated and it is hard to incorporate them in common tools used in software engineering, such as compilers or static code analysis tools. Therefore lightweight formal checkers are required and type system is considered to be one of them. [14]

A type system assigns a type to every construct representing a value in a programming language. The system then uses these types to build interfaces between the constructs and specifically to determine whether two incompatible types are interacting together (and prevent them from doing so). Such incompatibilities can be e.g. logical errors.

Type systems used in programming languages are sometimes rather practical, however there are also formalized type systems with mathematical proven qualities.

Formalized type systems usually require precise notation and definition, sometimes relying on heavy abstraction. [15] Some of the formalizations will be summarized in the following sections, however the type system in this thesis will be rather constructed from real type systems from other programming languages. These foundations for the type system will be also described.

### 2.1.1 Definition

In this section, the definitions of a type and a type system will be presented.

#### 2.1.1.1 Type definition

Before defining a type system, the definition of a type will be given first. [16] defines the type in programming languages as follows:

> (Data type) A set of values from which a variable, constant, function, or other expression may take its value. A type is a classification of data that tells the compiler or interpreter how the programmer intends to use it. For example, the process and result of adding two variables differs greatly according to whether they are integers, floating point numbers, or strings.

This is a rather technical definition and so more general one from [17] is also presented:

> A type is any property of a program that we can establish without executing the program.

It can be deduced from both of the definitions, that a data type in programming languages is basically a meta datum or a classification of a data value in the language. The main difference between the two definitions is that the second definition does not assume the direct presence of a type in a code and implies that types can also be inferred from the code.

Such definition is more precise, which will be shown in the following sections, where different type systems will be presented.

#### 2.1.1.2 Type system definition

One of the possible definitions of a type system in programming languages is stated by [14]:

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

The definition is quite broad, as it tries to cover both branches of type system research in computer science: the practical branch, which focuses mostly on usage of types in common programming languages and the theoretical branch, which aims mostly on logical concepts in proof theory, $\lambda$-calculus, etc.[2].

Some basic properties of a type system can be deduced from the definition:

[2]The first formalizations of a type system are from the beginning of 20th century. [14]

- A type system is a syntactic method, as it is usually built-in in the programming language which uses the type system.

- A type system should help the compiler or the run-time environment to determine whether there are type errors in the code, which occur, if two incompatible types try to interact together[3]. The compiler wants such a behaviour to be absent.

- Some type systems can classify the types from the surrounding expressions in the code, not only from the syntactic type declaration itself.

### 2.1.2 Typed and untyped languages

From the definitions above, we can clearly state the difference between typed and untyped languages: [15]

#### 2.1.2.1 Typed languages

Typed languages use a type system, which means that every variable (or any other language construct, such as expression or function) in an application has a type and therefore only values of such type can be assigned to it. Most of the currently used programming languages have at least some kind of a type system.

#### 2.1.2.2 Untyped languages

Untyped languages do not leverage a type system and its constructs are not restricted. These constructs either do not have type at all or just one universal type is used throughout the language. However, during the execution of the application, operations with these variables can lead to faults, exceptions or even unspecified effects. A pure $\lambda$-calculus is an extreme case of an untyped language.

### 2.1.3 Implicit and explicit types

#### 2.1.3.1 Explicitly typed languages

An explicitly typed language is such a programming language, where the types are part of the language syntax and types are assigned to variables inside the code. Annotating programs with types can however lead to a lot of redundancies and increase the size of the source code significantly.

[3]These errors are also called *execution errors*, see subsection 2.1.4.

#### 2.1.3.2 Implicitly typed languages

Implicitly typed language is a language, where types are not in the syntax and are inferred from the context. However, such procedure can sometimes lead to unwanted results and it is possible, that some variable types are inferred wrongly.

#### 2.1.3.3 Comparison

Because of the properties of these two type systems, a combination of both can be used, where a certain degree of implicit *type reconstruction* is used in a generally explicit type system. [15] [18]

### 2.1.4 Execution errors and language safety

As stated above, the prevention of execution errors is one of the main reasons to introduce a type system to a language. There are two main types of execution errors: **trapped errors**, that cause the computation to stop immediately and **untrapped errors**, which can be undetected for an unspecified time and cause an arbitrary behaviour.

A programming language is considered *safe* if it does not allow occurrence of untrapped errors. Both typed and untyped languages can be safe. Untyped languages can perform runtime checking of adherence to the error free rules. Typed languages can do so as well and in addition they perform static checking of source code and reject any application that can be unsafe. [15]

### 2.1.5 Statically and dynamically checked languages

Based on these definitions, we can distinguish between **statically checked** and **dynamically checked** languages.

#### 2.1.5.1 Statically checked language

A statically checked language performs the type checking process before the program execution. If a source code passes the static checking, it is guaranteed that the program will satisfy some safety properties. Ill typed applications are not allowed to be executed. Compiled languages such as Pascal are an example of this kind.

However, there are features in programming languages, which cannot be statically checked. An example of such a feature is *type casting*. [14]

#### 2.1.5.2 Dynamically checked language

A dynamically checked language performs sufficient checks during the application run time to rule out untrapped errors (or even some trapped errors, if required). LISP is an example of this kind.

Even languages with static checker usually include some part of dynamic checking as well (e.g. to support aforementioned type casting). Such typing can be called *soft typing*.

### 2.1.6 Strongly and weakly typed languages

A language is **strongly typed** if "each data area (has) a distinct type and each process (states) its communication requirements in terms of these types." [19]

Generally, it means that if a language strongly requires explicit type assignment, it is considered strongly typed, otherwise it is weakly typed.

### 2.1.7 Nominal and structural type systems

Type systems used in practice can be either nominal or structural.

Types in **nominal type systems** always have names, these names are significant for the type classification and subtypes of them are explicitly declared. The name of a type is then used to determine, whether two types are equal or if one is subtype of another. An example of such a type system is the one in Java language.

However types in **structural type systems** can be nameless, their names are simply a label for readability purposes. In these systems, the equality is determined purely using structural properties of the types, e.g. two object types can be equal, if they offer same methods and have same properties. [14] An example of such a type system is the one in Go language.

If a structural type system checks only for equality of methods and not other properties, it is called **duck typing**. [20]

### 2.1.8 Advantages and disadvantages of the presence of a type system

Introducing a type system to a programming language brings a new level of complexity and abstraction to it and therefore it is important to examine the impact of the type system to the language. [14] [17] [15]

#### 2.1.8.1 Advantages

- **Error detection** - One of the main advantages is detection of execution errors, because thanks to a type system, programmers can locate some errors very fast even before compiling and running the code (in case of static type checking). Even dynamic type checking prevents undefined behaviour of an application. However, introducing a type system cannot prevent all errors and bugs from happening.

- **Abstraction** - Type system always brings certain level of abstraction to a programming language, which makes the application programming easier. Thanks to abstraction, the programmer does not need to deal with byte representation of data. Abstraction can also enforce more disciplined programming and better component composition of an application.

- **Documentation** - Declaring types in the programming language makes the code more self-documented, as the type declarations give the programmer hints about the code behaviour.

- **Language safety** - As stated above, a type system can make a language safe, which is strongly connected to the error detection advantage.

- **Efficiency** - Even the languages without type systems have to perform certain type inference and statistic calculations about values and using a type system with already declared types makes these calculations obsolete and therefore the applications can run faster. First type systems were actually introduced because of efficiency issues. [21] Presence of a type system can also significantly help the compiler, as it can optimize the operations differently for different data types.

- **Code readability** - An introduction of a type system to a language does not enforce a coding style, however it can prohibit certain kinds of bad coding styles and therefore improve the code readability

#### 2.1.8.2 Disadvantages

- **Complexity** - Introducing many different types to a programming language can significantly increase its complexity, especially with complicated abstractions.

- **Limitations** - A type system limits the language syntax by putting constraints on it and these constraints can reject even some valid applications.

- **Poorly designed type systems** - This is not a flaw of type systems in general, however some type systems are designed in such a way, that often different programming ”hacks” are necessary, such as type casts in Java language.

## 2.2 Review of existing type systems

In this section, important formal and informal kinds of type systems will be described.

Type system is considered formal, if it is mathematically defined and proven, otherwise it is considered as informal (which is the case of most of the real type systems in programming languages).

### 2.2.1 Formal type systems

The following review of formal type systems is derived from [14] and [15].

#### 2.2.1.1 Untyped systems

Untyped languages were already defined in 2.1.2.2 however it is worth mentioning their non-existing type system as an important representative in the type system theory.

**2.2.1.1.1 Pure $\lambda$-calculus** The most important untyped system is *pure $\lambda$-calculus* as it was used in specifications of many programming languages. It is a formal system where all computations are composed of the function definitions and applications. The importance of the $\lambda$-calculus comes from the fact that it can be viewed as a simple programming language, but also as a precise mathematical object, about which correct mathematical statements can be proven.

In $\lambda$-calculus, every construct is a function, arguments received by functions are functions and so are their return values. Therefore the syntax of $\lambda$-calculus is very simple and it consist of just three sorts of terms: *variables*, *abstractions* (function definitions) and *applications*.

These sorts of terms $t$ are usually denoted as follows:

- variable $x$
- abstraction $\lambda x.t$, where $x$ is the head of an abstraction and $t$ is its body.
- application $t\,t$

$\lambda$-calculus expressions also include brackets for brevity and operation order.

Applications can be written with just this three sorts of terms, as $\lambda$-calculus can serve as a programming language on its own.

The evaluation of a $\lambda$-calculus expressions is very simple and resembles cut-paste operation in text processors. Functions can be resolved if they are followed by an expression. All occurrences of the head variable in the function body are replaced with the expression after the function and this new value serves as a new expression.

Functions can also be chained together, e.g. as $\lambda x.\lambda y.xzy$ which can be shortly denoted as $\lambda xy.xzy$. The evaluation of such functions works on the

same principle, first head variable is replaced with first expression after the function, second with the next one, etc.

To demonstrate this on an example, $(\lambda y.x(yz))(ab)$ can be resolved to $x(abz)$. The only operation here was the replacement of $y$ occurrence in the body with the following expression $ab$. [22]

#### 2.2.1.2 First-order type systems

First-order type systems are such type systems, which include higher order functions, but these systems lack type parametrization and type abstraction. First-order systems assign types only to language constructs, not to the types themselves. [23]

Similarly to the previous section, an example of a first-order type system is *typed λ-calculus*, however even type systems in well known programming languages such as Pascal or FORTRAN are first-order.

**2.2.1.2.1 Typed λ-calculus** To extend the definition from 2.2.1.1.1, types need to be added to λ-calculus terms. It needs to be done only for abstractions:

- abstraction $\lambda x : A.t$, where $x$ is the head of an abstraction, $A$ is its type and $t$ is its body.

Every abstraction (or function) therefore has an input type, but it also has its body, which can also have a type. That means that a function can transform one type to another. Therefore two kinds of types are distinguished: *basic types* which are usually part of the language and function types $A \to B$, which is the type of functions with arguments of type $A$ and results of type $B$.

#### 2.2.1.3 Second-order type systems

Second-order type systems include type parametrization, type abstraction, or both. Type parameters are present in languages where a generic component of an application is parametrized by a type, which is then supplied later. Type abstractions are also found in application modules, where they appear as opaque types inside the interfaces.

The main reason behind the development of second-order type systems are the limitations of first-order type systems. In these systems, every function written in its language works only for exactly one type. If there is a function, which works exactly the same for multiple types (e.g. sorting arrays with different item types) and differs only in the type, it is necessary to write this function for each of these types. [23]

Typical feature of second-order type systems is *type polymorphism*, which solves the aforementioned issue.

**2.2.1.3.1 Second-order typed $\lambda$-calculus** To obtain a second-order type system, the typed $\lambda$-calculus from the previous section can be extended with *universally quantified type* and two new terms: *polymorphic abstraction* and *type instantiation.*

- polymorphic abstraction $\lambda X.t$, where $X$ is a type variable and $t$ is a term, meaning that program $t$ is parametrized with type $X$.
- type instantiation $tA$, where $t$ is a term and $A$ is a concrete type, which is used to instantiate polymorphic abstractions.

*Universally quantified type* is the third kind of types and is written as $\forall X.A$. We can assign such a type e.g. to term $\lambda X.t$, meaning that for all $X$, the body $t$ returns type $A$.

**2.2.1.3.2 Type polymorphism** is an interesting feature of second-order type systems. Polymorphism is an ability of a program construct to have multiple types, which means that if a function is polymorphic, it can be applied to multiple different argument types. [23]

Polymorphism allows a code construct to have a generic type, which is used instead of specific type the construct actually has. This specific type can be instantiated later during the execution, if it is required.

#### 2.2.1.4 Type systems of object oriented languages and subtyping

Object oriented languages are one of the main families of programming languages and they usually have complex type systems. One of their main features is *subtyping* which intuitively allows inclusion of one type in the other. A language construct with a type can be also considered as an element with type of any of its supertypes, which allows the construct to be used in many different functions and contexts.

Formal languages with subtyping feature therefore define a subtype relation $A <: B$ meaning that A is a subtype of B.

### 2.2.2 Informal type systems

In this section, most important kinds of type systems from widely used programming languages will be briefly summarized. The focus is on the most popular languages in 2016 according to [24] and [25]. Basic comparison of the type systems in these languages can be seen in Table 2.1.

From the comparison it can be seen, that modern and popular languages leverage all kinds of type systems.

More traditional languages such as C (and its extensions) or Java have static type systems, however they also introduce elements like type casting,

which can only be checked dynamically. On the other hand, dynamic languages such as Javascript often offer extensions which can introduce static type checking into the language. TypeScript is an example of such a extension.

From the table it can be seen that there is quite a connection between static and strong typing and dynamic and weak typing, however low level language like C also has weak typing and on the other hand modern Python is strongly typed.

It can be seen that dynamic language use duck typing, meaning that as long as the object of a type offers the same methods as the other type, it can be considered compatible. In dynamic languages, this check is done at the runtime, meaning that unimplemented method will not be discovered until it is tried. Most common static languages use nominal typing.

Table 2.1: Comparison of type system in popular programming languages

| Language | Static / Dynamic | Strong / Weak | Nominal / Structural |
|---|---|---|---|
| C | static | weak | nominative |
| Java | static | strong | nominative |
| Python | dynamic | strong | duck |
| C++ | static | strong | nominative |
| JavaScript | dynamic | weak | duck |
| PHP | dynamic | weak | duck |
| SQL | static | strong | nominative |
| C# | static | strong | nominative |

Sources: [26], [27], [28], [29], [30], [31], [32], [33]

## 2.3 Relevant type systems for MCH-DFP

In this section, Hindley-Milner type system will be briefly described. Afterwards, the type system in Haskell programming language will be summarized. The type system developed in this thesis is strongly inspired by some elements from Haskell language and Haskell's type system is based on Hindley-Milner theorem.

### 2.3.1 Hindley-Milner type system

The title Hindley-Milner (sometimes also called Damas-Hindley-Milner) refers to a $\lambda$-calculus type system, which was developed by Hindley [34] and later described again by Milner [35].

Their work also presents an algorithm for this type systems, which allows type inference of values in the programs based on their use. If formalized

Hindley-Milner type system is used, any type in the program can be deduced by its functionality. [36] The Hindley-Milner theorem specifies rules, by which these types can be inferred from expression statements (called *terms* in previous section) and the algorithm applies these rules.

Hindley-Milner type system has several interesting properties, which cause the high popularity of this theorem: [37]

- It is abstract and generic, so it can be applied to arbitrary language and the types can be inferred purely from the form of the statements.
- It leverages definitions of expressions from $\lambda$-calculus, which is well known theoretical concept in computer science, to unambiguously define types of these expressions.
- Its formalization can be easily used in practice by implementing it in a programming language.

The following formalizations are based on [37], [34], [35].

#### 2.3.1.1 Formal foundations

This subsection defines basic formal foundations required for the definition of the type system itself.

**2.3.1.1.1 Expressions** Expressions in the Hindley-Milner type system are terms from pure $\lambda$-calculus, see 2.2.1.1.1.

**2.3.1.1.2 Statements about types** Similarly to 2.2.1.2.1, claim "expression $e$ is of type $t$" can be denoted as:

$$e : t$$

*Functions* (called *abstractions* in previous section) also have types and so do their input parameters and return values. Function with input value of type $t$ and return value $s$, where $t$ and $s$ are types, has type:

$$t \to s$$

Polytype functions can exist, which means that for any type $t$, their type is $t \to t$. The type of such polytype function is denoted as:

$$\forall t.t \to t$$

**2.3.1.1.3 Type inference rules** The Hindley-Milner type system consists of set of rules used for type inference and these rules can be denoted by special mathematical syntax, using type inference operator $\vdash$. An example of such inference is:

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0(e_1) : \tau'}$$

Horizontal line divides premises and conclusions, meaning that with knowledge of statements above the line, the statements bellow the line can be concluded.

$\Gamma$ stands for the set of statements that are already known or assumed. These are statements and rules about types which were declared before.

Generally speaking, the syntax describes following claim: "It can be inferred that the type of $e_0$ is $\tau \to \tau'$ (so it is a function). It can be inferred that the type of $e_1$ is $\tau$. Therefore it can be concluded that it can be inferred that the type of application of function $e_0$ to $e_1$ is $\tau'$."

### 2.3.1.2 Type system formalization

This subsection will describe the rules for deducing types of statements from the Hindley-Milner type system.

#### 2.3.1.2.1 [Var]

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

According to this rule, if a statement about expression $x$ having type $\sigma$ is in assumptions $\Gamma$, the type $\sigma$ of this expression $x$ can be inferred from the assumptions $\Gamma$.

#### 2.3.1.2.2 [App]

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0(e_1) : \tau'}$$

This rule was already used as an example in 2.3.1.1.3 and was explained there. The rule describes the types of function applications.

#### 2.3.1.2.3 [Abs]

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

According to this rule, if it can be inferred from the assumptions and from $x$ having type $\tau$ that $e$ has type $\tau'$, then it can be concluded that it can be inferred that function $\lambda x.e$ has type $\tau \to \tau'$. This rule exists because of polytyped functions, which can e.g. expect an array and return a first element of such array. Such a function would have a type $Array[T] \to T$.

#### 2.3.1.2.4 [Let]

$$\frac{\Gamma \vdash e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : \tau}$$

According to this rule, if it can be inferred that $e_0$ has type $\sigma$ and if it can be inferred from the fact that $x$ also has type $sigma$ that $e_1$ has type $\tau$, then it can be inferred that if $e_0$ is substituted with $x$ in $e_1$, the type of such expression is $\tau$. This rule exists to allow valid (in terms of types) substitutions in the expressions.

#### 2.3.1.2.5 [Inst]

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma}$$

This rule uses the "less specific" operator $\sqsupseteq$, meaning that a type is less specific than another type. The operator is used with polytypes.

According to this rule, if it can be inferred that $e$ has type $\sigma'$ and $\sigma$ is less specific than $\sigma'$, then it can be inferred that $e$ also has type $\sigma$. This rule basically allows subtyping (the direction from child to parent type).

#### 2.3.1.2.6 [Gen]

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma}$$

According to this rule, if the variable $\alpha$ is not a free variable in $\Gamma$ assumptions and if it can be inferred that $e$ has type $\sigma$, it can be inferred that $e$ has polytype $\forall \alpha.\sigma$. This rule basically allows definitions of polytypes with variables, which have not been used in the context before.

### 2.3.2 Type system in Haskell

Haskell is a purely functional and statically typed language, where the type of every expression is determined during the compilation. It highly leverages the type inference, which is basically the implementation of aforementioned Hindley-Milner algorithm. Evaluation of functions and its arguments

in Haskell is lazy, which means that they are evaluated only when it is needed, which brings several performance benefits. [38]

In this section, some interesting features from the Haskell type system will be described, as they will serve as a basis for the type system in the practical part.

#### 2.3.2.1 Type system classification

As it was already mentioned, the type system in Haskell is **static**, because the type soundness of a program is checked during the compilation, before any application can be executed. However, unlike e.g. Java, Haskell does not require explicit type assignment, these types can be inferred, and therefore can be considered as **weakly typed**. Finally, Haskell determines its types rather from structural properties than names and so it is a **structural** type system. [39]

#### 2.3.2.2 Types in Haskell

Haskell provides basic types for all common data structures, such as `Bool`, `Char`, `String`, `Integer` etc. Haskell also has lists (denoted as `[item_a, item_b]`) and function types. [40]

An interesting fact is that these types behave exactly as in Hindley-Milner theorem, meaning that e.g. function which converts all letters in a `String` (which is just a list of `Char`s) to lower case and returns such a converted string has a type `[Char] -> [Char]`.

Haskell also supports polytypes, which are called *type variables*. They are used to introduce polymorphic functions in Haskell. An example of such a function is `head` returning the first element of a list. The list can be composed of items of one arbitrary type, so the type of the function is `[a] -> a`. The letter `a` represents a type variable here. [41]

It is possible to introduce your own types in Haskell.

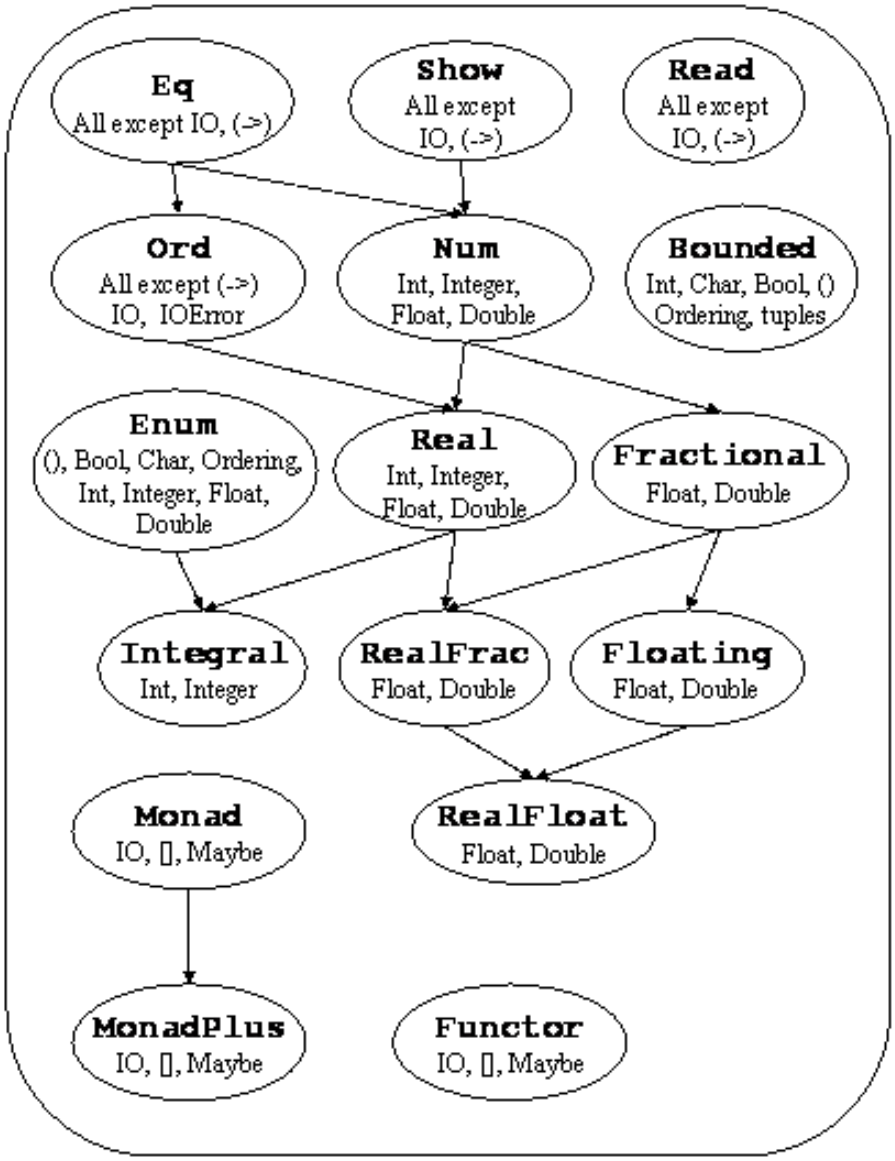
#### 2.3.2.3 Typeclasses in Haskell

Another important notion in Haskell is a typeclass. A typeclass is basically an interface which prescribes some behaviour, meaning that if a type supports a typeclass, then it implements this prescribed behaviour. Types in Haskell can be part of any number of typeclasses and new typeclasses can be defined by user.

An example of a built-in typeclass is `Eq`. Its members have to implement methods `==` and `/=` which are used for equality and inequality testing.

The Figure 2.1 shows the overview of built-in typeclasses in Haskell. The figure also presents their hierarchy, as typeclasses can be part of another typeclasses, like `Eg` is part of `Ord`. Text under each typeclass enumerates which basic Haskell types are part of this typeclass.

Figure 2.1: Overview of Haskell typeclasses.




Source: [40].

#### 2.3.2.4 Type Maybe

Very useful type in Haskell is a `Maybe`, which encapsulates an optional value, i.e. a value that may have not been initialized. When an expression has a type `Maybe` it can either have an empty value (represented as type `Nothing`) or a value of some type variable `a` (represented as `Just a`). Having such type allows much cleaner data processing and avoidance of runtime errors. [42]

Chapter 3

# Tracing and logging of Data Flow Programming applications

In this chapter, the tracing and logging means of Data Flow Programming applications will be presented. Firstly, the general concept of tracing and logging in computer programs will be described. Then possible solutions for tracing and logging in DFP applications will be outlined.

## 3.1 Tracing and logging of computer programs

Application programming is a complex discipline and emergence of errors during the creation process is inevitable. Almost every programming platform therefore comes with a possibility of logging these errors and tracing the computer steps which led to them.

In this section, the terms tracing and logging will be defined and then the approaches to tracing and logging of computer programs will be enumerated.

### 3.1.1 Definition

#### 3.1.1.1 Logging

Logging is a process of tracking events and operations which occur in a computer program. An entry in a log is usually a message describing an event that just happened, possibly with the data in the event's variables. Messages in the log can also have different importance, often called *severity*. [43] Logging collects the events of interest that occur during the program execution. [44]

The logs are usually supposed to be readable by humans, but certain level of computer processing might also be necessary. The logs should have a sufficient level of detail and they should follow some standardized structure.

From each log entry, a reader should be able to determine what happened in the program, when, where and how. That usually requires including infor-

mation like *username* of the user working with the application, *object* affected by the event, *status* of the action taken, *location* of the event in the system, originating *source* of the event, *timestamp* of the event or *reason* why this information is logged. Entries could also contain *advices* what to do with the information the message carries and its *severity*. [45]

#### 3.1.1.2 Tracing

Tracing is a special form of logging and it keeps information about execution of a computer program. While logging can be focused only on some high level events, such as accesses to the application, tracing can record every operation related to such access. Tracing is mostly used by the application developers for debugging purposes in the application development process, whereas logging can serve many other reasons. [46]

### 3.1.2 Tracing and logging approaches

In practice, there are several main approaches to logging and tracing. The basics of these different ways will be described in this section.

#### 3.1.2.1 Logging

Event logging is part of computer programs for more than three decades and therefore there is a variety of logging systems in practical use.

Important differences arise from a distinct approach to what should be logged. Generally, the logs existence should help indicate an occurrence of unwanted software faults, which requires only these failures to be logged, however many systems might even need logging of harmless events and operations. In the first case, where the logs serve to localize failures, it is important that these logs are able to record every operation, that leads to such failures. Many traditional logging patterns, such as logging with a code placed at the end of set of instructions (see bellow), do not fulfil this requirement, as they are prone to miss certain kind of errors, e.g. infinite loops.

Some approaches such as IBM Common Event Infrastructure offer various means of manipulation with logs, however they do not address how the log entries are actually created in the application code. Aspect Oriented Programming (AOP) promotes having a logger as a cross concern feature in the system, which can lead to more reliable systems, however to support AOP, a special programming framework is required. [44]

[44] localizes three most common error logging patterns in programming languages:

- `if (condition) then log_error()`
- `try {...} catch (...) {log_error()}`

- `else log_error()`

The aforementioned paper also claims that these patterns are ineffective and proposes a logger capable of monitoring changes in the control flow of the application.

A logging approach can be also either centralized or decentralized. In the first case, the log entries are directed to one or small number of locations, while in the latter case there might be a log location for every module of the application. [47]

#### 3.1.2.2 Tracing and debugging

Tracing is usually part of the debugging process and almost every debugger approaches it similarly: when a bug is spotted, a developer usually sets a breakpoint near a problematic section of code and when the execution reaches the breakpoint, the developer can perform single steps forward to analyse the cause of the problem.

Such an approach might lead to problems in event driven systems, because it might not be clear where to set up the breakpoints. That holds also for data driven systems, where also DFP belongs. Another problem may arise from the fact, that usually the environment where the application is executed and where the bug was found is different from the one where the program is debugged. Some errors therefore might be irreproducible. This issue is tackled with complex tracing systems which are included in the application and which log all the operations and values even without debug environment. [48]

There are several approaches to software debugging and all of them leverage tracing as an underlying tool. The most common approaches are found bellow: [49]

- **Brute force method** - Method incorporating collection and analysis of the whole tracing log.
- **Back tracking method** - Method starts at the point of failure and backtracks through the call in tracing log to the original cause.
- **Cause elimination** - The method systematically removes parts of system from the list of possible causes of an error.

## 3.2 Tracing and logging of DFP applications

The nature of DFP programs is very specific, especially of the visual ones, as they do not consist of set of instructions, but rather of interconnected modules between which the data flow. From this reason, there is not much research

about the direct tracing and logging of DFP applications[4]. As these programs depend heavily on the data inside the applications, it is crucial that the user can determine where the dataflows lead to and what data they contain.

It has been recognized that visual DFP programs need to have a way how to debug them and examine their logs visually, thereby matching the style of debugging task to the style of the programming one. [50] This also means that users of a DFP language should be able to debug the programs in this language, they are not interested in debugging the underlying layer, on which the DFP program is implemented. Such a layer should be considered flawless by the user.

A concern closely related to tracing and debugging is software testing. [51] and [52] identify common DFP faults, such as missing control modules or links between them, and propose a framework for testing DFP applications (presented on Prograph language).

As stated above, it is important that a user can examine the DFP program visually. [53] propose a debugger which leverages so called "Fish eye view", which enables the user to view both local information in certain part of a DFP program and global information about the whole application. An important outcome from this paper is that it is essential for debugging to be able to quickly switch between levels of details of the debugging information.

[4]Many visual DFP applications offered debugging tools, however the debugging happened on the lower non-visual level, which contradicted the requirement, that the development in these application will not require high technical abilities. [50]

# Part II

# Practical part

In the practical part, solutions achieving the goals of this thesis will be described. The resolutions presented in this part are based on the theoretical background presented in the previous part. Firstly, a proposed type system for MCH-DFP will be described, including its design and development process and then its functionality will be demonstrated on real world example. Secondly, a tracing and logging solution for MCH-DFP will be presented, its design and implementation.

It is important to remind the reader, that MCH-DFP is an existing product already in advanced stage of development and therefore the goals of this thesis were to extend it with new modules, rather than implement a system from scratch. The designs from this part therefore had to follow not only the state-of-the-art practices presented in the previous part, but also the current architecture of the MCH-DFP. However, it is worth mentioning that main parts of both the type system and logging system are present in the front-end part of the application, which was developed solely by author of this thesis in previous years.

During the design and implementation of the new modules, the following process has been ensued:

1. Requirements for the module were collected.
2. Based on the requirements and state-of-the-art solutions presented in previous part, an architecture for the module was designed and picked.
3. Proper technologies for the module were chosen, based on the requirements, current technologies in the system and available helper libraries.
4. The module was implemented according to proposed design.
5. The module was tested on real world examples.

During all phases of the process, current progress of work was discussed with the main stakeholder of the project, Manta tools company representatives, and the requirements and designs were adjusted according to their feedback. From the software engineering point of view, even as the process seems like typical waterfall development, the stakeholder feedback was present during all stages of modules creation process, making the work more agile.

Chapter 4

# Type system for MCH-DFP

MCH-DFP interpret developed by [2] already contained basic type system. As stated in subsubsection 1.2.2.1, MCH-DFP system consists of nodes with ports, which are connected by edges. Two ports can be only connected if their types are compatible. This means, that if two ports cannot be connected, there cannot be any dataflow between them and therefore no execution happens on these ports.

From this notion it can be deduced, that the original MCH-DFP interpret language was definitely a **typed language** and its type system was **explicit**, because every port had a specified type. On the other hand, the interpret itself did not perform any static type checking and as it was based on dynamic language JavaScript, it was also substantively **dynamic**. That also means, that even though the types were named on the ports, these names were never checked, and therefore the type system was inherently **structural**. The ports were required to have a type specified, which made the type system **strongly typed**.

During the development of MCH-DFP interpret, there was not much focus on the type system and therefore there were significant concerns with its original design. These concerns will be evaluated in the following section and a system resolving these issues will be designed and implemented.

## 4.1 Design of type system

### 4.1.1 Original type system issues

Following problems were localized in the original type system:

- The type system relied purely on the JavaScript type system and did not leverage any information about types it already had on its ports.
- The type system was inconsistent in terms of its data types. E.g. there were several different collection types such as `List` and `Seq` which were

in the end implemented just by simple JavaScript array and did not differ from each other.

- The type system used complicated concepts from JavaScript type system which might not be understandable for the typical MCH-DFP user, such as type coercion[5].
- The type system did not address the problem of possible `NULL` values, which could occur on certain nodes. For example, a node returning the first element from a collection can return `NULL` value if the collection is empty. An occurrence of such values led to unexpected errors.
- The type system did not use any kind of abstraction and polymorphism, and therefore if a user wanted to have e.g. five collections of five different types, he would have to have five different computational nodes for operations on these collections.

### 4.1.2 Requirements

Based on the flaws in the original type system and the expectations of the product owner, following requirements for the new type systems were set. The type system should:

1. be simple enough so even users with lesser technical knowledge will be able to work with it.
2. control the whole dataflow of an application, allowing handling of potential errors.
3. offer type polymorphism, so e.g. collection functions can have just one implementation as a node.
4. be applied on the DFP editor layer, so the user can see which types DFP nodes have.
5. help its user with understanding of the concept of type polymorphism by visually showing which specific type is actually in use on a given node.

Requirement 1 is strongly connected to the fact that the whole application is supposed to be used by users with lesser technical knowledge and therefore also the type system should be adequately simple.

Second requirement is directly related to the issue with handling of `NULL` values and the third requirement to the issue about the need of function repetition.

[5]Type coercion is a way of type casting, in the application it was e.g. possible to connect a port of type `T` to port of type `Collection of T`. Such operation goes against purity of the type systems and makes it more complicated to understand.

Requirement number 4 tackles the first issue and basically enforces static type checking on the front-end layer. Last requirement leverages the visual DFP so a user can be assisted in his work with types using visual hints.

### 4.1.3 Addressing the requirements

As it is required to apply the type system mostly on the layer of DFP editor, it means that the type system will be **statically checked** during the creation of visual dataflow. The editor will only allow connecting two ports together if they types match according to the type system.

The nature of data flow programs is that they are inherently *lazy evaluated*, any node in the dataflow is evaluated if and only if the dataflow reaches the node.

As was described in subsection 2.3.2, Haskell language is also statically typed and has lazy evaluation, so it was one of the main candidates for inspiration for the type system of MCH-DFP. Haskell is also considered to have very strong type system, as its type inference is basically an implementation of the Hindley-Milner algorithm.

The type system in MCH-DFP and the requirements on it are considerably less complex than the ones in Haskell, however Haskell has proven to be a good inspiration for some issues in MCH-DFP type system. There is no need for type inference in MCH-DFP, nonetheless e.g. the `Maybe` data type or typeclasses in Haskell could have been useful also in this type system.

Therefore the type system developed in this thesis is leveraging several principles from Haskell language, as is described in the following section.

### 4.1.4 Type system design

The type system was designed based on the requirements and on the existing MCH-DFP architecture. The proposed type system has following characteristics:

- It is **static** as all of the type checking is performed on the layer of front-end DFP editor. The types are not checked at ”compile” time, as there is no event such as compilation, but rather at ”create” time – the user is not allowed to connect two ports with incompatible edges.

- It is **nominal** as the type checking is based on comparing the names of types. It is not possible for the type system on the front-end layer to be structural, because the DFP editor does not exactly know the structure of data in the nodes.

- It is **strongly typed** and **explicit** as every port must have its type specified.

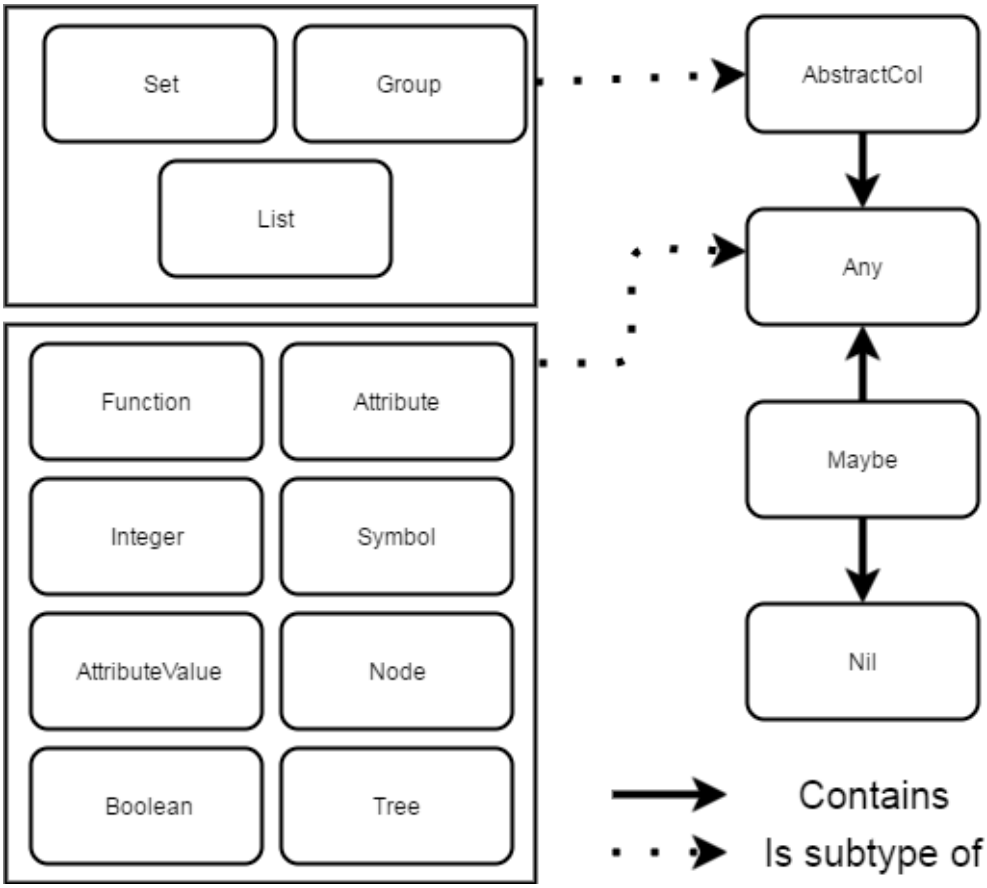
#### 4.1.4.1 Types in MCH-DFP

The designed type system consists of only a few basic types in a simple structure. This list of types can be easily extended as there are no dependencies between them, so if a new node with new types is added to the MCH-DFP interpret, the type system can be easily extended with these types.

There are two types in the top hierarchy of the type system and both of them are abstract: `Any` and `AbstractCol`. Data in the application cannot physically be only of an abstract type, so these types are used only as super types in nodes with polymorphic ports. Another special type related to these two is `Maybe` (and `Nil`). These types and also the concept of polymorphism in MCH-DFP will be described in following sections.

The list of types in MCH-DFP can be found in Table 4.1 and the hierarchy of the types is depicted on Figure 4.1.

Figure 4.1: Overview of types in MCH-DFP.




Source: Author

Table 4.1: Types in MCH-DFP.

| Type name | Type description |
|---|---|
| **Abstract types** | |
| `Any` | General abstract type representing any elementary type. All of the elementary types are subtypes of `Any`. |
| `AbstractCol` | General abstract type representing any collection type. All of the collection types are subtypes of `AbstractCol`. |
| `Maybe` | Special type for handling possible appearance of null values. Can be either `Any` or `Nil`. For more details, see subsubsection 4.1.4.2. |
| `Nil` | Special type representing null value. |
| **Collection types** | |
| `Set` | A collection preserving the order of addition where all elements are unique. |
| `List` | A collection preserving the order of addition. |
| `Group` | Special type of a collection used by `GroupOnCollection` node. |
| **Elementary types** | |
| `Symbol` | Represents a string or a number. |
| `Integer` | Represents an integer value. |
| `Boolean` | Represents a boolean value. |
| `Function` | Represents a function, used e.g. by lambda components in MCH-DFP. |
| `Node` | Represents a node in a tree (e.g. XML node). |
| `Tree` | Represents a tree (e.g. XML tree). |
| `Attribute` | Represents an attribute of a node. |
| `AttributeValue` | Represents a value of an attribute. |

Source: Author

#### 4.1.4.2 Maybe type

It was already mentioned that the original MCH-DFP type system had problems with null values. Many inbuilt nodes of the application had a possibility of returning a null (or invalid) value for some inputs.

An example of such a problematic node is `GetNodeAttribute` with the following description:

> Component returns the attribute of the node with the given key.

The component accepts a *node* of type `Node` as first input and a `Symbol` representing the name of the attribute *key* to be obtained as second input. The component originally returned one output of type `Attribute`.

It is however possible that the supplied *node* does not have any attribute with the given *key* and therefore it has to return a null (or empty) value. However, the empty value can cause problems in the dataflow which the user did not expect, and therefore it was necessary to develop a way of handling these errors.

In subsubsection 2.3.2.4, the Haskell datatype `Maybe` was described, which handles exactly the same situation. Therefore the design of this type was leveraged and type `Maybe` was introduced.

Type `Maybe` represents either a value of specified `Any` type or *empty* `Nil` type.

In the aforementioned example with `GetNodeAttribute` node, the output data type was changed to `Maybe, Attribute` meaning that the node can either return an `Attribute` or a `Nil`.

Introducing the new type is however just one step in resolving this issue. The returned value of type `Maybe` must be somehow processed by the following nodes and therefore a new computational node was developed: `NilResolver`.

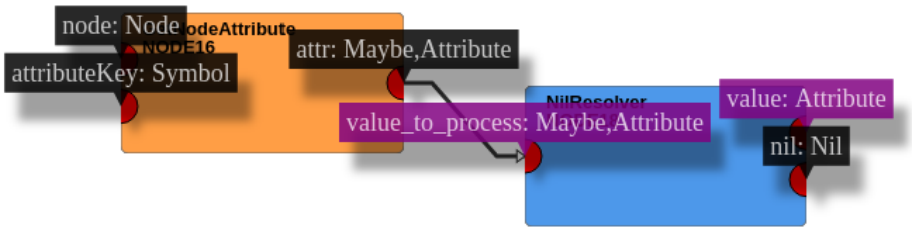
This node accepts a value of `Maybe` data type and based on its actual value, it either writes its `Any` value to first output port, where it can continue in dataflow, or its `Nil` value to the second output port, where it can be properly handled.

This design is depicted on the actual GUI implementation on Figure 4.2, where there is interaction of `GetNodeAttribute` with the `NilResolver` together.

#### 4.1.4.3 Type polymorphism

Type polymorphism was already defined in paragraph 2.2.1.3.2 and this feature was one of the requirements of the designed type system. The requirement appeared mostly to tackle the issue with function repetition, because without type polymorphism there had to be a separate function for each data type, even if this function behaved the same for all of the types.

Figure 4.2: Interaction of `GetNodeAttribute` with `NilResolver` handling the `Maybe` data type. All labels of types are shown for clarity. Purple color marks type bubbling, where the abstract type was changed to a concrete type.




Source: Author

An example of such a situation is a function, which obtains the first element from a collection (such a node in MCH-DFP is called `GetFirstItemFromCollection`). Without abstract types and type polymorphism, a separate function would be needed for `Set, Node`, another for `Set, Tree`, another for `List, Node`, etc. If the requirements were to have these functions for all type combinations, the amount of functions would be enormous.

Thankfully, type polymorphism solves this issue. The input port of `GetFirstItemFromCollection` node is of type `AbstractCol, Any`, which is a supertype of any collection type of any elements. It means, that any node with any collection output port can be connected to this node. The output port of this node has type `Maybe, Any`.

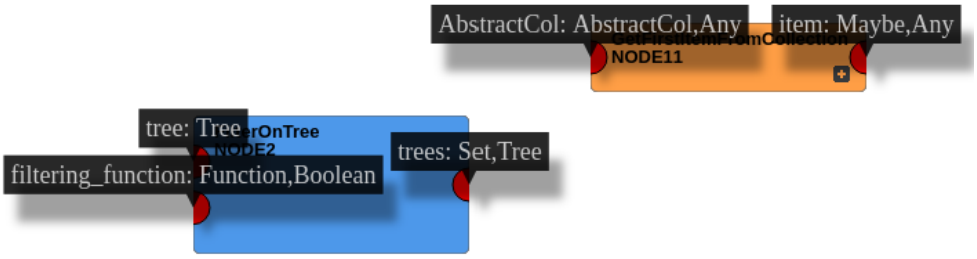
However, when a node with concrete collection type is connected to `GetFirstItemFromCollection` (e.g. node returning `List, Node`), the general abstract type in the node is instantiated with this concrete type. At that moment, it is no longer possible to return a general type, because the input collection contains items of type `Node`. Therefore, the output type of `GetFirstItemFromCollection` is now changed to `Maybe, Node`.

This case is depicted on Figure 4.3 and Figure 4.4.

Type instantiation can actually happen not just on a single node, but also on multiple interconnected nodes.

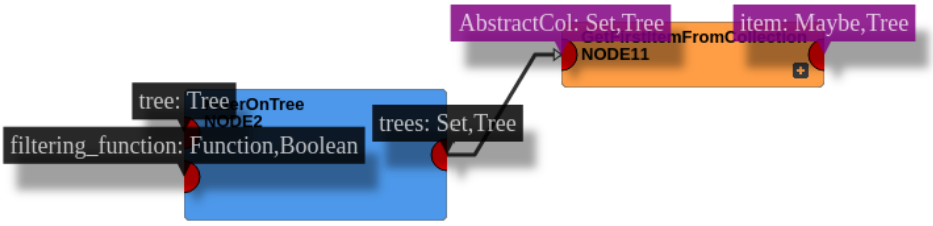
It is important for the MCH-DFP UI to comprehensibly visualize this behaviour, which will be referred to as *type bubbling*. Therefore, the abstract types and types which had their type changed because of type bubbling had to be distinguished from the basic types. UI also has to be able to view which type is currently valid for the given port.

Figure 4.3: Type polymorphic node `GetFirstItemFromCollection` before edge connection. All labels are shown for clarity.




Source: Author

Figure 4.4: Type bubbling on `GetFirstItemFromCollection` after edge connection. All labels are shown for clarity. Purple labels mark bubbled types which were abstract before.




Source: Author

#### 4.1.4.4 Type system elements in the UI

An implementation of several UI elements was needed to fulfil the requirements stated above. These elements are necessary to allow proper functioning of the application according to the designed type system and to help the user understand the types throughout the MCH-DFP.

The designed UI features include:

- Contextual viewing of information about types of ports on graph representation.
- Contextual viewing of information about types of ports in the preferences menu.

- Presenting information about types of ports in the palette of nodes.
- Visualization of type bubbling.
- Prevention of incompatible edge connection between two ports with incompatible types.
- Removal of incompatible edges when another edge is removed and because of type bubbling this edge is suddenly also incompatible

## 4.2 Implementation of type system

The implementation of the type system in MCH-DFP consisted of several parts which will be described in this section.

### 4.2.1 Technologies

The choice of technologies used for the type system implementation was strongly dependant on the original technologies used in MCH-DFP. As the type system is rather a low level part of the whole MCH-DFP DSL, it requires to be inbuilt inside the core system components and therefore the technologies used should be the same.

Because of this reasons, the nodes with ports and their types are still defined in JSON format. These definitions are parsed to JavaScript on back-end for the evaluation of rules. The front-end module is also written in JavaScript, it leverages the jQuery library for viewing types in preferences and KineticJS framework for viewing types directly on visualized graph.

### 4.2.2 Back-end implementation

The MCH-DFP interpret on back-end is written in JavaScript, which is a dynamically typed language and this language therefore does not perform type checking prior to execution. It was therefore decided that the static type system will be placed on front-end layer to assist the user during creation of rules and prevent him from introducing any type errors.

From this reason, the type system has little presence in the back-end layer. The only significant change, which had to made to the back-end, is correct definition of all existing DFP nodes, especially the types of their ports.

Every DFP node in `components.json` was therefore reviewed and its port types were cautiously determined.

### 4.2.3 Front-end implementation

Most of the type system implementation happened on front-end side, in the MCH-DFP editor, and its main components will be described in this section.

#### 4.2.3.1 Type system module in MCH-DFP editor architecture

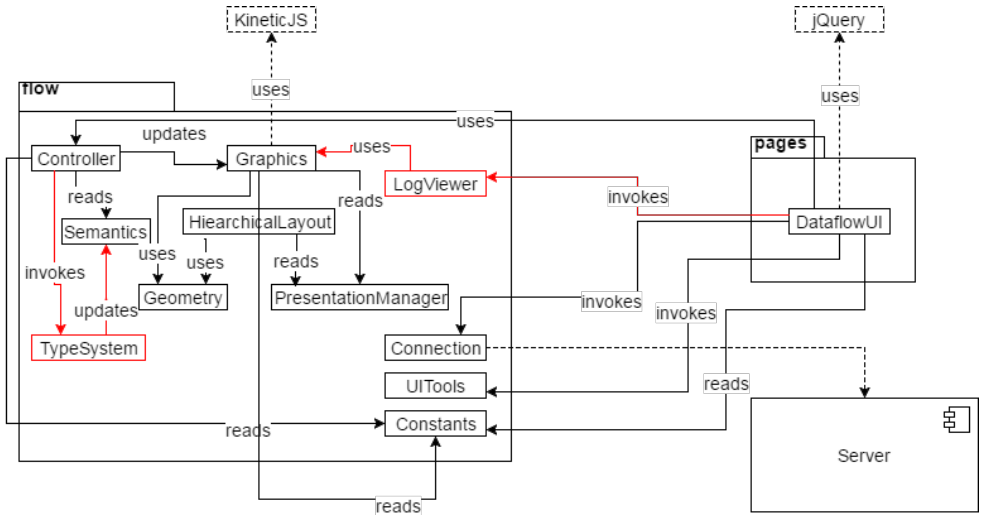
To support features required from the type system, a new front-end module was implemented and integrated to the MCH-DFP editor architecture. The main functionalities of this module are to:

- Offer functionalities to check type compatibility.
- Support type bubbling when a new edge is connected to port with polymorphic type and backward bubbling when such an edge is deleted.

The controller of the application therefore uses functions of this module to determine whether an edge connection between two ports is valid and it also initializes the type bubbling. When it is initiated, the type system module updates the semantic model of the DFP graph with the propagated types.

The integration of the module to the current MCH-DFP editor architecture is depicted on Figure 4.5.

Figure 4.5: Integration of new modules to MCH-DFP editor architecture.




Source: Author

#### 4.2.3.2 Type checking

Type checking is the core functionality of the type system, which determines whether two types are compatible with each other, specifically whether there can be an edge from port of *source type* to port of *target type*.

The function `typesMatch(requested, actual)` has to properly handle compatibility of abstract types and their subtypes and also compatibility of

type `Maybe` and therefore it consist of a sequence of conditionals determining the type compatibility.

Function differentiates between three possible relations between two types:

- Incompatible types
- Purely compatible types
- Maybe compatible types, which are types which may be compatible if the abstraction is removed from them (if the type bubbling is applied.)

As this function is essential for the functioning of the type system, it was also thoroughly tested using unit tests with the majority of the potential type combinations.

#### 4.2.3.3 Type bubbling

Type bubbling handles the propagation of types to ports with abstract types. This is the second most important functionality of the *TypeSystem* module. The type propagation proceeds in several steps:

1. After an edge is connected (or a whole graph is rebuilt from database), all edges with *maybe compatible types* are checked if the type bubbling can be performed on them.
2. If type bubbling is possible on an edge, the target port of the edge is updated to the type of the source port. The original type is still kept, so it can be switched back when an edge is disconnected.
3. If the parent node of currently changed port has some abstract output ports, these ports are also subjected to type bubbling, etc.
4. After one type bubbling process is finished, it continues from step 1 on other *maybe compatible* edges.
5. Incompatible edges are then reported to the user.

The process is very similar for edge deletion, where it essentially proceeds backwards.

#### 4.2.3.4 UI elements

Implementation of UI features specified in 4.1.4.4 was mostly focused on presenting the right information to the user at the right time. All information about types are therefore included to the tooltips about nodes and types are also shown when the user hovers with mouse over an edge or port.

It is important to distinguish between normal types and types propagated through type bubbling, and therefore their hover labels have different colour, so the user can immediately see that the type bubbling process took place.

The type system also prevents a user from connecting two incompatible ports together and it removes an edge, if it suddenly became incompatible.

## 4.3 Demonstration of the type system in practice

The developed type system was used and tested on several real MCH-DFP rules. All of these rules were based on already existing rules in original MCH product and they were translated from XPath and regular expressions into MCH-DFP DSL.

All of these rules represent the real needs of industrial customers, who use the MCH product.

To give the reader better understanding of the type system and its features, one of these rules will be described from the type system point of view.

### 4.3.1 MCH-DFP rule example

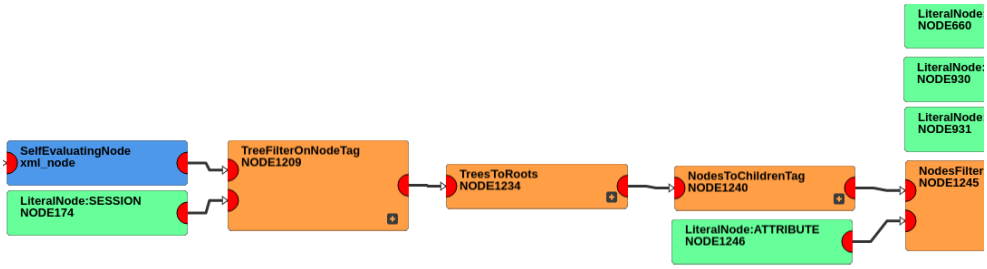
An example rule from the industrial use of MCH-DFP is Commit Interval rule, which checks using an XPath expression whether a certain XML node in the Informatica query has a specified value. The original MCH-DFP rule is not complicated, however it anticipates that the user can work with XPath expressions and for some more complex rules also with regular expressions.

DFP implementation of this rule using nodes, which have ports of types from the developed type system, can be divided into three parts, which are described and demonstrated bellow.

The first part, which can be seen on Figure 4.6, parses the input XML query to a tree and gets all nodes with an `ATTRIBUTE` tag, which are children of a node with a tag `SESSION`. In the second part, which is depicted on Figure 4.7, these nodes are further filtered to obtain only those which have an attribute of `NAME="Commit Interval"` and from these nodes attributes with a name `VALUE` are filtered. In the third part, these attributes are converted to their values and it is checked whether these values are equal to 1000000. The result is afterwards reported.

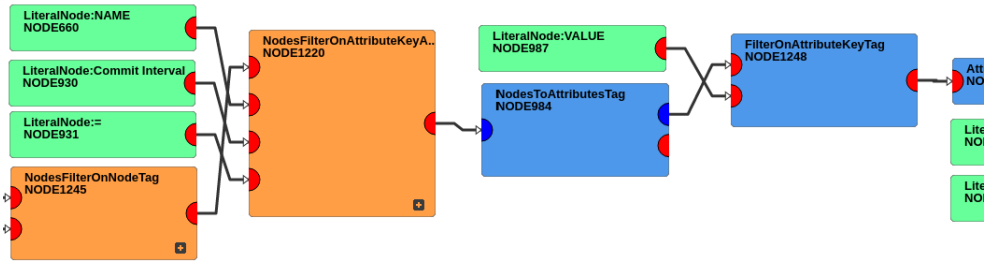
Type system of MCH-DFP offers a user several hints when inspecting and creating such a rule. Firstly, many of the nodes in the rule have ports with abstract types and cascading type bubbling (see Figure 4.9) makes it very easy for the user to understand what is the current type of a port. Secondly, during the rule creation, the user is prevented from connecting incompatible ports and therefore no type error can occur. And finally, the correctly designed type system gives the user another hints about how to use certain nodes. The user can easily understand the functionality of nodes just from their concise names and the types of their ports.

Figure 4.6: Demonstration of DFP Commit Interval rule, first part.



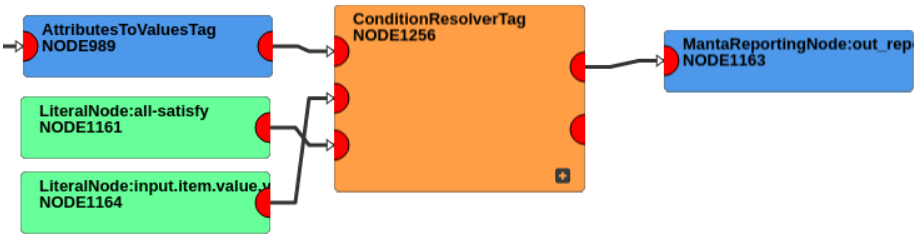

Source: Author

Figure 4.7: Demonstration of DFP Commit Interval rule, second part.



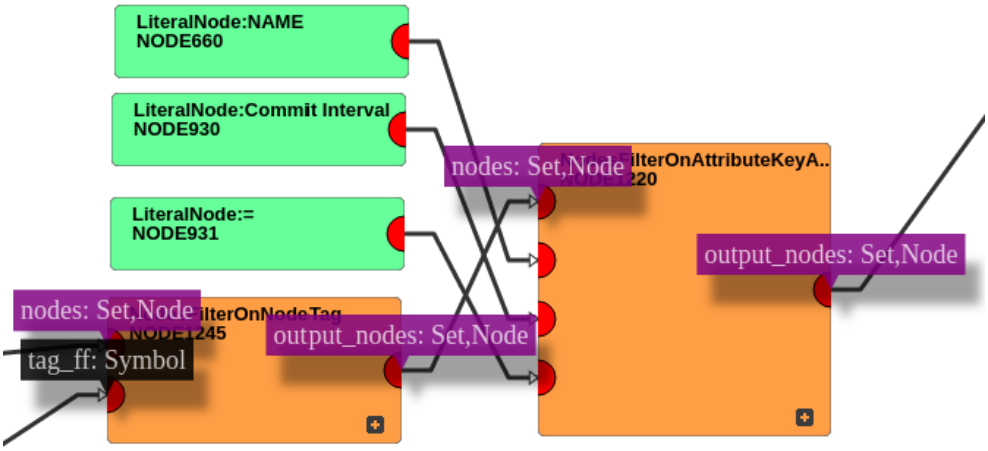

Source: Author

Figure 4.8: Demonstration of DFP Commit Interval rule, third part.




Source: Author

Figure 4.9: Example of cascade bubbling through multiple nodes on Commit Interval node. Multiple labels of types are shown for clarity.




Source: Author

Chapter 5

# Tracing and logging system for MCH-DFP

From the related work about logging and tracing systems (see chapter 3) it can be deduced, that a component with logging and tracing capabilities is a vital part of every computer system, which serves well both to application developers and also its users.

It was also stated that if a logging system for a DFP program aims to be useful and successful, it has to provide means of visualizing the logs directly on the DFP layer. However, the original MCH-DFP interpret and its front-end editor offered no such a functionality, arbitrary logging was only used in both parts on the underlying implementation level to help developers of MCH-DFP. The application offered no means for the MCH-DFP user to trace and examine the execution of the designed DFP rule, other than examining the result itself.

Therefore, one of the main goals of this thesis was to implement a system which would tackle these issues and it will be described in this chapter.

## 5.1 Design of tracing and logging system

### 5.1.1 Requirements

From the state-of-the-art related work and from the discussions with the product owner, several requirements for the tracing and logging module arisen:

The logging and tracing module should:

- collect the information necessary to allow the user examination of the flow in a DFP rule.
- visualize these information on the DFP graph in MCH-DFP editor.
- allow a simple examination of the collected information.

### 5.1.2 Addressing the requirements

The requirements naturally divide the required work into three different parts:

- The first part, collection of information, is basically the logging and tracing itself and it has to be implemented in MCH-DFP interpret. There are two types of information which can be collected:
  - Executions logs – logging the execution of the interconnected nodes.
  - Dataflow logs – logging the data flowing through the nodes.

  For the designed module, the *dataflow logs* were chosen, because user is aware of the order of node execution just from the visualization of the rule. What the user however does not know is what data are send from the executed nodes and he requires this information. Moreover, the order of execution can be also recovered from the *dataflow logs*. These logs need to be properly structured so they can be easily viewed on the front-end.

- The second part, a visualization of the logs on the DFP graph, requires a concise representation of the log entries directly on the designed rule. The following approach for this visualization was chosen:
  1. User obtains the log entries from the interpret.
  2. User switches the rule editor to debug mode with the log entries.
  3. The editor visualizes the basic information from log entries (data type of entry and shortened value) on the graph in a similar manner as it was seen for type visualization.
  4. User can click on this information to examine the data more thoroughly.

- The third part is about simple examination of single log entries on the graph. As stated by [53], it is important that the user can easily switch between levels of details and therefore it is necessary to enable examination even of complex log entries, such as trees. The following approach was chosen:
  - When a user clicks on a single log entry, a new window opens, where he can see more detailed information about the log entry: its type and its value.
  - If the value is more complex type, such as a collection or a tree, the user can examine single items in the collection or single nodes in the tree and dive into more details.

### 5.1.3 Logging and tracing on back-end

The design of the logging and tracing system of *dataflow logs* in the MCH-DFP interpret is composed of two main parts: deciding where the logging of data should happen and designing the structure of the log, so it can be properly visualized.

#### 5.1.3.1 Logging location

Because the log entries should be *dataflow logs*, the logging should happen on places where data flows. In the DFP, ports on nodes are such a place. Therefore the implemented module should be implemented on top of ports and every time the value on a port changes, this value should be written to the log. In the design proposed by this thesis it only logs the latest value on a port (because e.g. values on lambda components can change more than once), however approach logging all changes would be also possible.

#### 5.1.3.2 Log structure

JSON was chosen as a format to represent the logs, as it is both human and computer readable format and it is also a standard in current web applications. Other reasons are similar as for choosing JSON as a format for the DFP rule representation (see [2]). The chosen JSON structure is following:

```
{
  "timestamp": ...,
  "runBy": ...,
  "nodes": [
    {
      "ID": "NODE1",
      "ports": [
        {
          "id": "output_port0",
          "value":"<root>...</root>"
        }
      ],
      "children": []
    },
    ...
  ]
}
```

The `timestamp` field represents when the execution of the DFP rule happened and `runBy` field logs which used executed the program. The most important are the log entries itself under `nodes` field, where each node is logged with its

ports and the value on this ports is recorded. If a node is a composite node, the log entries of its children nodes will be logged under the `children` field.

### 5.1.4 Visualization on the DFP rule

Visualization of the logged information on the DFP rule is done very similarly to viewing types of ports, because the log entries are also related directly to the ports. Visual labels above ports are therefore used for visualizing the entries. It is however important that the user can distinguish between log labels and type labels, so they should have different color. The user should be also able to tell when he entered the debug mode; such a mode should be visually different.

### 5.1.5 Examination of single log entry

After the user clicks on a concise log entry visualization on the rule, a new window should appear with the detailed information about the logged value. This value should be easily examined and therefore proper visualization is required. Simple types, such as `Strings` or `Booleans` can be viewed as they are, however types such as any subtypes of `AbstractCol` or `Trees` are more complicated. Details can be found in Table 5.1.

The collections should be visualized as numerous items and each of this items should be inspectable on its own.

The trees are recursive structures and therefore it should be possible to visualize the different levels the tree and inspect its internal nodes.

To fulfil this additional requirements, a JavaScript library was sought, which would allow similar kind of collection and tree visualization.

#### 5.1.5.1 Visualization library selection

The demanded library should fit into the current technology stack of the front-end and it should be flexible enough to enable both visualizing of collections and trees.

[54] collects several libraries which implement the desired functionality, however after more detailed investigation most of them turned out to be inadequate or too old. jsTree jQuery plugin [55] seemed to be the most recommended and used solution, however it was not flexible enough for MCH-DFP needs, as it would not easily support collection view and also XML tree view. From this reason, Bootstrap Tree View [56] was chosen, as it is also based on jQuery and also on Bootstrap framework, which are both already part of MCH-DFP editor.

Table 5.1: Examination style of different types

| Type name | Type visualisation |
|---|---|
| **Abstract types** | |
| `AbstractCol` and subtypes | Shows all items in the collection, any item can be clicked on and examined on its own depending on its type. |
| `Maybe` | If Nil value occurred, it is printed. Otherwise actual value is shown. |
| **Elementary types** | |
| `Symbol` | The whole string or number is shown. |
| `Integer` | The number is shown. |
| `Boolean` | The boolean value is shown. |
| `Function` | Logger just notes that this value is function, it cannot be examined further. |
| `Node` | Shows node name without its attributes, if it is clicked, it will show the whole node. |
| `Tree` | Shows expandable tree structure, every node in the tree can be clicked on and examined separately. |
| `Attribute` | Shows attribute name and its value. |
| `AttributeValue` | Shows just attribute value. |

Source: Author

## 5.2 Implementation of tracing and logging system

This section will describe the most important implementation remarks about the logging and tracing system and it will present the implemented solution.

### 5.2.1 Technologies

In summary, these technologies were used for the logging and tracing system, in addition to underlying technologies of MCH-DFP (JavaScript for both of its parts):

- JSON for representing the logs.
- KineticJS labels for viewing log entries on DFP graph.
- Bootstrap Tree View library for examination of single log entries.

### 5.2.2 Back-end implementation

The implementation on back-end consisted of one new module, `Logger.js`, which is then injected to other parts of application using RequireJS library.

The module consists of three public methods:

- `init`, which initializes the whole log structure.
- `logNode`, which performs the log action itself and saves the values of ports on referenced node to the log.
- `report`, which converts the internal log structure to JSON.

The main issues with the logger were connected with the fact, that it is written in JavaScript, but actual runtime interpretation of the rules is happening in Java application, where the interpret is compiled as a module using Rhino. The problem was that JavaScript itself is capable of converting any object to JSON, however the Rhino implementation of this capability is sometimes more problematic. The logging procedures therefore had to be adjusted to properly log even after compilation to Java using Rhino.

### 5.2.3 Front-end implementation

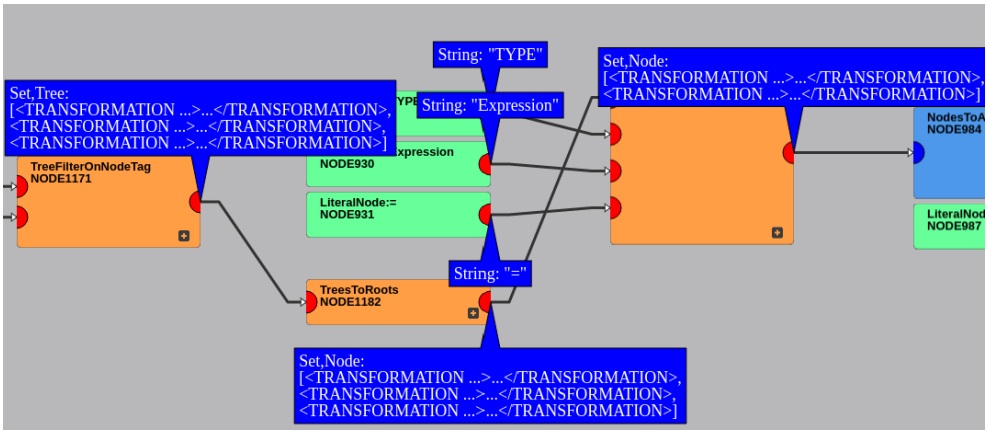
The front-end implementation of the logging system is in the new module called `LogViewer` and its employed to the architecture as seen on Figure 4.5.

#### 5.2.3.1 Visualization of log entries on DFP rule

The visualization of a log can be triggered by clicking "Debug JSON result" button and inserting the execution log. All the log entries are then visualized on respective ports and the whole graph has a darker background so it can be distinguished from normal graph editing.

An example of log visualization can be seen on Figure 5.1.

Figure 5.1: Example of log visualization



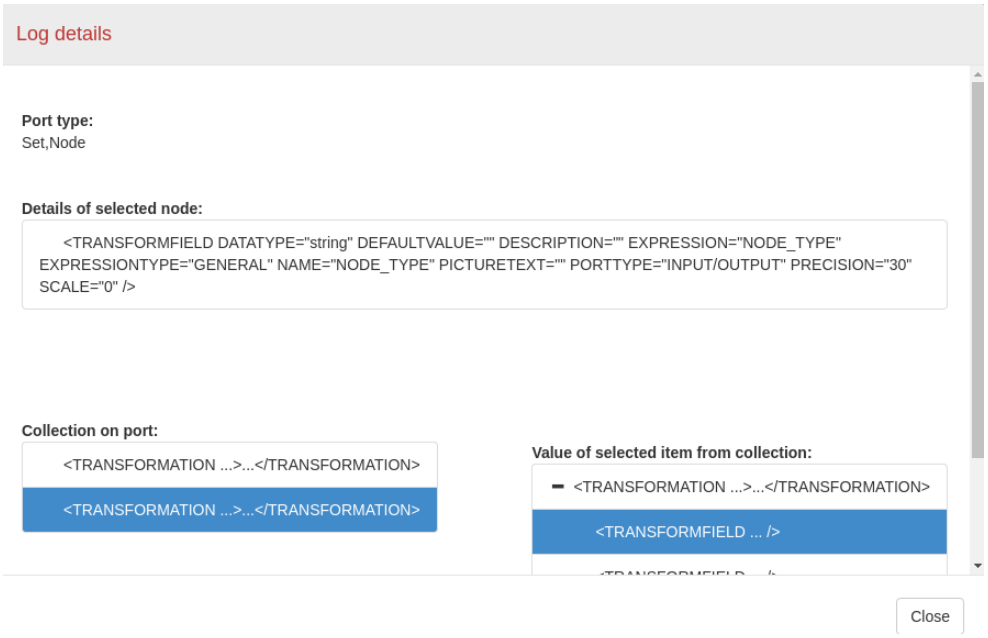

Source: Author

#### 5.2.3.2 Examination of single log entry

Examination of single log entry happens in a pop up window, where the user can properly examine the value on different levels of detail. If the examined entry is not a collection, only the entry is shown and if it is a tree, the user can traverse its recursive structure and view information about its internal nodes. If the entry is a collection, it firstly shows all the items in the list, which can be then examined further.

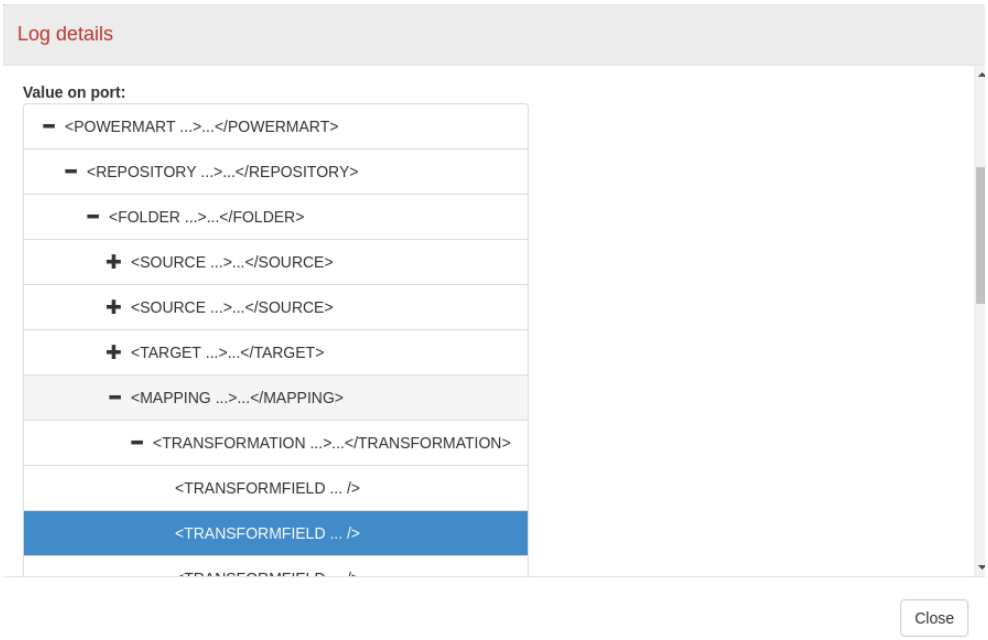
Collection examination is shown on Figure 5.2 and tree examination on Figure 5.3.

Figure 5.2: Example of examination of collection log entry




Source: Author

Figure 5.3: Example of examination of tree log entry




Source: Author

# Conclusion

The goals of this thesis were to advance the already existing system MCH-DFP with two new modules, which bring an extra functionality: a type system for the MCH-DFP domain specific language and means for logging and tracing MCH-DFP rules.

In the first chapter of the thesis, where theoretical part begins, MCH-DFP system was introduced and its important parts with relation to type system and logging were described. The first chapter also presented the summary of underlying theory behind the MCH-DFP system, such as Dataflow programming paradigm and DSL.

The second chapter focused on both formal and informal type systems in programming languages and presented the state-of-the-art research about this problematic. Definitions of types and type systems were given, a possible type system classification was presented and the basics of the type system theory was reviewed. Finally, Hindley-Milner type system was described, following with its implementation – Haskell, which served as an inspiration for the developed type system for MCH-DFP.

The third chapter presented theoretical insights about tracing and logging of computer programs and especially DFP applications. Logging and debugging of DFP programs have not get much research attention so far, thus the main outcome of the state-of-the-art research works shows that it is important to allow a user to visualize the tracing information directly in the DFP interface.

The practical part began with the fourth chapter, in which the designed type system for MCH-DFP was described. Firstly, issues with the original type system were evaluated and based on them requirements for new type system were set. These requirements then served as a basis for designing the static type system, which leverages few notions from Haskell language. The type system should help the user of MCH-DFP to understand which DFP nodes can be connected with each other to form a functioning DFP rule. The usage of the type system was then demonstrated on the real world example

rule from industry.

The last chapter focused on designing the tracing and logging system for MCH-DFP. Requirements were set based on the state-of-the-art research outcomes and three modules of tracing system were designed: a logging module, a module allowing the visualization of a log on the DFP rule and a module allowing examination of a single log entry. The implemented tracing and logging system is a novel approach for debugging DFP applications, as current DFP applications do not leverage visualizing the real dataflows directly on the DFP interface.

Both of the implemented components, type system and tracing system, will make the users' work with MCH-DFP easier, as the type system features offer them with tools which prevent the users to connect incompatible DFP nodes together and the logging and tracing systems allows them to easily track the flow of data in the rule and then debug it, if needed. It is important to remember, that the end users of MCH-DFP are not programmers but rather technical managers who design the business rules and therefore the ease of use is of the essence.

It is possible to further extend the designed and implemented modules. The type system can have more types introduced, when it is needed (even more abstract types if necessary) and it could leverage even more features from Haskell, such as typeclasses. The logging and tracing system might also offer means for examining even the most complex types, such as Functions and it could allow the user to put breakpoints inside the DFP rule and perform step by step execution.

# Bibliography


[1] Czech Technical University in Prague, Profinit, s.r.o., Technology Agency of the Czech Republic. Nástroje pro automatizaci Quality Assurance rozsáhlých Business Intelligence systémů a datových skladů, jan 2013.

[2] Maksimau, A. *Data-Flow programování podnikových pravidel nad databázemi*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, may 2015.

[3] Yazar, T. Design of Dataflow. *Nexus Network Journal*, volume 17, no. 1, 2015: pp. 311–325, ISSN 1522-4600, doi:10.1007/s00004-014-0222-8. Available from: `http://dx.doi.org/10.1007/s00004-014-0222-8`

[4] Myers, B. A. Taxonomies of Visual Programming and Program Visualization. *J. Vis. Lang. Comput.*, volume 1, no. 1, Mar. 1990: pp. 97–123, ISSN 1045-926X, doi:10.1016/S1045-926X(05)80036-9. Available from: `http://dx.doi.org/10.1016/S1045-926X(05)80036-9`

[5] Manta Tools, s.r.o. Manta Checker - automate code reviews, fix errors and enforce policies. feb 2017, [cit. 2017-02-16]. Available from: `https://getmanta.com/manta-checker/`

[6] Westfall, L. Software requirements engineering: what, why, who, when, and how. 2005.

[7] Czech Technical University in Prague, Profinit, s.r.o., Technology Agency of the Czech Republic. Závěrečná zpráva: Jazyk pro zadávání podnikových metodik a algoritmy vyhodnocování pravidel, jan 2014.

[8] Johnston, W. M.; Hanna, J. R. P.; et al. Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, volume 36, no. 1, Mar. 2004: pp. 1–34, ISSN 0360-0300, doi:10.1145/1013208.1013209. Available from: `http://doi.acm.org/10.1145/1013208.1013209`

[9] Hudak, P. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, volume 21, no. 3, Sept. 1989: pp. 359–411, ISSN 0360-0300, doi:10.1145/72551.72554. Available from: http://doi.acm.org/10.1145/72551.72554

[10] Jenkins, K. What Is Functional Programming? dec 2015, [cit. 2017-02-17]. Available from: http://blog.jenkster.com/2015/12/what-is-functional-programming.html

[11] Ward, M. P. Language-oriented programming. *Software-Concepts and Tools*, volume 15, no. 4, 1994: pp. 147–161.

[12] Czech Technical University in Prague, Profinit, s.r.o., Technology Agency of the Czech Republic. Závěrečná zpráva: Nástroj pro syntaktickou a sémantickou analýzu specifických SQL dialektů - Oracle PL/SQL, Microsoft Transact SQL, Teradata BTEQ a jiných procedurál, jan 2017.

[13] Czech Technical University in Prague, Profinit, s.r.o., Technology Agency of the Czech Republic. Závěrečná zpráva: Nástroj pro analýzu XML souborů, jan 2016.

[14] Pierce, B. C. *Types and programming languages*. MIT press, 2002, ISBN 978-0262162098.

[15] Cardelli, L. Type systems. *ACM Computing Surveys*, volume 28, no. 1, 1996: pp. 263–264.

[16] Howe, D. type. dec 2003, [cit. 2017-03-02]. Available from: http://foldoc.org/type

[17] Krishnamurthi, S. Programming Languages: Application and Interpretation. 2003. Available from: http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf

[18] Rémy, D. Type systems for programming languages. *Course notes for the academic year 2014–2015*, jan 2017. Available from: http://gallium.inria.fr/~remy/mpri/cours1.pdf

[19] Jackson, K. *Parallel processing and modular software construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1977, ISBN 978-3-540-37260-8, pp. 436–443, doi:10.1007/BFb0021435. Available from: http://dx.doi.org/10.1007/BFb0021435

[20] BigDingus. Just what is this Javascript object you handed me? 2007.

[21] Backus, J. The History of FORTRAN I, II, and III. *SIGPLAN Not.*, volume 13, no. 8, Aug. 1978: pp. 165–180, ISSN 0362-1340, doi:10.1145/960118.808380. Available from: http://doi.acm.org/10.1145/960118.808380

[22] Bach, J. The Lambda Calculus. may 2012, [cit. 2017-03-08]. Available from: http://palmstroem.blogspot.cz/2012/05/lambda-calculus-for-absolute-dummies.html

[23] Weimer, W. Lecture notes in Design And Analysis Of Programming Languages. 2006. Available from: https://www.cs.virginia.edu/~weimer/2006-655/

[24] Stack Overflow. Developer Survey Results 2016. 2017, [cit. 2017-03-20]. Available from: http://stackoverflow.com/insights/survey/2016

[25] Cass, S. The 2016 Top Programming Languages. 2016, [cit. 2017-03-20]. Available from: http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[26] ISO/IEC. *ISO/IEC 9899 - Programming languages - C*. ISO/IEC, 2011, [cit. 2017-04-14]. Available from: http://www.open-std.org/JTC1/SC22/WG14/www/standards

[27] Oracle. *Java Language and Virtual Machine Specifications*. Oracle, 2015, [cit. 2017-04-14]. Available from: https://docs.oracle.com/javase/specs/

[28] Foundation, P. S. *The Python Language Reference*. Python Software Foundation, 2017, [cit. 2017-04-14]. Available from: https://docs.python.org/2/reference/

[29] International, E. *ECMAScript® 2016 Language Specification*. Ecma International, 2016, [cit. 2017-04-14]. Available from: https://www.ecma-international.org/publications/standards/Ecma-262.htm

[30] group, T. P. *PHP Language Reference*. The PHP group, 2017, [cit. 2017-04-14]. Available from: http://php.net/manual/en/langref.php

[31] ISO/IEC. *Working Draft, Standard for Programming Language C++*. ISO/IEC, 2014, [cit. 2017-04-14]. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

[32] MySQL. *MySQL Technical Specifications*. MySQL, 2017, [cit. 2017-04-14]. Available from: https://www.mysql.com/products/enterprise/techspec.html

[33] Microsoft. *C# Language Specification*. Microsoft, 2015, [cit. 2017-04-14]. Available from: https://msdn.microsoft.com/en-us/library/ms228593.aspx

[34] Hindley, R. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, volume 146, 1969: pp. 29–60, ISSN 00029947. Available from: http://www.jstor.org/stable/1995158

[35] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, volume 17, no. 3, 1978: pp. 348 – 375, ISSN 0022-0000, doi:http://dx.doi.org/10.1016/0022-0000(78)90014-4. Available from: http://www.sciencedirect.com/science/article/pii/0022000078900144

[36] Spiewak, D. What is Hindley-Milner? (and why is it cool?). 2008, [cit. 2017-03-21]. Available from: http://www.codecommit.com/blog/scala/what-is-hindley-milner-and-why-is-it-cool

[37] Gupta, A. K. So You Still Don't Understand Hindley-Milner? 2013, [cit. 2017-03-21]. Available from: http://akgupta.ca/blog/2013/05/14/so-you-still-dont-understand-hindley-milner/

[38] Haskell.org. Haskell Language. 2017, [cit. 2017-03-22]. Available from: https://www.haskell.org/

[39] C2.com. Nominative And Structural Typing. 2010, [cit. 2017-03-22]. Available from: http://wiki.c2.com/?NominativeAndStructuralTyping

[40] Haskell.org. The Haskell 98 Report: Predefined Types and Classes. 2002, [cit. 2017-03-22]. Available from: https://www.haskell.org/onlinereport/basic.html

[41] LearnYourHaskell.com. Types and Typeclasses. [cit. 2017-03-22]. Available from: http://learnyouahaskell.com/types-and-typeclasses

[42] The University of Glasgow 2001. *Haskell 2010 Data.Maybe*. Available from: https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-Maybe.html

[43] Python Software Foundation. Logging HOWTO - Python 3.6.1 Documentation. 2017, [cit. 2017-03-29]. Available from: https://docs.python.org/3/howto/logging.html

[44] Cinque, M.; Cotroneo, D.; et al. Event Logs for the Analysis of Software Failures: A Rule-Based Approach. *IEEE Transactions on Software Engineering*, volume 39, no. 6, June 2013: pp. 806–821, ISSN 0098-5589, doi:10.1109/TSE.2012.67.

[45] Chuvakin, A.; Peterson, G. How to Do Application Logging Right. *IEEE Security Privacy*, volume 8, no. 4, July 2010: pp. 82–85, ISSN 1540-7993, doi:10.1109/MSP.2010.127.

[46] Borghetti, S.; Sgró, A. Dynamic software tracing. Mar. 20 2012, uS Patent 8,140,911. Available from: `https://www.google.com/patents/US8140911`

[47] Reese, R. A. Centralized and decentralized logging mechanisms. 2011, [cit. 2017-04-01]. Available from: `http://softwareengineering.stackexchange.com/questions/95988/centralized-and-decentralized-logging-mechanisms`

[48] Wygodny, S.; Barboy, D.; et al. System and method for remotely analyzing the execution of computer programs. Feb. 10 2000, wO Patent App. PCT/US1999/017,251. Available from: `http://google.com/patents/WO2000007100A1?cl=en`

[49] SoftwareTestingGenius. What are the different approaches to debug the Software Applications. 2011, [cit. 2017-03-30]. Available from: `http://www.softwaretestinggenius.com/what-are-the-different-approaches-to-debug-the-software-applications`

[50] Aizenbud-Reshef, N.; Shaham-Gafni, Y.; et al. Monitoring execution of an hierarchical visual program such as for debugging a message flow. June 8 2004, uS Patent 6,748,583. Available from: `https://www.google.ch/patents/US6748583`

[51] Karam, M. R.; Smedley, T. J.; et al. Unit-level Test Adequacy Criteria for Visual Dataflow Languages and a Testing Methodology. *ACM Trans. Softw. Eng. Methodol.*, volume 18, no. 1, Oct. 2008: pp. 1:1–1:40, ISSN 1049-331X, doi:10.1145/1391984.1391985. Available from: `http://doi.acm.org/10.1145/1391984.1391985`

[52] Karam, M. R.; Smedley, T. J. A testing methodology for a dataflow based visual programming language. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No.01TH8587)*, 2001, pp. 280–287, doi:10.1109/HCC.2001.995275.

[53] Sui, Y.; Pang, L.; et al. Dataflow Visual Programming Language Debugger Supported by Fisheye View. In *2008 The 9th International Conference for Young Computer Scientists*, Nov 2008, pp. 1024–1029, doi:10.1109/ICYCS.2008.77.

[54] jQuery rain. Best jQuery Treeview Plugins & Tutorials with Demo. 2017, [cit. 2017-04-14]. Available from: `http://www.jqueryrain.com/demo/jquery-treeview/`

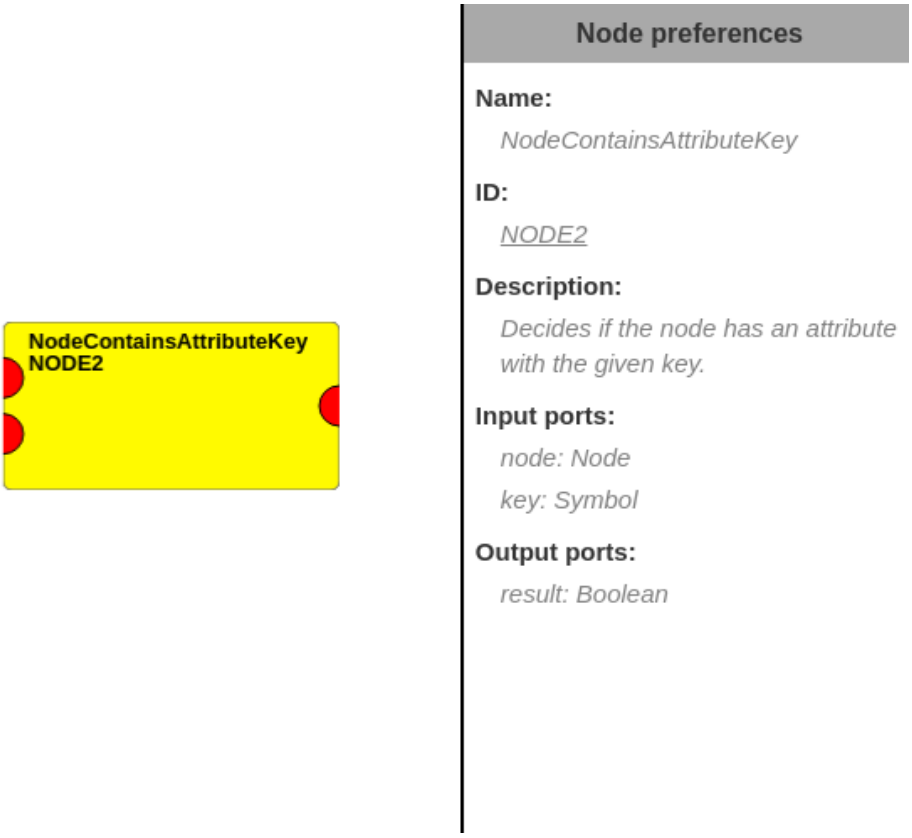
[55] Bozhanov, I. jstree. https://github.com/vakata/jstree, 2017, [cit. 2017-04-14]. Available from: https://github.com/vakata/jstree

[56] Miles, J. Bootstrap Tree View. https://github.com/jonmiles/bootstrap-treeview, 2015, [cit. 2017-04-14]. Available from: https://github.com/jonmiles/bootstrap-treeview

Appendix A

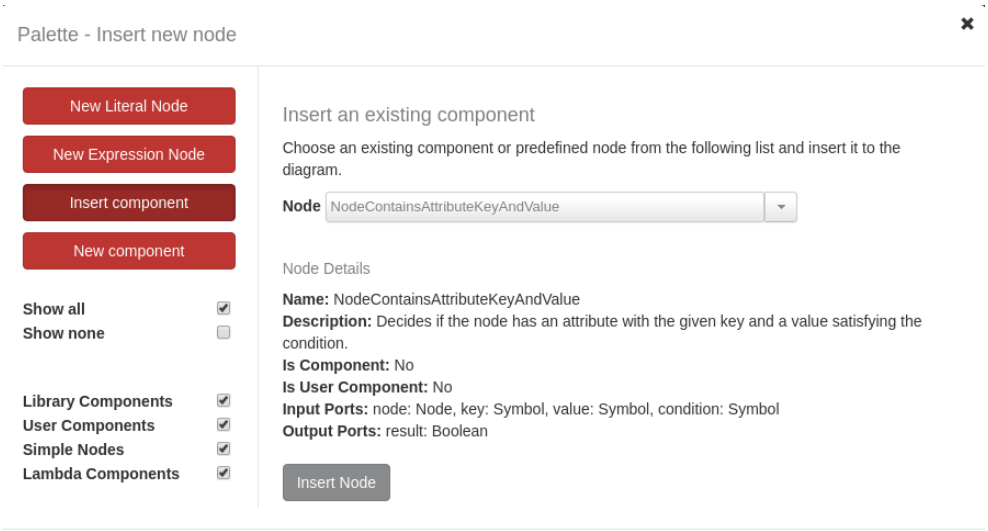
# Examples of MCH-DFP UI

Figure A.1: Basic information about a selected node.



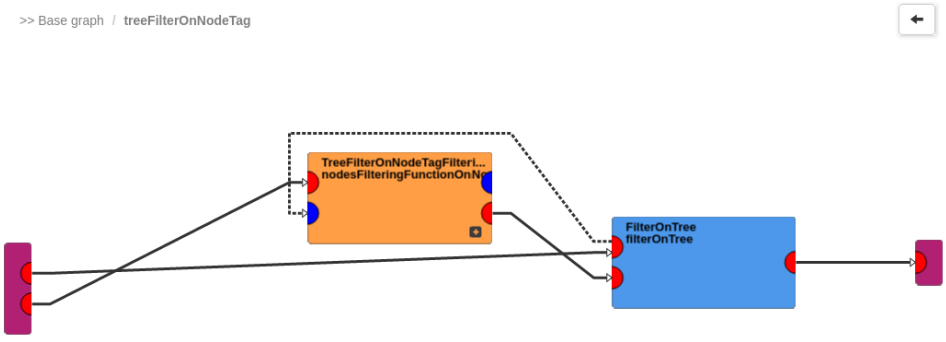

Source: Author

Figure A.2: Palette of nodes enabling inserting of a library node.



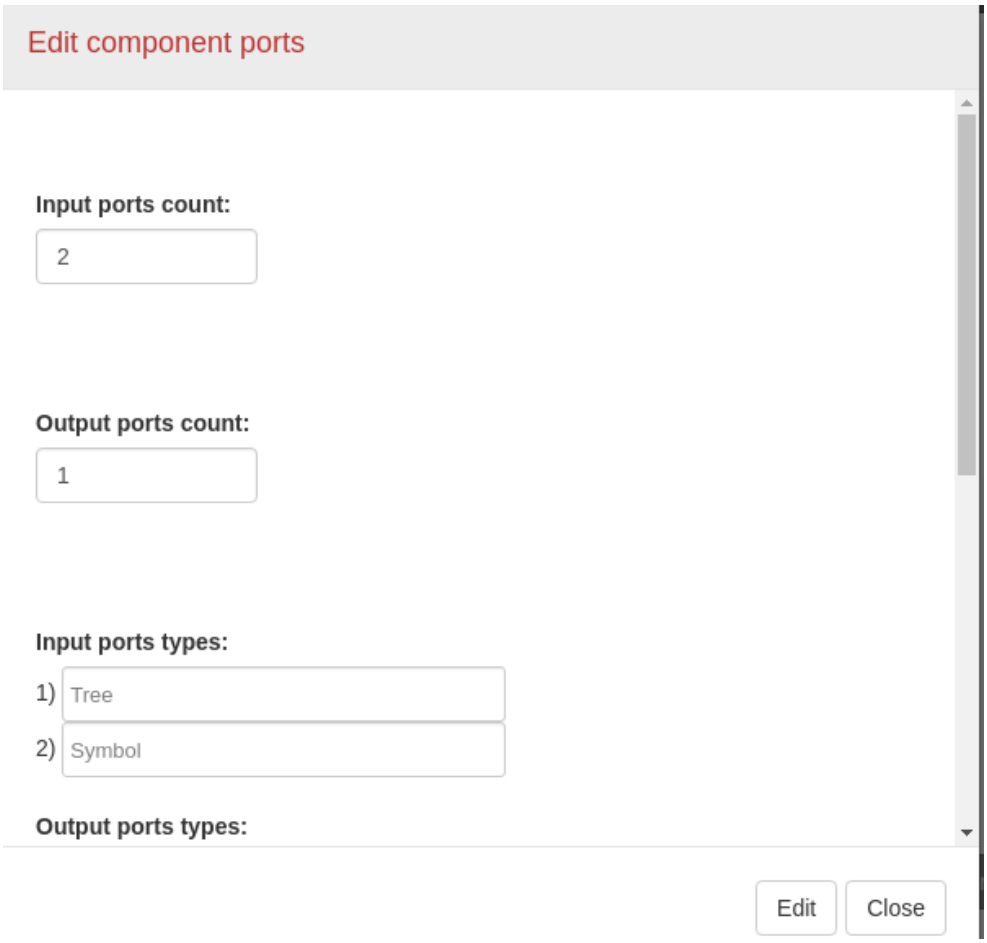

Source: Author

Figure A.3: Viewing inner graph of `TreeFilterOnNodeTag` component.




Source: Author

Figure A.4: Editing structure of component copied from `TreeFilterOnNodeTag`.




Source: Author

Appendix B

# Installation guide

## B.1 Building from sources

Maven tool is required to build the application from sources. The build process requires bulding of several WAR artifacts and bundling them together. Unfortunatelly, many of these artifacts are available only to the employees of Manta Tools. Enclosed CD (see D) contains build script `manta_install.sh` which executes the Maven commands (it has to be edited to contain proper paths to given `pom.xml` files).

## B.2 Deploying the application

After performing the build, you obtain a WAR file. This WAR file is also located in `war` folder on the enclosed CD. This WAR has to be deployed to an application server, e.g. Tomcat 7. Copy this file to the server webapps location (e.g. `/var/lib/tomcat7/webapps`). Restart the application server. Working with DFP editor should be possible right now, however if you want to use other features of Manta Checker, including its DFP interpret, you need to obtain a license from Manta Tools and put it in the `WEB-INF` folder.

## B.3 Running the application

Open the WAR artifact folder in browser on the application server. Default testing account has username `admin` and password `admin`. To add a new Interpret rule, go to Settings, List rules, Add, choose Interpret rule and fill in the details.

To open the DFP editor, open the WAR address with `/rules` suffix. For the first execution, it is necessary to initialize the database from the top menu Manage database. You can either load a Test JSON or start creating your own rule. Some of the test JSONs are located in `src` folder in

`manta-connector-rules-interpret` package in `rule` folder. Some of these files might not be compatible with the type system and need adjustments. To overcome some of these issues with test JSONs, it is better to first save them to the database and then reload them. These JSONs were there purely for tests purposes and user created rules do not suffer with these issues.

The example shown in this thesis is in folder `checker-test` and is called `expressionTransformationName.json`. If you click on Debug JSON result, the text area will already contain a sample log of exuction of this rule, so you can easily use the log features without running the interpret.

Appendix C

# Acronyms

**AOP** Aspect oriented programming

**AST** Abstract syntax tree

**DFP** Dataflow programming

**DSL** Domain specific language

**FIT CTU** Faculty of Information Technology at Czech Technical University in Prague

**GUI** Graphical user interface

**JSON** JavaScript Object Notation

**MCH** Manta Checker

**MCH-DFP** Manta Checker Dataflow Programming module

**PNG** Portable Network Graphics

**TACR** Technology Agency of the Czech Republic

**XML** Extensible markup language

Appendix D

# Contents of enclosed CD

- `readme.txt` ..... the file with brief CD contents description
- `src` ..... the directory of source codes
  - `mch-dfp` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
- `war` ..... the directory with deployable artifact