ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Formalizing Typing Rules for VeriMon

Bachelor Thesis

Nicolas Kaletsch

October 19, 2021

Professor: Prof. Dr. David Basin
Supervisors: Dr. Srđan Krstić, Martin Raszyk

Department of Computer Science, ETH Zürich

**Abstract**

Type systems are lightweight formal methods that can be used to syntactically prove the absence of certain kind of errors in programs. This methodology can also be applied to policy specifications, which express desired system behavior. In this thesis we focus on specifications written in metric first-order dynamic logic (MFODL) for a runtime verification tool called VeriMon. We formalize typing rules for MFODL formulas prove them sound, and present a type inference algorithm that uses the rules. Due to the lack of time we prove the algorithm correct only for a subset of MFODL formulas. Together with the formalized notions of correctness this algorithm paves the way to replace the unverified type checking algorithm used by VeriMon. During the process of formalization we improved MFODL semantics and found bugs in the unverified type checking algorithm.

# Contents

**Contents** **iii**

**1 Introduction** **1**

**2 Preliminaries** **3**
2.1 MFODL . . . 3
2.1.1 Syntax and Semantics . . . 3
2.1.2 Safe formulas . . . 6
2.2 Type Inference in MonPoly . . . 7

**3 Type System** **9**
3.1 Rules . . . 9
3.1.1 Terms . . . 9
3.1.2 Formulas . . . 10
3.2 Generalized Semantics . . . 13
3.3 Soundness . . . 15
3.4 Completeness . . . 18

**4 Type Inference Algorithm** **19**
4.1 Partial type symbols . . . 19
4.2 Terms . . . 20
4.2.1 Algorithm . . . 20
4.2.2 Correctness . . . 22
4.3 Formulas . . . 25
4.3.1 Introducing Type Symbol Functions . . . 25
4.3.2 Algorithm . . . 26
4.3.3 Correctness . . . 28

**5 Discussion** **31**

**6 Conclusion** **33**

**Bibliography** **35**

Chapter 1

# Introduction

The monitoring problem is to decide for a given stream of events and a policy whether the policy holds at all points in the stream. Whenever a violation of the policy occurs, it must be reported. A tool that solves the monitoring problem is called a monitor and it must output all policy violations. The focus of this thesis is a monitor called VeriMon [1]. Policies can be specified in different ways [5]. In VeriMon they are specified in metric first-order dynamic logic (MFODL, Section 2.1).
Since mistakes in programs can often be very costly, monitors can be used for runtime verification. It is clear that then the trustworthiness of the monitor is of great importance. VeriMon is the only monitor which has been proven correct [11, 1] for an expressive language like MFODL.
While using verified monitors solves one problem, arguably a harder problem remains: often the policy itself is incorrectly specified. There are some specifications which are clearly incorrect and undesirable. Consider the case where a policy includes an addition between an integer and a string, as this operation is not meaningfully defined. MFODL allows such policies to be written as its syntax relies on a single sort with a single corresponding carrier set.
In programming languages the absence of such errors can be proven with the help of a type system. We can also apply this methodology for MFODL formulas: A type system consists of rules which specify when a formula does not include such errors. A type inference algorithm (or type checking algorithm) then is a program that checks whether a given formula adheres to the rules of the type system. If this is not the case, the type inference algorithm rejects the formula and notifies the user. Currently VeriMon includes a type inference algorithm (Section 2.2) which is, in contrary to VeriMon's core algorithm, unverified.
The goal of this thesis is to further increase the trustworthiness of VeriMon by expanding its verified core to include a type inference algorithm for MFODL. Specifically, we formalize and sanity check the typing rules of the MFODL's type system, proving them sound (Chapter 3). Along the way we managed to improve the MFODL seman-

[1] https://bitbucket.org/jshs/monpoly/src/master/

tics, by generalizing and correcting a minor flaw. In addition to that, we formalize a type inference algorithm and partially prove its correctness (Chapter 4)[2]. This leads to a bug fix in the unverified type checking algorithm and improvements to MFODL semantics (Chapter 5) and paves the way for a formally verified type checking algorithm for MFODL.

**Related Work** This thesis contributes to VeriMon, a verified version of the MonPoly monitor [2, 3]. The unoptimized core algorithm of VeriMon for metric first-order temporal logic (MFOTL) formulas has been formally verified [11] and then extended to additionally support MFODL formulas [1].
There exist already various Isabelle/HOL formalizations of typing rules and type inference algorithms: Naraschewski and Nipkow [8, 7] formalize type inference rules and an algorithm for Mini-ML, a simply-typed lambda calculus enriched with let-constructors. They prove soundness and completeness of the algorithm with respect to the typing rules. Unlike this thesis, they focus only on the correctness of the algorithm and not the soundness of the type system itself. Such a type soundness proof for a ML-like language is presented by Boite and Dubois [4]. They formalize a language, which is an extension of Mini-ML and present a soundness proof using the progress and preservation approach [10]. Both of these papers talk cover typing in functional programming languages, while we consider typing formulas.

**Acknowledgment** I would like to thank my supervisors Dr. Srđan Krstić and Martin Raszyk for their great support. Their inputs and guidance proved very helpful throughout the thesis, in both practical and theoretical aspects.

[2]Complete formalization: https://github.com/cc23/MFODL-Typing/

Chapter 2

# Preliminaries

In this chapter we first give a brief overview of metric first-order dynamic logic (MFODL). Then we describe the existing unverified type check algorithm for MFODL implemented in MonPoly.

## 2.1 MFODL

The syntax and semantics of MFODL formulas, as well as the rest of the concepts in this thesis are presented using Isabelle/HOL [9] syntax. In some cases we mildly depart from Isabelle syntax for the sake of readability. Isabelle/HOL is a proof assistant, i.e. it mechanically checks human written proofs. Since all proofs must pass through the small, well-understood kernel, Isabelle proofs are very trustworthy. For instance, the core algorithm of VeriMon has been proven correct in Isabelle/HOL [11].

Metric first-order dynamic logic (MFODL) is a language extending metric first-order temporal logic (MFOTL) with regular expressions [1]. MFOTL includes past and future temporal operators such as *Since* and *Until*, as well as the usual first-order logic operators such as the existential operator and predicates.
Extended versions of MFOTL feature complex functional terms, non-recursive let operators and aggregation formulas similar to those in SQL. The formula supports sum, average, median, count, minimum and maximum [2]. MFODL which we consider here inherits all these MFOTL expansions.

### 2.1.1 Syntax and Semantics

We now present a slightly modified version of MFODL's syntax and semantics [1].

**datatype** $data = \mathsf{Int}\ int \mid \mathsf{Flt}\ double \mid \mathsf{Str}\ string$
**type_synonym** $db = (string \times data\ list)\ set$
**type_synonym** $ts = nat$

**typedef** $trace = \{s :: (db \times ts)\ stream.\ \mathsf{trace}\ s\}$
**typedef** $\mathcal{I} = \{(a :: nat, b :: enat).\ a \leq b\}$
**datatype** $trm = \mathsf{V}\ nat \mid \mathsf{C}\ data \mid -trm \mid trm + trm$
$\mid\ trm - trm \mid trm \cdot trm \mid trm\ /\ trm \mid trm\ \%\ trm \mid \mathsf{i2f}(trm) \mid \mathsf{f2i}(trm)$
**datatype** $frm = string(trm\ list) \mid \mathsf{let}\ string = frm\ \mathsf{in}\ frm \mid trm \approx trm$
$\mid\ trm \prec trm \mid trm \preceq trm \mid \neg frm \mid frm \vee frm \mid frm \wedge frm \mid \bigwedge(frm\ list) \mid \exists frm$
$\mid\ nat \leftarrow (agg\_type, data)\ trm;\ nat\ frm \mid \bullet_{\mathcal{I}} frm \mid \bigcirc_{\mathcal{I}} frm$
$\mid frm\ \mathsf{S}_{\mathcal{I}}\ frm \mid frm\ \mathsf{U}_{\mathcal{I}}\ frm \mid \rhd_{\mathcal{I}}\ (frm\ re) \mid \blacktriangleleft_{\mathcal{I}}\ (frm\ re)$
**datatype** $'a\ re = \star^{nat} \mid 'a? \mid 'a\ re + 'a\ re \mid 'a\ re \cdot 'a\ re \mid ('a\ re)^*$

**fun** etrm :: $data\ list \Rightarrow trm \Rightarrow data$ **where**
$\quad \mathsf{etrm}\ v\ (\mathsf{V}\ x) = v\,!\,x$
$\mid \mathsf{etrm}\ v\ (\mathsf{C}\ x) = x$
$\mid \mathsf{etrm}\ v\ (t_1 + t_2) = \mathsf{etrm}\ v\ t_1 + \mathsf{etrm}\ v\ t_2$
$\mid \mathsf{etrm}\ v\ (t_1 - t_2) = \mathsf{etrm}\ v\ t_1 - \mathsf{etrm}\ v\ t_2$
$\mid \mathsf{etrm}\ v\ (-t) = -\mathsf{etrm}\ v\ t$
$\mid \mathsf{etrm}\ v\ (t_1 * t_2) = \mathsf{etrm}\ v\ t_1 * \mathsf{etrm}\ v\ t_2$
$\mid \mathsf{etrm}\ v\ (t_1 / t_2) = \mathsf{etrm}\ v\ t_1\ \mathbf{div}\ \mathsf{etrm}\ v\ t_2$
$\mid \mathsf{etrm}\ v\ (t_1 \% t_2) = \mathsf{etrm}\ v\ t_1\ \mathbf{mod}\ \mathsf{etrm}\ v\ t_2$
$\mid \mathsf{etrm}\ v\ \mathsf{f2i}(x) = \mathsf{Int}\ (\mathsf{int\_of\_data}\ (\mathsf{etrm}\ v\ x))$
$\mid \mathsf{etrm}\ v\ \mathsf{i2f}(x) = \mathsf{Flt}\ (\mathsf{double\_of\_data}\ (\mathsf{etrm}\ v\ x))$
**fun** sat :: $trace \Rightarrow data\ list \Rightarrow nat \Rightarrow frm \Rightarrow bool$ **where**
$\quad \mathsf{sat}\ \sigma\ V\ v\ i\ (r(ts)) = (\mathbf{case}\ V\ r\ \mathbf{of}$
$\qquad \mathsf{None} \Rightarrow ((r, \mathsf{map}\ (\mathsf{etrm}\ v)\ ts) \in \Gamma\ \sigma\ i)$
$\qquad \mid \mathsf{Some}\ X \Rightarrow \mathsf{map}\ (\mathsf{etrm}\ v)\ ts \in X\ i)$
$\quad \mathsf{sat}\ \sigma\ V\ v\ i\ (\mathsf{let}\ p = \varphi\ \mathsf{in}\ \psi) =$
$\qquad \mathsf{sat}\ \sigma\ (V(p \mapsto \lambda i.\ \{v.\ \mathsf{length}\ v = \mathsf{nfv}\ \varphi \wedge \mathsf{sat}\ \sigma\ V\ v\ i\ \varphi\}))\ v\ i\ \psi$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (t_1 \approx t_2) = (\mathsf{etrm}\ v\ t_1 = \mathsf{etrm}\ v\ t_2)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (t_1 \prec t_2) = (\mathsf{etrm}\ v\ t_1 < \mathsf{etrm}\ v\ t_2)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (t_1 \preceq t_2) = (\mathsf{etrm}\ v\ t_1 \leq \mathsf{etrm}\ v\ t_2)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\neg\varphi) = (\neg \mathsf{sat}\ \sigma\ v\ i\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\exists\varphi) = (\exists z.\ \mathsf{sat}\ \sigma\ V\ (z \# v)\ i\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha \vee \beta) = (\mathsf{sat}\ \sigma\ V\ v\ i\ \alpha \vee \mathsf{sat}\ \sigma\ V\ v\ i\ \beta)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha \wedge \beta) = (\mathsf{sat}\ \sigma\ V\ v\ i\ \alpha \wedge \mathsf{sat}\ \sigma\ V\ v\ i\ \beta)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\bigwedge l) = (\forall \varphi \in \mathsf{set}\ l.\ \mathsf{sat}\ \sigma\ V\ v\ i\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\bullet_I \varphi) = (\mathbf{case}\ i\ \mathbf{of}\ 0 \Rightarrow \mathsf{False} \mid j+1 \Rightarrow \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge$
$\qquad \mathsf{sat}\ \sigma\ V\ v\ j\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\bigcirc_I \varphi) = (\mathsf{T}\ \sigma\ (i+1) - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ V\ v\ (i+1)\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha\ \mathsf{S}_I\ \beta) = (\exists j \leq i.\ \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ V\ v\ j\ \beta \wedge$
$\qquad (\forall k \in \{j <.. i\}.\ \mathsf{sat}\ \sigma\ v\ k\ \alpha))$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha\ \mathsf{U}_I\ \beta) = (\exists j \geq i.\ \mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ V\ v\ j\ \beta \wedge$
$\qquad (\forall k \in \{i ..< j\}.\ \mathsf{sat}\ \sigma\ V\ v\ k\ \alpha))$

$$
\begin{aligned}
&\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (y \leftarrow (\Omega, d)\ t; b.\ \varphi) = \big(\textbf{let}\ M = \{(x, \mathsf{ecard}\ Z) \mid x\ Z.\\
&\qquad Z = \{z.\ \mathsf{length}\ z = b \wedge \mathsf{sat}\ \sigma\ (z\ @\ v)\ i\ \varphi \wedge \mathsf{etrm}\ (z\ @\ v)\ t = x\} \wedge Z \neq \{\}\}\\
&\quad \textbf{in}\ (M = \{\} \longrightarrow \mathsf{fv}\ \varphi \subseteq \{0\ ..{<}\ b\}) \wedge v\,!\,y = \mathsf{agg\_op}\ (\Omega, d)\ M)\\
&\mid \mathsf{sat}\ \sigma\ v\ i\ (\blacktriangleleft_I\ r) = (\exists j \leq i.\ \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge (j, i) \in \mathsf{match}\ (\mathsf{sat}\ \sigma\ v)\ r)\\
&\mid \mathsf{sat}\ \sigma\ v\ i\ (\triangleright_I\ r) = (\exists j \geq i.\ \mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge (i, j) \in \mathsf{match}\ (\mathsf{sat}\ \sigma\ v)\ r)
\end{aligned}
$$

Domain of values in MFODL is a union of three sorts: integers, floating point numbers and strings (type *data*). Events that can occur have a name (type *string*) and a list of parameters of type *data*. Events that occur at the same time are grouped into a database (type *db*). The semantics are defined with respect to a *trace*, which is an infinite stream of databases and corresponding time-stamps $ts$. The $i$th database in the trace $\sigma$ is denoted as $\Gamma\ \sigma\ i$, while the $i$-th time-stamp as $\mathsf{T}\ \sigma\ i$. In the type definition of *trace*, the function trace makes sure that the time-stamps are monotonically and eventually strictly increasing. MFODL terms *trm* include variables (constructor V), using de Bruijn indeces. Every variable is thus represented as a natural number. Constants (constructor C) can be of any type *data*. In examples we will use standard notation for constants and variables, i.e. omit the constructors and use variable names ($x$, $y$,...). Furthermore, the terms include usual arithmetic operations, as well as conversions from integer to floating point numbers (i2f()) and vice-versa (f2i()).
The semantics of terms is defined by the function etrm which takes as input a list of data $v$. This serves as a variable assignment: the variable with index $i$ gets assigned value $v\,!\,i$. The function etrm evaluates a term $t$ with respect to a variable assignment $v$ by recursive calls and calls to functions over data (e.g. $+$ over *data*). There are cases where these functions are not meaningful. For example, an addition of a string and an integer. In these cases, to have total function, arbitrary choices had to be made. The same holds for the conversion functions which are used for i2f() and f2i(). We point out some of these choices:

$$
\begin{aligned}
&\mathsf{String}\ s \star x = \mathsf{Flt}\ \mathsf{nan}, \qquad \star \in \{+, -, *, /, \%\}\\
&\mathsf{Flt}\ a\ \%\ \mathsf{Flt}\ b = \mathsf{Flt}\ \mathsf{nan}\\
&\mathsf{int\_of\_data}\ (\mathsf{Str}\ s) = 0\\
&\mathsf{double\_of\_data}\ (\mathsf{Str}\ s) = \mathsf{nan}
\end{aligned}
$$

MFODL formulas *frm* include the above mentioned first-order logic constructs, as well as temporal operators, regular expressions *frm re* and aggregations. The set of free variables is denoted fv $\varphi$ (fv_trm $t$ for terms) and nfv $\varphi$ is the lowest natural number, higher than all free variables. The semantics of MFODL formulas is defined by the function sat, which for a given trace $\sigma$, variable assignment $v$, time-point $i$ and a formula $\varphi$ returns True iff $\varphi$ is satisfied by $v$ on $i$ in $\sigma$. As an additional argument sat takes a partial function $V$ which maps a predicate name $p$ introduced by a let ... in construct to the set of all assignments by which $p$ is satisfied on a given time-point.
When interpreting the relations between terms another arbitrary choice has been made: $\mathsf{Int}\ i \prec \mathsf{Flt}\ f \prec \mathsf{Str}\ s$. The aggregation formula $y \leftarrow\ (\Omega, d)\ t;\ b\ \varphi$ binds $b$ variables in

the subformula $\varphi$ and uses the remaining free variables of $\varphi$ for grouping. The variable $y$ gets assigned the result of the aggregation over term $t$. The aggregation operator consists of one of CNT, MIN, MAX, SUM, AVG, MED and a default value $d$, for empty groups. The sat function creates a multiset $M$ in the case of an aggregation, which then gets evaluated by the function agg_op.

```
fun agg_op :: agg_type × data ⇒ (data × enat) set ⇒ data where
  agg_op (CNT, y) M = Int (integer_of_int (length (flatten_multiset M)))
| agg_op (MIN, y) M = (case flatten_multiset M of
        [] ⇒ y
      | x#xs ⇒ foldl min x xs
| agg_op (MIN, y) M = (case flatten_multiset M of
        [] ⇒ y
      | x#xs ⇒ foldl max x xs
| agg_op (SUM, y) M = foldl plus y (flatten_multiset M)
| agg_op (AVG, y) M = Flt (let xs = flatten_multiset M in case xs of
        | [] ⇒ 0
        | _,⇒ double_of_data (foldl plus (Int 0) xs) / double_of_int (length xs))
| agg_op (MED, y) M = Flt (let xs = flatten_multiset M; u = length xs in
      if u = 0 then 0 else
        let u' = u div 2 in
          if even u then (double_of_data (xs ! (u' − 1)) +
            double_of_data (xs ! u') / double_of_int 2)
          else double_of_data (xs ! u'))
```

The function agg_op takes as input the aggregation type and a default value as well as the multiset $M$. This multiset is first flattened into a list of type *data list* and then based on the aggregation type the aggregate value is computed. The function sat then checks that this value equals the value assigned to $y$ by $v$. For most cases in agg_op the computation is a simple fold over the list.

### 2.1.2 Safe formulas

VeriMon uses finite tables to represent valuations of formulas, these are computed using standard table operations (e.g. join, union). Therefore, it supports only formulas which do not produce infinitely many satisfying valuations. This is only a fragment of all MFODL formulas. For example the formula $\neg A(x)$ has infinitely many satisfying valuations. Formulas which do not produce infinitely many satisfying valuations are called monitorable, or safe [11]. The safe_formula function formalizes this notion.

```
fun safe_formula :: frm ⇒ bool where
  safe_formula (r(ts)) = (∀t ∈ set ts. is_Var t ∨ is_Const t)
  safe_formula (let p = φ in ψ) =
      ({0.. < nfv φ} ⊆ fv φ ∧ safe_formula φ ∧ safe_formula ψ)
```

$$
\begin{aligned}
&| \ \text{safe\_formula}\ (t_1 \approx t_2) = (\text{is\_Const}\ t_1 \wedge (\text{is\_Const}\ t_2 \vee \text{is\_Var}\ t_2) \vee \\
&\qquad \text{is\_Var}\ t_1 \wedge \text{is\_Const}\ t_2) \\
&| \ \text{safe\_formula}\ (t_1 \prec t_2) = \mathit{False} \\
&| \ \text{safe\_formula}\ (t_1 \preceq t_2) = \mathit{False} \\
&| \ \text{safe\_formula}\ (\neg (V\ x \approx V\ y)) = (x = y) \\
&| \ \text{safe\_formula}\ (\neg \varphi) = (\text{fv}\ \varphi = \{\} \wedge \text{safe\_formula}\ \varphi) \\
&| \ \text{safe\_formula}\ (\exists \varphi) = \text{safe\_formula}\ \varphi \\
&| \ \text{safe\_formula}\ (\alpha \vee \beta) = (\text{fv}\ \alpha = \text{fv}\ \beta \wedge \text{safe\_formula}\ \alpha \wedge \text{safe\_formula}\ \beta) \\
&| \ \text{safe\_formula}\ (\alpha \wedge \beta) = (\text{safe\_formula}\ \alpha \wedge \\
&\qquad (\text{safe\_assignment}\ (\text{fv}\ \alpha)\ \beta \vee \text{safe\_formula}\ \beta \vee \\
&\qquad \text{fv}\ \beta \subseteq \text{fv}\ \alpha \wedge (\text{is\_constraint}\ \beta \vee \\
&\qquad (\textbf{case}\ \beta\ \textbf{of}\ \neg \beta' \Rightarrow \text{safe\_formula}\ \beta' \mid \_ \Rightarrow \text{False})))) \\
&| \ \text{safe\_formula}\ (\bigwedge l) = (\textbf{let}\ (pos, neg) = \text{partition safe\_formula}\ l\ \textbf{in}\ pos\ \neq [] \wedge \\
&\qquad \text{list\_all safe\_formula}\ (\text{map remove\_neg neg})\ \wedge \\
&\qquad \bigcup(\text{set}\ (\text{map fv neg})) \subseteq \bigcup(\text{set}\ (\text{map fv pos}))) \\
&| \ \text{safe\_formula}\ (\bullet_I \varphi) = \text{safe\_formula}\ \varphi \\
&| \ \text{safe\_formula}\ (\bigcirc_I \varphi) = \text{safe\_formula}\ \varphi \\
&| \ \text{safe\_formula}\ (\alpha\ \mathsf{S}_I \beta) = (\text{fv}\ \alpha \subseteq \text{fv}\ \beta \wedge (\text{safe\_formula}\ \alpha \vee (\textbf{case}\ \alpha\ \textbf{of}\ \neg \alpha' \Rightarrow \\
&\qquad \text{safe\_formula}\ \alpha' \mid \_ \Rightarrow \text{False})) \wedge \text{safe\_formula}\ \beta) \\
&| \ \text{safe\_formula}\ (\alpha\ \mathsf{U}_I \beta) = (\text{fv}\ \alpha \subseteq \text{fv}\ \beta \wedge (\text{safe\_formula}\ \alpha \vee (\textbf{case}\ \alpha\ \textbf{of}\ \neg \alpha' \Rightarrow \\
&\qquad \text{safe\_formula}\ \alpha' \mid \_ \Rightarrow \text{False})) \wedge \text{safe\_formula}\ \beta) \\
&| \ \text{safe\_formula}\ (y \leftarrow (\Omega, d)\ t; b.\ \varphi) = (\text{safe\_formula}\ \varphi \wedge y + b \notin \text{fv}\ \varphi \wedge \{0.. < b\} \subseteq \\
&\qquad \text{fv}\ \varphi \wedge \text{fv\_trm}\ f \subseteq \text{fv}\ \varphi) \\
&| \ \text{safe\_formula}\ (\blacklozenge_I\ r) = \text{safe\_regex fv}\ (\lambda g\ \varphi.\ \text{safe\_formula}\ \varphi \vee \\
&\qquad (g = \text{Lax} \wedge (\textbf{case}\ \varphi\ \textbf{of}\ \neg \varphi' \Rightarrow \text{safe\_formula}\ \varphi' \mid \_ \Rightarrow \text{False}))\ \text{Past Strict}\ r \\
&| \ \text{safe\_formula}\ (\blacksquare_I\ r) = \text{safe\_regex fv}\ (\lambda g\ \varphi.\ \text{safe\_formula}\ \varphi \vee \\
&\qquad (g = \text{Lax} \wedge (\textbf{case}\ \varphi\ \textbf{of}\ \neg \varphi' \Rightarrow \text{safe\_formula}\ \varphi' \mid \_ \Rightarrow \text{False}))\ \text{Futu Strict}\ r
\end{aligned}
$$

A disjunction of two formulas, for example is only monitorable iff both subformulas are monitorable and they the same free variables. The for every formula $\varphi$ function is_constraint $\varphi$ returns true iff $\varphi$ is a relation between terms ($\approx$, $\prec$, $\preceq$) or the negation of a relation (e.g. $\neg(t_1 \approx t_2)$). The predicate safe_assignment $(\text{fv}\ \alpha)\ \beta$ for two formulas $\alpha$ and $\beta$ is satisfied iff $\beta$ is an equality between a variable $x$ not free in $\alpha$ and a term $t$, whose variables are all free in $\alpha$. With these additional predicates, safety of a conjunction is defined. The function safe_regex which determines safety of formula regular expressions is omitted.

## 2.2 Type Inference in MonPoly

The existing type inference algorithm [1] is implemented in OCaml. It consists of two main functions: type_check_term and type_check_formula. The former type checks a term: as an input it takes a signature, which specifies the types for the arguments of

[1] https://bitbucket.org/jshs/monpoly/src/master/src/rewriting.ml

predicates, an environment which maps each free-variable to a type it can have, the expected type of the term and the term to be typed. If a type clash exists (e.g. a string in an addition) the function fails. Otherwise, the function returns the inferred type of the term with a updated signature and environment.
The latter function is used for formulas and it operates similarly: as an input it takes a signature, an environment and the formula to be type checked. It returns an updated signature and an environment or it reports a failure.

Chapter 3

# Type System

In this chapter we describe the type system that we used. First, we present the formalized typing rules for MFODL. We then discuss the changes in the MFODL's semantics that we opted to implement after formalizing the type system. Finally, we present two different theorems proving the soundness of safe MFODL formulas and quickly address why there is no such theorem for completeness.

## 3.1 Rules

In the following, we discuss and describe the formalized typing rules for MFODL terms and formulas. Ihe well-typedness of both terms and formulas is formalized as an inductive predicate. This is done using the inductive definition in Isabelle. We specify the typing rules and Isabelle then computes the least predicate closed under the given rules.
For notational simplicity, we depart from Isabelle syntax and present the typing rules using the Gentzen-style notation for inference rules.

### 3.1.1 Terms

In order to specify a type system, we first define a type for each sort of the VeriMon syntax (the *ty* datatype), a function that maps each constant to its type (the ty_of function), as well as a type environment, which maps variables to types. Recall, that VeriMon uses de Bruijn indices, and therefore variables are represented by natural numbers. We group integer and float types as numeric types.

**datatype** *ty* = TInt | TFloat | TString
**fun** ty_of :: *data* ⇒ *ty* **where**
  ty_of Int _ = TInt
| ty_of Flt _ = TFloat
| ty_of Str _ = TString
**type_synonym** *tyenv* = *nat* ⇒ *ty*
**definition** numeric_ty = {TInt, TFloat}

$$\frac{E \vdash E\ x = \tau}{E \vdash v :: \tau}\ \textsc{Var} \qquad \frac{\mathsf{ty\_of}\ c = \tau}{E \vdash c :: \tau}\ \textsc{Const}$$

$$\frac{E \vdash t :: \tau \qquad \tau \in \mathsf{numeric\_ty}}{E \vdash -t :: \tau}\ \textsc{UMinus}$$

$$\frac{E \vdash t_1 :: \tau \qquad E \vdash t_2 :: \tau \qquad \tau \in \mathsf{numeric\_ty}}{E \vdash t_1 \oplus t_2 :: \tau}\ \textsc{BinOp}\ \ \oplus \in \{+, -, *, /\}$$

$$\frac{E \vdash t_1 :: \mathsf{TInt} \qquad E \vdash t_2 :: \mathsf{TInt}}{E \vdash t_1 \% t_2 :: \mathsf{TInt}}\ \textsc{Mod}$$

$$\frac{E \vdash t :: \mathsf{TFloat}}{E \vdash \mathsf{f2i}(t) :: \mathsf{TInt}}\ \textsc{f2i} \qquad \frac{E \vdash t :: \mathsf{TInt}}{E \vdash \mathsf{i2f}(t) :: \mathsf{TFloat}}\ \textsc{i2f}$$

**Figure 3.1:** Typing rules for MFODL terms

With these definitions we then formalize the typing rules for terms (Figure 3.1). We use the notation $E \vdash t :: \tau$ that states that a term $t$ is well-typed with respect to the type environment $E$ and has type $\tau$.

The typing rules for both MFODL terms and for formulas are based on an unpublished draft by Joshua Schneider and Srđan Krstić [6]. The rules are mostly the same, only presented using a different notation, closer to the Isabelle formalization. For example, we use the introduced definitions and group the constant cases. The only difference, which significantly changes the typing rules for terms, is that modulo is only defined for integers and not for every numeric type.

### 3.1.2 Formulas

Using the typing rules for MFODL terms, we can now present the typing rules for MFODL formulas.

We need the additional concept of a signature, which maps predicate names to list of types.

**type_synonym** *sig* = *string* ⇀ *ty list*

If for a signature $S$ and a predicate name $r$, $S\ p = \mathsf{Some}\ tys$ then the predicate $p$ expects arguments of the types $tys$ in this order. Since the *frm* datatype uses de Bruijn indices, bound variables have no names outside of the scope of their corresponding quantifiers. Hence, the types cannot be stored in the type environment. We change the *frm* datatype, to a polymorphic datatype. The typing rules will be defined with respect to *ty frm*, where existential quantifiers and aggregations carry the type of the introduced bound variables.

**datatype** $'t\ frm = string(trm\ list) \mid \mathsf{let}\ string = frm\ \mathsf{in}\ ('t\ frm) \mid trm \approx trm$
$\mid\ trm \prec trm \mid trm \preceq trm \mid \neg('t\ frm) \mid ('t\ frm) \vee ('t\ frm)$
$\mid\ ('t\ frm) \wedge ('t\ frm) \mid \bigwedge(('t\ frm)\ list) \mid \exists\ 't.\ ('t\ frm)$
$\mid\ nat \leftarrow\ (agg\_type, data)\ trm;\ 't\ list\ ('t\ frm) \mid \bullet_{\mathcal{I}}('t\ frm) \mid \circ_{\mathcal{I}}('t\ frm)$
$\mid ('t\ frm)\ \mathsf{S}_{\mathcal{I}}\ ('t\ frm) \mid ('t\ frm)\ \mathsf{U}_{\mathcal{I}}\ ('t\ frm) \mid \triangleright_{\mathcal{I}}\ ('t\ frm) \mid \blacktriangleleft_{\mathcal{I}}\ ('t\ frm)$

The places where arguments of type $'t$, have been introduced are highlighted.
Note that the forth argument of the Agg constructor was previously a natural number indicating how many variables the aggregations binds. With the definition of $'t\ frm$ it now is a list of objects of type $'t$, one for each bound variable. If we substitute the natural number by the length of the list the semantics remains unchanged. Thus $()\ frm$ with this changes corresponds exactly to the monomorphic *frm* datatype.

Due to the use of de Bruijn indices, all variable names have to be incremented when encountering a quantifier or aggregation operator. For example, variable $i \geq 0$ becomes variable $i+1$ in the scope of a new existential quantifier, and the variable bound by the quantifier will be 0. In order to keep track of this, we use case_nat, predefined in Isabelle's Nat theory. This function takes as arguments an object $t$ and a function $E$ mapping from *nat* to the type of object $t$. It then returns a function that maps 0 to $t$ and any other *nat* $n$ to $E\ (n+1)$. For aggregations, where more than one bound variable can be introduced, we create an analogous definition.

**definition** agg_env :: $tyenv \Rightarrow ty\ list \Rightarrow tyenv$ **where**
agg_env $E\ tys = (\lambda z.$ **if** $z <$ length $tys$ **then** $tys\,!\,z$ **else** $E\ (z -$ length $tys))$

We write $S,\ E \vdash \varphi$, if formula $\varphi$ is well-typed with respect to signature $S$ and type environment $E$.
Figure 3.2 shows the formalized typing rules for formulas. As mentioned above, the rules are similar to [6]. However, the draft includes typing rules with regular expressions whereas in the formalization we omit them. We simply say that a match operator formula of a regular expression $r$ is well-typed if and only if every formula in $r$ is well-typed. Additionally, we also make sure that the default value for aggregations is of the right type.

$$\frac{E \vdash t_1 :: \tau \qquad E \vdash t_2 :: \tau}{S,\ E \vdash t_1 \bowtie t_2}\ \textsc{Rel}\quad \bowtie \in \{\approx, \prec, \preceq\}$$

$$\frac{S\ p = \mathsf{Some}\ [\tau_1, \ldots, \tau_n] \qquad E \vdash t_1 :: \tau_1 \qquad \ldots \qquad E \vdash t_n :: \tau_n}{S,\ E \vdash p(t_1, \ldots, t_n)}\ \textsc{Pred}$$

$$\frac{S,\ E \vdash \varphi}{S,\ E \vdash \star\varphi}\ \textsc{UnFma}\quad \star \in \{\neg, \bullet_I, \bigcirc_I\} \qquad \frac{S,\ E \vdash \varphi_1 \qquad S,\ E \vdash \varphi_2}{S,\ E \vdash \varphi_1 \star \varphi_2}\ \textsc{BinFma}\quad \star \in \{\wedge, \vee, \mathsf{S}_I, \mathsf{U}_I\}$$

$$\frac{\forall \varphi \in \mathsf{atms}\ r.\ S,\ E \vdash \varphi}{S,\ E \vdash \star r}\ \textsc{MatchRex}\quad \star \in \{\triangleright_I, \blacktriangleleft_I\} \qquad \frac{S,\ \mathsf{case\_nat}\ \tau\ E\ \vdash\ \varphi}{S,\ E \vdash \exists \tau.\ \varphi}\ \textsc{Exists}$$

$$\frac{S,\ E'' \vdash\ \varphi_1 \qquad S\,(p \mapsto\ \mathsf{Some}\ [E''\ 0, \ldots,\ E''\ n]),\ E \vdash \varphi_2}{S,\ E \vdash \mathsf{let}\ p = \varphi_1\ \mathsf{in}\ \varphi_2}\ \textsc{Let ... in}\quad n \text{ is highest free variable in } \varphi$$

$$\frac{E\ y = \tau \qquad E' \vdash\ t :: \tau \qquad S,\ E' \vdash\ \varphi \qquad \tau \in \mathsf{numeric\_ty} \qquad \mathsf{ty\_of}\ d = \tau}{S,\ E \vdash y \leftarrow\ (\mathsf{SUM}, d)\ t;\ tys\ \varphi}\ \textsc{Sum}$$

$$\frac{E\ y = \mathsf{TInt} \qquad E' \vdash\ t :: \tau \qquad S,\ E' \vdash\ \varphi \qquad \mathsf{ty\_of}\ d = \mathsf{TInt}}{S,\ E \vdash y \leftarrow\ (\mathsf{CNT}, d)\ t;\ tys\ \varphi}\ \textsc{Cnt}$$

$$\frac{E\ y = \mathsf{TFloat} \qquad E' \vdash\ t :: \tau \qquad S,\ E' \vdash\ \varphi \qquad \tau \in \mathsf{numeric\_ty} \qquad \mathsf{ty\_of}\ d = \mathsf{TFloat}}{S,\ E \vdash y \leftarrow\ (A, d)\ t;\ tys\ \varphi}\ \textsc{AvgMed}\quad A \in \{\mathsf{AVG}, \mathsf{MED}\}$$

$$\frac{E\ y = \tau \qquad E' \vdash\ t :: \tau \qquad S,\ E' \vdash\ \varphi \qquad \mathsf{ty\_of}\ d = \tau}{S,\ E \vdash y \leftarrow\ (A, d)\ t;\ tys\ \varphi}\ \textsc{MinMax}\quad A \in \{\mathsf{MIN}, \mathsf{MAX}\}$$

$$\text{where } E' = \mathsf{agg\_env}\ E\ tys$$

**Figure 3.2:** Typing rules for MFODL formulas

```
fun less_eq_data where
  Int x ≤ Int y ⟷ x ≤ y
| Flt x ≤ Flt y ⟷ x ≤ y
| Str x ≤ Str y ⟷ x ≤ y
| (_ :: data) ≤ _ ⟷ undefined

fun uminus_data where
  − Int x = Int (−x)
| − Flt x = Flt (−x)
| − (_ :: data) = undefined

fun plus_data where
  Int x + Int y = Int (x + y)
  Flt x + Flt y = Flt (x + y)
| (_ :: data) + _ = undefined

fun modulo_data where
  Int x mod Int y = Int (mod_to_zero x)
| (_ :: data) mod _ = undefined

primrec int_of_data :: data ⇒
    integer where
  int_of_data Int _ = undefined
| int_of_data Flt x =
      integer_of_double x
| int_of_data Str _ = undefined

primrec double_of_data :: data ⇒
    double where
  double_of_data Int _ =
      double_of_integer x
  double_of_data Flt _ = undefined
| double_of_data Str _ = undefined
```

**Figure 3.3:** Generalized Semantics, where minus_data $(-)$, times_data $(*)$ and divide_data (**div**) are defined similar to plus_data $(+)$

## 3.2 Generalized Semantics

As pointed out in Chapter 2 some arbitrary choices had been made for the evaluation of terms in VeriMon. Now that we have a clear definition of which formulas are well-typed and which are not, we can remove those arbitrary choices. Instead we leave these cases undefined.

Intuitively, if a formula is well-typed with respect to our rules, then the semantics from Chapter 2 should be the same as our generalized the semantics.
In Section 3.3 we will present a theorem which gives us this guarantee.

Another place where we changed the semantics was in the function *agg_op*. We discovered that the default values are not handled consistently across different aggregations. For example, the sum of an empty table is defined to be the default value whereas in the average case the result will be 0, regardless of its default value.
There are cases where this leads to unexpected results, significantly when the default value does not coincide with the neutral element of the aggregation operation. For example, the sum aggregation would always add the default value to the sum. If its default value were different from 0 this would produce unintuitive results.
In addition to this, we found a bug in agg_op (see Section 2.1). In the median case, if flatten_muiltiset $M$ is a list of even length, the average of the two middle elements agg_op was incorrectly calculated (further discussed in Chapter 5).

```
primrec flt_of_data_agg :: data ⇒ double where
  double_of_data_agg (Int x) = double_of_integer x
| double_of_data_agg (Flt x) = x
| double_of_data_agg (Str _) = undefined
fun agg_op :: agg_type × data ⇒ (data × enat) set ⇒ data where
  agg_op (CNT, y) M = (case (flatten_multiset M, finite_multiset M) of
        (_, False) ⇒ y
      | ([], _) ⇒ y
      | (xs, _) ⇒ Int (integer_of_int (length xs))
| agg_op (MIN, y) M = (case (flatten_multiset M, finite_multiset M) of
        (_, False) ⇒ y
      | ([], _) ⇒ y
      | (x#xs, _) ⇒ foldl min x xs
| agg_op (MAX, y) M = (case (flatten_multiset M, finite_multiset M) of
        (_, False) ⇒ y
      | ([], _) ⇒ y
      | (x#xs, _) ⇒ foldl max x xs
| agg_op (SUM, y) M = (case (flatten_multiset M, finite_multiset M) of
        (_, False) ⇒ y
      | ([], _) ⇒ y
      | (x#xs, _) ⇒ foldl (+) x xs
| agg_op (AVG, y) M = (case (flatten_multiset M, finite_multiset M) of
        (_, False) ⇒ y
      | ([], _) ⇒ y
      | (x#xs, _) ⇒ Flt (double_of_data_agg (foldl (+) x xs) /
            double_of_int (length (x#xs))))
| agg_op (MED, y) M = (case (flatten_multiset M, finite_multiset M) of
        (_, False) ⇒ y
      | ([], _) ⇒ y
      | (xs, _) ⇒ Flt (let len = length xs; mid = len div 2 in
            if even len then
                  (double_of_data_agg (xs ! (mid − 1)) +
                double_of_data_agg (xs ! mid)) / double_of_int 2
            else double_of_data_agg(xs ! mid))
```

We corrected the bug in the median case and changed the function such that the default values are handled consistently across the different aggregations: They are only used if the multiset $M$ is infinite or empty. At the moment the default values for aggregations are fixed (for the sum aggregation it is 0). Thus, this problem did not lead to unexpected behavior. However our changes improve consistency. Also, it is conceivable to later give the user the opportunity to specify a default value. Unlike previous semantics, our new semantics would support such a case.
Furthermore, we added the new function double_of_data_agg, which is only used in

agg_op for average and median cases. This function must be used as the new version of double_of_data, since double_of_data is undefined for floats. For the function f2i() this change is intentional. However, in the average and median cases of agg_op we support aggregations over floats.

## 3.3 Soundness

We present two theorems, proving different notions of soundness of our type system. The main soundness theorem guarantees us, as mentioned above, that the generalized semantics are the same as the one in Chapter 2 for well-typed safe formulas.
We create a locale in Isabelle where we define new functions sat′, etrm′, agg_op′ which are similar to sat, etrm, agg_op except for the functions from Figure 3.3. Every such function we replace with a new function (e.g. undef_plus for $(+)$) which we leave undefined. We only assume that it agrees with the functions from Figure 3.3 on meaningful inputs, i.e. the ones for which we did not change the function. For example, undef_plus must be equal to $(+)$, if either both arguments are integers or both are floats.

**locale** sat_general = **fixes**
undef_plus :: *data* ⇒ *data* ⇒ *data* **and**
undef_minus :: *data* ⇒ *data* ⇒ *data* **and**
undef_times :: *data* ⇒ *data* ⇒ *data* **and**
undef_divide :: *data* ⇒ *data* ⇒ *data* **and**
undef_modulo :: *data* ⇒ *data* ⇒ *data* **and**
undef_double_of_data :: *data* ⇒ *double* **and**
undef_double_of_data_agg :: *data* ⇒ *double* **and**
undef_int_of_data :: *data* ⇒ *integer* **and**
undef_less_eq :: *data* ⇒ *data* ⇒ *bool*

**assumes** undef_plus_sound : $\forall x\ y.$ undef_plus (Int x) (Int y) = Int x + Int y
$\forall x\ y.$ undef_plus (Flt x) (Flt y) = Flt x − Flt y
**assumes** undef_minus_sound : $\forall x\ y.$ undef_minus (Int x) (Int y) = Int x − Int y
$\forall x\ y.$ undef_minus (Flt x) (Flt y) = Flt x − Flt y
**assumes** undef_uminus_sound : $\forall x.$ undef_uminus (Int x) = −Int x
$\forall x.$ undef_uminus (Flt x) = −Flt x
**assumes** undef_times_sound : $\forall x\ y.$ undef_times (Int x) (Int y) = Int x ∗ Int y
$\forall x\ y.$ undef_times (Flt x) (Flt y) = Flt x ∗ Flt y
**assumes** undef_divide_sound : $\forall x\ y.$ undef_divide (Int x) (Int y) = Int x **div** Int y
$\forall x\ y.$ undef_divide (Flt x) (Flt y) = Flt x **div** Flt y
**assumes** undef_double_of_data_sound : $\forall x.$ undef_double_of_data (Int $x$) =
double_of_data (Int $x$)
**assumes** undef_double_of_data_agg_sound :
$\forall x.$ undef_double_of_data_agg (Int $x$) = double_of_data_agg (Int $x$)
$\forall x.$ undef_double_of_data_agg (Flt $x$) = double_of_data_agg (Flt $x$)

**assumes** undef_int_of_data_sound : $\forall x.$ undef_int_of_data (Flt $x$) = int_of_data (Flt $x$)
**assumes** undef_less_eq_sound : $\forall x\ y.$ undef_less_eq (Int $x$) (Int $y$) $\longleftrightarrow$ Int $x \leq$ Int $y$
$\forall x\ y.$ undef_less_eq (Flt $x$) (Flt $y$) $\longleftrightarrow$ Flt $x \leq$ Flt $y$
$\forall x\ y.$ undef_less_eq (Str $x$) (Str $y$) $\longleftrightarrow$ Str $x \leq$ Str $y$

sat′ and etrm′ are defined the same way as seen in Chapter 2, with the undefined functions instead of the defined ones. agg_op′ is defined the same way as agg_op, but double_of_data_agg is replaced by undef_double_of_data_agg.
Except for the assumptions above, the fixed functions are unknown.
The idea is, to show that sat′ is the same as sat for a well-typed formula, under some more assumptions. This gives us the guarantee, that our changes in semantics do not change how well-typed formulas are evaluated. It even shows that it does not matter at all how the ground functions like plus or minus are defined, as long as they satisfy the above stated assumptions.

Before we state the theorem, we present another soundness theorem, which is then needed in the proof of the main theorem.
This soundness theorem states that under some more assumptions, the satisfying assignment of a well-typed formula assigns to a free variable only data of the right type with respect to the type environment.
One assumption that we need is that the signature coincides with the argument $V$ of the sat function. This means that all possible parameters, which satisfy a given predicate are of the same type, the signature assigns them. We introduce a Isabelle definition wty_envs describing this fact.

**definition** wty_event :: *sig* $\Rightarrow$ *string* $\Rightarrow$ *data list* $\Rightarrow$ *bool* **where**
wty_event $S\ \sigma\ xs \longleftrightarrow$ (**case** $S\ \ p$ **of**
Some $ts \Rightarrow$ list_all2 ($\lambda t\ x.$ ty_of $x = t$) $ts\ xs$ | None $\Rightarrow$ False

**definition** wty_envs :: *sig* $\Rightarrow$ *trace* $\Rightarrow$ (*string* $\rightharpoonup$ *nat* $\Rightarrow$ *data list set*) $\Rightarrow$ *bool* **where**
wty_envs $S\ \sigma\ V \longleftrightarrow$ ($\forall i.$
$(\forall (p, xs) \in \mathsf{T}\ \sigma\ i.\ p \notin \text{dom}\ V \longrightarrow$ wty_event $S\ p\ xs) \wedge$
$(\forall p \in \text{dom}\ V.\ \forall xs \in \text{the}\ (V\ p)\ i.$ wty_event $S\ p\ xs)$)

We can now formulate the theorem.

**Assumption 3.1** *$\varphi$ is well-typed with respect to $S$ and $E$:* $S, E \vdash \varphi$

**Assumption 3.2** *$\varphi$ is satisfied by variable assignment $v$ and predicate assignment $V$, with respect to trace $\sigma$ at time-point $i$:* sat′ $\sigma\ V\ v\ i\ \varphi$

**Assumption 3.3** *$x$ occurs free in $\varphi$:* $x \in$ fv $\varphi$

**Assumption 3.4** *$\varphi$ is a safe formula:* safe_formula $\varphi$

**Assumption 3.5** *Signature $S$ coincides with $V$:* $\mathsf{wty\_envs}\ S\ \sigma\ V$

**Assumption 3.6** *$v$ assigns a value to every free variable in $\varphi$:* $\mathsf{nfv}\ \varphi \leq \mathsf{length}\ v$

**Theorem 3.7** *If assumptions 3.1 – 3.6 hold, then the type of the value assigned to $x$ by $v$ is of the same type as assigned to $x$ by $E$:*

$$\mathsf{ty\_of}\ (v\,!\,x) = E\ x$$

Note that the definition of function safe_formula in assumption 3.4 is not exactly the same as presented in Section 2.1.2. We changed it slightly without loss of generality. The exact changes and reasons for it are discussed in Chapter 5.
Theorem 3.7 is proven in Isabelle by a computational induction over the structure of the safe_formula function. We will not show it here. Instead we will argue why the additional assumptions 3.4 to 3.6 are needed to prove the theorem.
The first three are the assumptions on which the theorem is based on. Thus, we will only look at why the others are needed.
If we did not restrict formula $\varphi$ to be a safe_formula, the claim would not hold. One counterexample we can find if $\varphi$ is $x \approx y$. Let us then choose $E = (\lambda n.\ \mathsf{TFloat})$ and $v\,!\,x = v\,!\,y = \mathsf{Int}\ 42$. All the assumptions would be satisfied but yet, $\mathsf{ty\_of}\ (v\,!\,x) = \mathsf{TInt} \neq \mathsf{TFloat} = E\ x$. Therefore, assumption 3.4 is needed.
Let us now consider assumption 3.5. If $\mathsf{wty\_envs}\ S\ \sigma\ V$ did not hold, we could choose $\varphi$ to be a predicate $p(x)$. Now say $S\ p = \mathsf{Some}\ [\mathsf{TString}]$ , but $V\ p = \mathsf{Some}\ (\lambda i.\ \{[42]\})$. Then for $v$ with $v\,!\,x = \mathsf{Int}\ 42$, $sat\ \sigma\ V\ v\ i\ p(x)$ holds and $S,\ E \vdash \varphi$, for $E$ with $Ex = \mathsf{TString}$ as well. Thus, all assumptions are satisfied, but $\mathsf{ty\_of}\ (v\,!\,x) = \mathsf{TInt} \neq \mathsf{TString} = E\ x$
Clearly, assumption 3.6 is required, since if this was not the case, the assignment list $v$ could be shorter than the number of free variables in $\varphi$. Then $v\,!\,x$ might not be well-defined. For example, if $v$ was the empty list $[\,]$.
It is now easy to show that Theorem 3.7 also holds if we have the original sat instead of the generalized $\mathsf{sat}'$ in assumption 3.2. Since it can be shown that sat is equal to $\mathsf{sat}'$ where the undefined functions are instantiated by the original ones. Then $\mathsf{sat}\ \sigma\ V\ v\ i\ \varphi$ follows from $\mathsf{sat}'\ \sigma\ V\ v\ i\ \varphi$. This gives us then our first notion of soundness for our type system.

Let us now show the main soundness theorem. We show the assumptions, then the claim proven and eventually motivate why the assumptions are needed.

**Assumption 3.8** *$\varphi$ is well-typed with respect to $S$ and $E$:* $S,\ E \vdash \varphi$

**Assumption 3.9** *$\varphi$ is a safe formula:* $\mathsf{safe\_formula}\ \varphi$

**Assumption 3.10** *Signature $S$ coincides with $V$:* $\mathsf{wty\_envs}\ S\ \sigma\ V$

**Assumption 3.11** *$v$ assigns a value to every free variable in $\varphi$:* $\mathsf{nfv}\ \varphi \leq \mathsf{length}\ v$

**Theorem 3.12** *If assumptions 3.8 to 3.11 hold, then*

$$\mathsf{sat}\ \sigma\ V\ v\ i\ \varphi \longleftrightarrow \mathsf{sat}'\ \sigma\ V\ v\ i\ \varphi$$

This theorem is again proven by an induction over the structure of safe_formula and using amongst others Theorem 3.7. Note that it was first proven with an additional assumption $\forall y \in \mathsf{fv}\ \varphi.\ \mathsf{ty\_of}\ (v\,!\,y) = E\ y$. We then proved that this assumption was not needed. Since, because of the assumptions 3.8– 3.11, $\forall y \in \mathsf{fv}\ \varphi.\ \mathsf{ty\_of}\ (v\,!\,y) = E\ y$ follows from Theorem 3.7.
All assumptions are needed such that one can use Theorem 3.7. We give one example why safe_formula is necessary: Define $\varphi = V\ x \preceq V\ y$, $E = (\lambda x.\ \mathsf{TInt})$ and some $v$ with $v\,!\,x = \mathsf{Int}\ 42$ and $v\,!\,y = \mathsf{Str}$ "*str*". We end up in a case where we must compare an integer with a string. For these arguments, we do not have enough information about undef_less_eq. We cannot conclude that $\mathsf{sat}\ \sigma\ V\ v\ i\ \varphi \longleftrightarrow \mathsf{sat}'\ \sigma\ V\ v\ i\ \varphi$ in this case.

## 3.4 Completeness

We quickly motivate why a theorem for completeness in the form of the ones for soundness seen in the previous section cannot be shown. Such a theorem would look like:

$$\mathsf{safe\_formula}\ \varphi \Longrightarrow \mathsf{wty\_envs}\ S\ \sigma\ V \Longrightarrow \mathsf{nfv}\ \varphi \leq \mathsf{length}\ v \Longrightarrow \\ (\forall V\ v\ i.\ \mathsf{sat}\ \sigma\ V\ v\ i\ \varphi \longleftrightarrow \mathsf{sat}'\ \sigma\ V\ v\ i\ \varphi) \Longrightarrow S, E \vdash \varphi$$

We prove by a counterexample why such a theorem cannot exist.

**Example 3.13** *Take as an example the formula* $\varphi = (\mathsf{Int}\ 42 \approx \mathsf{Str}$ "*hello world*"$) \wedge (\mathsf{Int}\ 0 \approx \mathsf{Int}\ 1)$. *It is clearly safe (Section 2.1.2) and not well-typed due to the type clash in the left-hand side of the conjunction. Let us consider a long enough assignment v, and S and V such that* wty_envs $S\ \sigma\ V$ *holds. The right-hand side is clearly unsatisfiable, thus so is* $\varphi$. *Hence,* $\mathsf{sat}\ \sigma\ V\ v\ i\ \varphi$ *and* $\mathsf{sat}'\ \sigma\ V\ v\ i\ \varphi$ *are equal for any i and* $\sigma$. *Thus all assumptions for the above theorem are met. We arrive at a contradiction since* $\varphi$ *is not well-typed.*

Chapter 4

# Type Inference Algorithm

In this chapter we present the type inference algorithm for MFODL terms and formulas and the respective soundness and completeness statements. We decided to create an algorithm which works for both safe and unsafe formulas. This way, the type-checking algorithm can be applied before checking the monitorability of a formula. Before specifying the algorithm, we need to introduce new definitions and functions. Furthermore, we want to give the user the option of specifying type annotations for free variables and bound variables.

## 4.1 Partial type symbols

If we want a type inference algorithm which only iterates once through a formula, it must be able to store partial typing judgments. For example, consider the formula $(x \approx y + z) \wedge x \approx$ "*hello world*". When encountering the left side of the conjunction the algorithm must store the information *the type of variable x must be numeric*, such that it can then reject the formula when checking the right hand side.
Additionally, it must store information in the sense of *variable x and y must be of the same type*, such that it can reject e.g. the formula $(x \approx y) \wedge x \approx$ "*hello world*" $\wedge\, y \approx 42$.
If we now introduce support for these two sorts of information, the user is also able to specify constraints on the types in this form (e.g. $x \approx y$ *and x and y should be numeric*).
Another benefit from using partial typing judgments is that a type inference algorithm can not only decide whether there exists a type clash in a formula, but also return the most general typing judgment. For example, consider the formula: $x + y \approx z$. There is more than one type environment for which the formula is well-typed: Either $x$, $y$ and $z$ are all of type TInt or TFloat. Instead of returning a set of all possible environments it is more pleasant to have a partial type environment, which stores this information. Thus, we introduce partial type symbols and the corresponding environment. In the following we will call them type symbols, whereas we refer to the types from Chapter 3 as ground types or just types.

**datatype** *tysym* = TAny *nat* | TNum *nat* | TCst *ty*
**type_synonym** *tysenv* = *nat* ⇒ *tysym*

The constructors stand for the subsets of possible ground types (TAny: can be any ground type *ty*, TNum must have numeric ground type, TCst ground type must be equal to the type in the constructor). The natural numbers in TAny and TNum serve as variables, to express that two types must be equal.
A type inference algorithm starts with the most general type symbols, unless the user restricts them. It then refines them accordingly to the typing rules. For this purpose we formalize the notion of a minimal type symbol, as well as a type clash. We merge these two concepts into one function min_type. For two given type symbols this function returns None if the two type symbols clash (they cannot be instantiated to any common ground type). If this is not the case, min_max_type returns an ordered pair of type symbols, where the first type is the more refined of the given types and the second one is the more general one. For a pair of TAny or TNum symbols, we make the same arbitrary choice as in the MonPoly type inference algorithm: We define the type symbol with the lower natural number to be the more refined type symbol.

Let us look at a few interesting cases of min_max_type:

min_max_type (TNum $a$) (TCst TFloat) = Some (TCst TFloat, TNum $a$)
min_max_type (TCst TString) (TNum $a$) = None
min_max_type (TAny $a$) (TNum $b$) = Some (TNum $b$, TAny $a$)
min_max_type (TCst $t$) (TAny $a$) = Some (TCst $t$, TAny $a$)

Note, the commutativity of min_max_type is intuitive and proven.

## 4.2 Terms

In this section we will present the type inference algorithm for MFODL terms and then show the proven theorems for soundness and completeness.

### 4.2.1 Algorithm

The type inference algorithm for terms takes as input a type symbol environment, the input type symbol restricting the type of the term it operates on and the respective term. It returns the updated type symbol environment, and the newly inferred type of input term. Before we present it, we show and explain helper functions of the type-checking algorithm.

**definition** update_env :: *tysym* × *tysym* ⇒ *tysenv* ⇒ *tysenv* **where**
update_env $x$ $E \equiv$ **case** $x$ **of** $(newt, oldt)$ ⇒ $(\lambda v.$ **if** $E\ v = told$ **then** *tnew* **else** $E\ v)$

```
definition clash_propagate :: tysym ⇒ tysym ⇒ tysenv ⇀ (tysenv ∗ tysym) where
clash_propagate a b E =
    map_option (λx. (update_env x E, fst x))( (min_max_type a b)

definition new_type_symbol :: tysym ⇒ tysym where
new_type_symbol x = case x of
    TCst t ⇒ TCst t | TAny n ⇒ TAny (Suc n) | TNum n ⇒ TNum (Suc n)
```

The function update_env updates a given type symbol environment. For a pair of type symbols and a type symbol environment it returns the new environment in which all variables that map to the second type are remapped to the first one. This function is then used in clash_propagate, which takes two type symbols and an environment. If the two types clash, then it returns None. Otherwise, clash_propagate returns the environment in which all occurrences of the more general type symbol are replaced by the more refined one. In addition to that, clash_propagate also returns the more refined of the two type symbols.
We also defined the function new_type_symbol which takes a type symbol and returns a type symbol, where the variable is incremented by one (for the constant case it stays the same). This function composed with a type symbol environment gives us an environment where all type variables are incremented by one. This way, we can be sure that the type symbols TAny 0 and TNum 0 are fresh.
Now we continue with the definition of the algorithm. Note that we omit some cases, as they are similar to others.

```
definition check_binop where
check_binop check_trm E typ t1 t2 exp_typ =
  (case clash_propagate exp_typ typ E of
    Some (E', newt) ⇒ (case check_trm E' newt t1 of
      Some (E'', t_typ) ⇒ check_trm E'' t_typ t2
    | None ⇒ None)
  | None ⇒ None)

fun check_trm :: tysenv ⇒ tysym ⇒ trm ⇀ tysenv ∗ tysym where
  check_trm E typ (V x) = clash_propagate (E x) typ E
| check_trm E typ (C x) = clash_propagate (TCst (ty_of c)) typ E
| check_trm E typ (f2i(t)) = (case clash_propagate typ (TCst TInt) E of
        Some(E', prec_typ) ⇒ (case check_trm E' (TCst Float) t of
          Some (E'', t_typ) ⇒ Some (E'', TCst TInt)
        | None ⇒ None)
      | None ⇒ None)
| check_trm E typ (− t) =
      (case clash_propagate (TNum 0) (new_type_symbol typ) (new_type_symbol ∘ E) of
          Some (E', prec_typ) ⇒ check_trm E' prec_typ t
        | None ⇒ None)
| check_trm E typ (t1 + t2) =
      check_binop check_trm (new_type_symbol ∘ E) (new_type_symbol typ) t1 t2 (TNum 0)
```

| check_trm $E$ $typ$ $(t_1 \ \% \ t_2)$ = check_binop check_trm $E$ $typ$ $t_1$ $t_2$ (TCst TInt)

In the variable and constant case a simple call to clash_propagate is enough to return the refined type symbol environment or None, in the case of a type clash.
For the conversion functions i2f() and f2i() we first check whether the input type symbol of the whole term is does not clash with the respective ground type of the conversion function (TInt for f2i() and TFloat for i2f()) and refine the environment accordingly. We then recursively continue type inference with the term to be converted, which has to be of type TFloat for f2i() (TInt for i2f()).
If the term is a unary minus, then the algorithm generates a fresh numeric type symbol, by incrementing the type variables in the environment and the expected type. It then compares the fresh TNum 0 to the expected type of the term. If there is no type clash, the more specific type symbol of both of these is used as input type symbol in the recursive call to the subterm.
Since addition, subtraction, multiplication, division and modulo all are quite similar, we group them into one function check_binop, which takes the function check_trm as an argument, such that it can then call it to check both subterms. As additional arguments check_binop takes: the input type symbol environment $E$, the input type symbol $typ$, the two subterms $t_1$, $t_2$ and a type symbol $exp\_typ$, corresponding to the types that our type system from Chapter 3 enforces the operands to be of. For addition, subtraction, multiplication and division all these calls are the same: A fresh numeric type symbol is created, which is then passed as $exp\_typ$ to check_binop. Even simpler is the modulo case. Since both operands must be integers, $exp\_typ$ is equal the constant type symbol TCst TInt and no fresh type variable needs to be created. The function check_binop operates similar to the unary minus case. First it is checked whether $typ$ and $exp\_typ$ (e.g. $exp\_typ$ = TNum 0 for addition) do not type clash. After updating the types accordingly, the two subterms are checked using check_trm and the refined environments and type symbols.

We had to prove an additional lemma, such that Isabelle could finish the termination proof for check_trm. The lemma states that check_binop only makes calls to check_trm with terms of size smaller or equal to size $t_1$ + size $t_2$.

### 4.2.2 Correctness

Intuitively, check_trm on a well-typed term should calculate a refined type symbol environment and a refined type of the term. We established the following formalization for what it means for a result of check_trm to be more refined.

**definition** wf_f :: $(tysym \Rightarrow tysym) \Rightarrow bool$ **where**
wf_f $f$ = $(\forall x.\ f\ (\mathsf{TCst}\ x) = \mathsf{TCst}\ x) \wedge (\forall n.$ **case** $f\ (\mathsf{TNum}\ n)$ **of**
TCst $x \Rightarrow x \in$ numeric_ty | TNum $x \Rightarrow$ True | _ $\Rightarrow$ False)

**definition** resless_trm $E'$ $E$ $typ'$ $typ$ $\longleftrightarrow$ $(\exists f.\ \mathsf{wf\_f} f \wedge E' = f \circ E \wedge typ' = f\ typ)$

We define one result to be less or equal (in terms of refinement) than another, if there exists a well-formed refining function, which maps the more general result to the more specific one. A function $f$ is well-formed and refining (wf_f $f$ ), if $f$ maps all constant types to itself and all numeric type symbols to another numeric type variable or to a numeric ground type.
If all the type symbols in the range of the refined environment $E'$ and the corresponding type symbol $typ'$ are constant type symbols (TCst _) we say $E'$ and $typ'$ is a ground type instantiation for $E$ and $typ$
With this definition of refinement we make sure that all variables in the domain of the more general environment $E$ are mapped to a more refined or equal type symbol in $E'$. Due to the fact that $E'$ is the same as $f$ composed with $E$ it is assured, that variables mapping to the same type variables in $E$ have still the same image under $E'$.
Note that for two results $(E, typ)$, $(E', typ')$ where $(E', typ')$ is a renaming of the type variables of $(E, typ)$, resless_trm $E'$ $typ'$ $E$ $typ$ and resless_trm $E$ $typ$ $E'$ $typ'$ both hold. One particular example for this is $(E', typ') = (\mathsf{new\_type\_symbol} \circ E, \mathsf{new\_type\_symbol}\ typ)$.

In order to formalize, what it means for a result of our algorithm to be correct, we define wty_result_trm $t$ $E'$ $typ'$ $E$ $typ$. It takes as arguments the term to be typed $t$, the result of the algorithm $E'$, $typ'$ and the input $E$, $typ$.

**definition** wty_result_trm $t$ $E'$ $typ'$ $E$ $typ \longleftrightarrow$ resless_trm $E'$ $E$ $typ'$ $typ \wedge$
$(\forall E''\ typ''.$ resless_trm $(\mathsf{TCst} \circ E'')\ E\ (\mathsf{TCst}\ typ'')\ typ \longrightarrow$
$(E'' \vdash t :: typ'' \longleftrightarrow$ resless_trm $(\mathsf{TCst} \circ E'')\ E'\ (\mathsf{TCst}\ typ'')\ typ'))$

Intuitively, this Boolean predicate should be satisfied if $E'$ and $typ'$ is the most general result for check_trm $E$ $typ$ $t$, which is sound, i.e. every ground instantiation of $E'$ $typ'$ respects $E$ $typ$ and types $t$. We explain wty_result_trm in more detail after presenting the theorem.

**Theorem 4.1** *If*

$$\mathsf{check\_trm}\ E\ typ\ t = \mathsf{Some}\ (E', typ')$$

*holds, then*

$$\mathsf{wty\_result\_trm}\ t\ E'\ typ'\ E\ typ$$

The theorem is proven in Isabelle by a computational induction over the recursive calls of the check_trm function. It states, that if check_trm returns a type symbol environment and a type symbol, these are correct with respect to wty_result_trm.
We now argue why this gives us a good correctness guarantee. By doing so, we motivate and explain the definition of wty_result_trm. First of all, the theorem states that the returned environment and type symbol are refining the input. The other side of the conjunction then assures, that for all possible ground type environments $E''$ and

ground type $typ''$, which are compatible with the input $E$ and $typ$ the following holds: Term $t$ is well-typed with respect to $E''$ and $typ''$ if and only if $E''$ and $typ''$ are also compatible with $E'$ and $typ'$, i.e. $\mathsf{TCst} \circ E''$ and $\mathsf{TCst}\ typ''$ refine $E'$ and $typ'$. Since we have an equivalence between the two statements, we get a soundness as well as a completeness property.
The second direction corresponds ($\longleftarrow$) to a soundness property: For every $E''$ and $typ''$ with $\mathsf{resless\_trm}\ (\mathsf{TCst} \circ E'')\ E'\ (\mathsf{TCst}\ typ'')\ typ'$, it follows that term $t$ is well-typed with respect to $E''$ and $typ''$. Thus, for the returned $E'$ and $typ'$ there does not exist a ground type instantiation for which $t$ is not well-typed, i.e. they only capture ground type instantiations for which $t$ is well-typed. It follows, the algorithm is sound with respect to our type system.
The other direction ($\longrightarrow$), corresponds to a completeness property: For every ground type instantiation $E''$, $typ''$ of $E$, $typ$, for which term $t$ is well-typed, there exists a well-formed function which maps the output $E'$ to $E''$ and $typ'$ to $typ''$. In other words, for every $E''$ and $typ''$, for which $t$ is well-typed, if $E$, $typ$ is compatible with $E''$ and $typ''$, then also $E'$, $typ'$ is compatible with $E''$ $typ''$. Thus, $E'$ and $typ'$ are (one of) the most general environment and type symbol. The result of our algorithm hence covers all possible ground type instantiations with respect to the input and the type system. This is a completeness property.

However, Theorem 4.1 only covers completeness if the type checking algorithm succeeds. We now show another completeness theorem which shows that if the algorithm fails, i.e. returns None, there indeed do not exist an environment and type which are compatible with the input and type the given term. To give intuition why such a theorem is necessary, we provide a quick example of a type inference algorithm.

**Example 4.2** *Let* $\mathsf{check\_trm}'\ E\ typ\ t = \mathsf{None}$*, for all inputs* $E$*,* $typ$*,* $t$*. Obviously, such an algorithm is not desired. But yet it is still sound and even Theorem 4.1 holds for this algorithm. The proof to this is trivial, since the assumption* $\mathsf{check\_trm}'\ E\ typ\ t = \mathsf{Some}\ (E', typ')$ *never holds.*

We have proven the following theorem in Isabelle by a computational induction over the function check_trm.

**Theorem 4.3** *If*

$$\mathsf{check\_trm}\ E\ typ\ t = \mathsf{None}$$

*then there is no* $E''$ *and* $typ''$ *such that*

$$\mathsf{resless\_trm}\ (\mathsf{TCst} \circ E'')\ E\ (\mathsf{TCst}\ typ'')\ typ \wedge E'' \vdash t :: typ''$$

The theorem proven in Isabelle is $\mathsf{check\_trm}\ E\ typ\ t = \mathsf{None} \Longrightarrow \mathsf{resless\_trm}\ (\mathsf{TCst} \circ E'')\ E\ (\mathsf{TCst}\ typ'')\ typ \Longrightarrow E'' \vdash t :: typ'' \Longrightarrow \mathsf{False}$, which is equivalent to Theorem 4.3.
The theorem states that if check_trm fails, then there does not exist a ground type

environment $E''$ and a ground type $typ''$, which refines the input $E$, $typ$ and for which the term to be checked is well-typed. This gives us the desired completeness property: If there exists a ground type instantiation $E''$, $typ''$ of $E$, $typ$ which types term $t$, check_trm will not fail (i.e. will find $E''$, $typ''$).

## 4.3 Formulas

Let us now move on to the type inference algorithm for MFODL formulas. There are three main differences compared to terms: Formulas are of type Boolean, but instead they include bound variables and the well-typedness of a formula is defined with respect to a signature $S$ of predicate symbols, in addition to the environment $E$. In our algorithm we consider a fixed signature specifying ground types. If we encounter a let operator we update it with the new predicate, else it remains the same. Recall that in Chapter **??**, we introduced the polymorphic formula datatype such that *ty formula* stores the types of bound variables. To store type symbols we now consider formulas of the type $tysym\ frm$.

### 4.3.1 Introducing Type Symbol Functions

Let us first consider an approach to a type inference algorithm for formulas similar to check_trm. It would take as arguments a signature $S$, type symbol environment $E$ and a formula $\varphi$ (of type *tysym frm*) and return a refined environment $E'$ and a formula $\varphi'$ with refined type symbol for the bound variables. We will now show an example why such a simple algorithm would not work.

**Example 4.4** *Consider the formula* $(\exists(x : \mathsf{TAny}\ 0).\ x \approx 42) \wedge (\exists(x : \mathsf{TAny}\ 0).\ x \approx$ "*hello world*"*), a type symbol environment E where* $\mathsf{TAny}\ 0$ *is not in the image and any signature S. If the algorithm evaluates the left-hand side of the conjunction it would return E and the formula* $(\exists(x : \mathsf{TCst}\ \mathsf{TInt}).\ x \approx 42)$*. It would lose the information, that the bound variable was of type* $\mathsf{TAny}\ 0$ *before. Thus, the right hand side would type as well and yield:* $(\exists(x : \mathsf{TCst}\ \mathsf{TString}).\ x \approx$ "*hello world*"*). The users specification, that both bound variables must be of the same type is thus violated, as this information was lost during the first recursive call. The algorithm could try to restore the lost information, however this would lead to a more involved design.*

In order for the algorithm's output to give more information about the bound variables, we change our algorithm such that it returns a type symbol function $f$, which maps each type symbol to the corresponding refined type symbol. This construct is stronger than an environment and a type symbol formula: If we are given the refining function, it is easy to compute the environment and the formula: $E' = f \circ E$ and $\varphi' = \mathsf{map\_frm}\ f\ \varphi$.

We introduce a function computing the set of all used type symbols for a given formula $\varphi$ and a type symbol environment $E$. This set is simply the union between the image of the free variables under E and the type symbols of the bound variables of $\varphi$.

**definition** used_tys $E\ \varphi \equiv E'$ fv $\varphi \cup$ set_frm $\varphi$

We add also a set $X$ of type symbols to the arguments of our type inference function. This set corresponds to all the relevant type symbols and is initialized to used_tys $E\ \varphi$ for the overall formula $\varphi$ and environment $E$. Throughout the execution $X$ changes, if the type symbols get remapped. For all type symbols outside this set it does not matter how the return function $f$ behaves.
With these changes the type for our type inference algorithm is as follows.

**fun** check :: $sig \Rightarrow tysenv \Rightarrow tysym\ set \Rightarrow tysym\ frm \rightharpoonup (tysym \Rightarrow tysym)$

We are very confident that this approach can be used for type-checking MFODL terms as well, i.e. we could reformulate check_trm such that it takes the set $X$ as additional argument and returns a function $f$, instead of a type symbol environment $E'$ and a type symbol $typ'$.

**fun** check_trm_f :: $tysenv \Rightarrow tysym \Rightarrow tysym\ set \Rightarrow trm \rightharpoonup (tysym \Rightarrow tysym)$

Due to lack of time we did not yet change the algorithm for terms. Instead we assume that such a function check_trm_f exists, and formulate check with respect to this function. Additionally, we the adapted Theorem 4.1 and the Theorem 4.3 for check_trm_f, and assume that they hold. We think, check_trm as described in Section 4.2 provides a good foundation for future work to implement such a function check_trm_f. Also for the correctness proofs, future work can adapt the proofs presented in Section 4.2. One observation which supports this assumption is, that in the definition of resless_trm we are already working with type symbol functions.

### 4.3.2 Algorithm

We created a Isabelle locale which includes the type inference algorithm for formulas and the corresponding correctness proof. It fixes the function check_trm_f and makes a soundness assumptions which we now present. First, we introduce a definition similar to wty_result_trm from Section 4.2 but now for the approach using type symbol functions.

**definition** wty_result_fX_trm $E\ typ\ t\ f\ X \longleftrightarrow$ wf_f $f \wedge$
$(\forall f''.\ \text{wf\_f}\ (\text{TCst} \circ f'') \longrightarrow (f'' \circ E \vdash t :: f''\ typ \longleftrightarrow (\exists g.\ \text{wf\_f}\ (\text{TCst} \circ g) \wedge$
$(\forall t \in X.\ f''\ t = g\ (f\ t)) \wedge f''\ typ = g\ (f\ typ)))$

The Boolean predicate wty_result_fX_trm is defined similarly to wty_result_trm, but now the ground type instantiation of $E$, *typ* is calculated using a function $f''$. Another change is that refinement of environments is now only over the set of relevant type symbols $X$. That is, the right hand-side of the equivalence now states that the ground instantiation $f'' \circ E$ is compatible with the refined environment $f \circ E$ only on the relevant type symbols (the ones in $X$). We get the following correctness assumption for check_trm_f.

**Assumption 4.5**

check_trm_f $E$ *typ* $X$ $t$ = Some $f \Longrightarrow$ fv_trm $t \subseteq X \Longrightarrow$ wty_result_fX_trm $E$ *typ* $t$ $f$ $X$

We now present the provably correct cases of our check function, which performs type inference on MFODL formulas.

**fun** check :: *sig* ⇒ *tysenv* ⇒ *tysym set* ⇒ *tysym frm* ⇀ (*tysym* ⇒ *tysym*)
  check $S$ $E$ $X$ $(t_1 \approx t_2)$ =
      (**case** check_trm_f (new_type_symbol ∘ E) (TAny 0) (new_type_symbol′ $X$) $t_1$ **of**
    Some $f$ ⇒ (**case** check_trm_f ($f$ ∘ new_type_symbol ∘ E) ($f$ (TAny 0))
        (($f$ ∘ new_type_symbol)′ $X$) $t_2$ **of**
      Some $f'$ ⇒ Some ($f'$ ∘ $f$ ∘ new_type_symbol) | None ⇒ None)
  | None ⇒ None)
| check $S$ $E$ $X$ $(\varphi \wedge \psi)$ = (**case** check $S$ $E$ $X$ $\varphi$ **of**
    Some $f$ ⇒ (**case** check $S$ $(f \circ E)$ $(f\,' X)$ (map_frm $f$ $\psi$) **of**
      Some $f'$ ⇒ Some ($f'$ ∘ $f$)
    | None ⇒ None)
  | None ⇒ None)
| check $S$ $E$ $X$ $(\exists\, t.\, \varphi)$ = check $S$ (case_nat $t$ $E$) $X$ $\varphi$
| check $S$ $E$ $X$ $(\neg\varphi)$ = check $S$ $E$ $X$ $\varphi$

Note that $\prec$ and $\preceq$ cases are exactly the same as $\approx$. In the cases $\vee$, S, U check is defined exactly the same as shown for $\wedge$. The case $\neg$, is representative all unary formulas ($\neg$,○, ●).
In the case of a relation between terms, we first do not know anything about the type of the term. Thus, a fresh type symbol TAny 0 is created and passed as the expected type to check_trm_f. Note that we not only have to increment the type symbol environment $E$ but also the set of relevant type symbols $X$. If there is no type clash in the first term $t_1$ the refining function $f$ is returned. This function $f$ is then prepended to every argument, to get the correctly refined type symbols and check_trm_f is called on $t_2$. If this type checks as well, all the refining functions (including new_type_symbol) are composed in the correct order and then returned. If one of both calls fails check returns None as well.
The binary formula case is very similar. After the first formula is checked, the arguments get refined (also the bound variables on the right-hand side) and the second one is checked and eventually the composition of both functions returned, if no clash

occured.
The case of an existential quantifier is very simple: Due to de Bruijn indices, the type symbol environment gets adjusted with the help of the case_nat function, we have already seen in Chapter 3. The case of a negated formula is one simple recursive call to the function below the negation with the same arguments.

### 4.3.3 Correctness

First of all we need a correctness definition similar to wty_result_fX_trm. We introduce a Boolean predicate for formulas which is very similar.

**definition** wty_result_fX $S\ E\ \varphi\ f\ X \longleftrightarrow$
$\mathsf{wf\_f}\ f \wedge (\forall f''.\ \mathsf{wf\_f}\ (\mathsf{TCst} \circ f'') \longrightarrow (\ S, f \circ E'' \vdash t :: (\mathsf{map\_frm}\ f''\ \varphi) \longleftrightarrow$
$(\exists g.\ \mathsf{wf\_f}\ (\mathsf{TCst} \circ g) \wedge (\forall t \in X.\ f''\ t = g\ (f\ t))))$

The differences of wty_result_fX compared to wty_result_fX_trm is, that instead of a type *typ* and a term *t*, wty_result_fX takes a tysym formula $\varphi$ as argument. Additionally, to formulate statements about the well-typedness of $\varphi$, a signature $S$ is needed. The form of the predicate stays similar to wty_result_fX_trm for terms. The main difference is the highlighted part and due to the difference arguments the well-typed predicate takes. As the typing judgments for formulas do not carry type symbols *typ* is no longer needed. This changes also the right-hand side of the equivalence ($f''\ typ = g\ (f\ typ)$ can just be omitted).

**Theorem 4.6** *If Assumption 4.5 holds, $\varphi$ only includes binary formulas, unary formulas, relations between terms and existential quantifiers, and*

$$\mathsf{check}\ S\ E\ X\ \varphi = \mathsf{Some}\ f$$

*and*

$$\mathsf{used\_tys}\ E\ \varphi \subseteq X$$

*hold, then*

$$\mathsf{wty\_result\_fX}\ S\ E\ \varphi\ f\ X$$

Note that this theorem would want to be generalized for all possible cases of $\varphi$. The cases for which Theorem 4.6 is not proven yet are namely: the predicate and let case, the conjunction of a list of formulas, the aggregation case and the match operators for regular expressions. The Isabelle proof of Theorem 4.6 includes an additional assumption wf_formula $\varphi$, which gives some guarantees about $\varphi$, without loss of generality. For the cases proven, this assumption was not needed and is therefore not listed in Theorem 4.6. However, this assumption could help simplify the proofs for the more involved cases. The proofs are completed for the simple base case of a relation, for the induction cases like binary or unary formulas, as well as for formulas that introduce a bound variable. The other sorts of formulas all fall into one of these

categories (or a combination), even though a bit more involved. Because of this observation we are very optimistic that the proof can be completed for the remaining cases. However, it is possible that the algorithm needs to be slightly adjusted for these cases.

We thus have proven soundness and completeness statements similar to the ones for terms (Theorem 4.1) for the above mentioned formula cases. What remains open is to prove Theorem 4.6 for all cases, as well as a completeness theorem in the sense of Theorem 4.3. We discuss future work further in Chapter 6.

Chapter 5

# Discussion

Besides formalizing the existing type inference algorithm and proving partially its correctness, we also extended the algorithm with new features. Namely, the algorithm supports additional typing constraints which can be specified by a user for the input formula. This works for free and bound variables. At the moment this is not supported in VeriMon. The unverified type inference algorithm in MonPoly is general enough to support this only for free variables. However, in practice users cannot specify these constraints. The type symbols are hand-coded to be the most general. Furthermore, there would be no way to support typing constraints of the following form: $\exists x :: \mathsf{TAny}\ 0.\ Q(x) \vee \exists y :: \mathsf{TAny}\ 0.\ P(y)$ , i.e. both bound variables must be of the same type. This additional feature was implemented with the help of the polymorphic formula datatype, which stores type symbols or types for bound variables.
Due to the formalization of typing rules, we generalized the semantics of VeriMon. We did not only discover inconsistencies concerning the evaluation of aggregations but also a bug in the respective semantics. The median over a list of even length was incorrectly defined. If $x$ and $y$ are the two elements in the middle of the sorted list, the median was defined to be $x + y/2.0$ instead of $(x+y)/2.0$. We corrected this bug and improved the consistency in how the default values of aggregation operations are handled.
During the process of formalizing the type inference algorithm and proving its correctness we were able to find a bug in the MonPoly type inference implementation. The case of a binary operation on terms was not handled correctly. For example, the inferred type symbols from formula $1 \approx x + y$ with starting environment $E$ with $E\ x = \mathsf{TAny}\ 1, E\ x = \mathsf{TAny}\ 2$ would be $x :: \mathsf{TNum}\ t3,\ y :: \mathsf{TAny}\ 1$. Note, that variable $y$ got assigned the type symbol $x$ started with. The reason for this bug was that the type symbol returned in the variable case was the old value of $E\ x$, even if it got refined. Thus, in our example, the recursive call in the addition refines the type of $x$ to $\mathsf{TNum}\ 3$ but returns $\mathsf{TAny}\ 1$, which then is used to refine the type of $y$. After the discovery the bug has been fixed in the unverified algorithm.

As mentioned in Chapter 3, we changed the safe_formula function without loss of generality. For the existential quantifier we added the statement that the bound variable needs to occur in the formula. Clearly, this is without loss of generality: If an existential quantifier does not satisfy this, one can just omit the quantifier. The second change to safe_formula was to omit the special case $\neg\,(V\ x \approx V\ y)$, which was defined to be safe iff variable $x$ and $y$ are the same. If this is a safe formula it is clearly equivalent to False, and one can omit it w.l.o. The only reason why this special formula was defined to be safe is the implementation of True and False. False was defined as $FF = \exists.\ \neg(V\ 0 \approx V\ 0)$ and True as $TT = \neg FF$. We changed these definitions to $FF = (\mathsf{Int}\ 0 \approx \mathsf{Int}\ 1)$ and $TT = (\mathsf{Int}\ 0 \approx \mathsf{Int}\ 0)$. Now $\neg\,(V\ x \approx V\ x)$ no longer needs to be safe.
After these changes it was possible to prove the following two lemmas:

**Lemma 5.1** $\mathsf{safe\_formula}\ \varphi \Longrightarrow S,\ E \vdash \varphi \Longrightarrow S, E' \vdash \varphi' \Longrightarrow \mathsf{rel\_frm}\ f\ \varphi\ \varphi' \Longrightarrow x \in \mathsf{fv}\ \varphi \Longrightarrow E\ x = E'\ x$

**Lemma 5.2** $\mathsf{safe\_formula}\ \varphi \Longrightarrow S,\ E \vdash \varphi \Longrightarrow S, E' \vdash \varphi' \Longrightarrow \mathsf{rel\_frm}\ f\ \varphi\ \varphi' \Longrightarrow \varphi = \varphi'$

Both lemmas were proven by one of the supervisors of the thesis. The former by a computational induction over safe_formula. The latter follows from $\mathsf{safe\_formula}\ \varphi \Longrightarrow S,\ E \vdash \varphi \Longrightarrow S, E' \vdash \varphi' \Longrightarrow \mathsf{rel\_frm}\ f\ \varphi\ \varphi' \Longrightarrow \mathsf{rel\_frm}\ (=)\ \varphi\ \varphi'$ which again was proven by a computational induction over safe_formula.
Together these lemmas show that if two safe formulas $\varphi$ and $\varphi'$ have the same shape (i.e. they are the same except for the types of the bound variables), and are well-typed with respect to the same signature $S$ the types of the bound and free variables both must be equal. This means, that for every formula and signature there exists only one ground type instantiation which is well-typed. Hence, we conjecture that for a safe formula $\varphi$ our type inference algorithm outputs a refining function which maps all the used type symbols to ground types.

Chapter 6

# Conclusion

We have formalized a type system for metric first-order dynamic logic (MFODL) terms and formulas and proven its soundness. By doing so we could generalize the semantics of VeriMon: there is no longer need for arbitrary choices in the interpretation of functional symbols. Additionally, we discovered and corrected a minor flaw in VeriMon's semantics for median aggregations. Furthermore, we have formalized a type inference algorithm for MFODL terms and formulas, which supports type constraints specified by the user for free and bound variables. By doing so we introduced a new feature. Due to the soundness proof we could find a bug in the unverified type-checking algorithm.
Our verified algorithm for terms returns a type symbol environment and the inferred type symbol for the term. We proved the soundness and completeness of this algorithm with respect to the type system. Due to bound variables the algorithm for formulas uses a different approach: returning a function that refines type symbol. We proved the soundness of the algorithm for a subset of formulas under the assumption that there exists a sound algorithm for terms which also returns a function.
Possible future work would be to generalize the soundness proof for all possible formulas and prove completeness. Furthermore, the algorithm for terms could be adjusted to return the type-refinement function and then the soundness and completeness proof adapted such that the statements the formula type-checking function assumes would be met. Next one would need to integrate the verified type checking algorithm into VeriMon in order to improve the trustworthiness. The algorithm would replace the unverified OCaml implementation.
Additionally, the algorithm for formulas could be extended to infer types of arguments for predicates. This means the algorithm would take as an argument a signature mapping from predicate names to type symbols instead of ground types. Then if the type symbols for arguments of a given predicate are not known to be constants, the algorithm would refine them.
Future work could additionally try to prove our conjecture from Chapter 5: For a safe formula our type inference algorithm should always return a ground type instantiation.

# Bibliography


[1] David Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In *International Joint Conference on Automated Reasoning*, pages 432–453. Springer, 2020.

[2] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal methods in system design*, 46(3):262–285, 2015.

[3] David Basin, Felix Klaedtke, and Eugen Zalinescu. The monpoly monitoring tool. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.

[4] Olivier Boite and Catherine Dubois. Proving type soundness of a simply typed ml-like language with references. 01 2001.

[5] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer*, 23(2):255–284, 2021.

[6] Srđan Krstić and Joshua Schneider. Typing rules for MFODL in MonPoly. Unpublished draft, October 2020.

[7] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm w in isabelle/hol. *Journal of Automated Reasoning*, 23(3):299–318, Nov 1999.

[8] Wolfgang Naraschewski and Tobias Nipkow. Mini ml. *Archive of Formal Proofs*, March 2004. https://isa-afp.org/entries/MiniML.html, Formal proof development.

[9] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

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

[11] Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. A formally verified monitor for metric first-order temporal logic. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification*, pages 310–328, Cham, 2019. Springer International Publishing.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Formalizing Typing Rules for VeriMon

**Authored by** (in block letters):

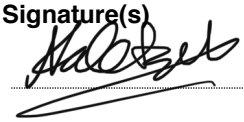
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Kaletsch | Nicolas |
| | |
| | |
| | |
| | |

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| Lengnau, 19.10.2021 |  |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*