Bachelor thesis
Computing Science

Radboud University

mTask semantics and its comparison to TopHat

Author:
Elina Antonova
s1057069

Supervisor:
dr. Pieter W.M. Koopman
pieter@cs.ru.nl

Daily supervisor:
Mart Lubbers MSc
mart@cs.ru.nl

Second reader:
dr. Peter M. Achten
p.achten@cs.ru.nl

July 12, 2022

Abstract

Task-Oriented Programming (TOP) is a relatively new programming paradigm designed to make the process of developing multi-purpose applications easier and faster. There are several TOP implementations. The first of them is iTask, a multi-purpose language embedded in the functional programming language Clean. As the result of the growing number and popularity of Internet of Things (IoT) systems, another TOP language—mTask—emerged for programming such systems. The mTask system is implemented in Clean, which enables the connection of iTask and mTask applications; creating distributed applications using internet forms with IoT systems. Even though the mTask language uses the same programming paradigm as iTask and is implemented in the same language, mTask is easier to reason about as it is less complex. Language formal reasoning plays an important role in its theorisation and the proof of the correctness of its implementation. TopHat ($\widehat{TOP}$) is the first TOP implementation that is fully formalized and has well-defined operational semantics. The way $\widehat{TOP}$'s syntax and semantics are described is used in this thesis as basis for formalising mTask. We describe the mTask language syntax, types and typing rules, and the language evaluation, normalisation and interaction with the system using operational semantics. $\widehat{TOP}$ and mTask semantics turned out to differ more than expected. Compared to $\widehat{TOP}$'s input-driven semantics, mTask's semantics is a continuously rewriting system that does not need any input for the rewriting process. The way user input is handled also differs due to the structure of the tasks that deal with the values. In addition, mTask has a notion of stability that is lacking in $\widehat{TOP}$. Stability of task values has to be considered when the rules for the mTask semantics are created.

Acknowledgements

This research would not have happened without the help and support from many people. I would like to express my sincere appreciation to my first supervisor, Pieter Koopman, who provided me with this interesting topic and guided through the whole process of the thesis writing. I wish to express my thanks to my daily supervisor, Mart Lubbers, who always provided an extensive feedback and tips how to improve my work. Along with my first supervisor, he helped me with research when I was stuck. With all this great supervision, the process of bachelor thesis research was a pleasant experience.

I would also like to thank Bachelor Thesis course's professors, Peter Achten and Perry Groot, for providing all the necessary knowledge about the thesis writing, planning and presentation, which made the process easier and more structured. In addition, I would like to thank Peter Achten for his input as a second reader.

Additionally, I would like to thank my loved ones and friends for helping me and providing support, which helped to stay on track and finish this research. Their feedback was also valuable for improving some written parts of the thesis.

Contents

Chapter 1

Introduction

Every day, people do all kinds of activities: they go to work, their work time consists of smaller jobs, they make food, eat food, sleep and do many other small activities. Such activities can be called *tasks*, and they describe an abstract unit of work. Such tasks are often done sequentially: a person wakes up, showers, makes coffee, drinks coffee, gets ready, leaves for work, etc. But sometimes coffee and breakfast are consumed at the same time, which means that these two tasks are done in parallel. Such tasks sequence or, simply, the workflow are an interconnected set of tasks, that create people's daily schedules.

Similar workflows are used in machines. When programs are divided into smaller tasks that are connected sequentially or parallely, it creates an interesting workflow that can be used in programming. This programming paradigm does not fall under imperative, object-oriented or functional programming, so a new paradigm was designed for the creation of such workflow programming—Task-Oriented Programming (TOP) [1].

The TOP paradigm is a relatively new concept created for the development of interactive, collaborative and distributed applications. It focuses on modelling collaboration patterns according to the needs of users to interact and share information. The TOP paradigm makes use of the definitions of a *task*, which is an abstract definition of a unit of work, and a *combinator*, which is used to create the workflow of the program by connecting tasks together [2]. Initially the paradigm was used for internet forms, but lately TOP found its use in IoT systems programming.

The iTask language is the first TOP language embedded in the Clean programming language. It is used for developing the real-world applications [3]. The language is too complex to define its formal semantics, as such it merely exists in implementation [1]. Moreover, iTask is considered to be very tightly integrated with Clean, making the essence of the TOP concept unclear. Thus, there was a need to create something to formalise the TOP paradigm, giving a start to the development of a new TOP language—

TopHat ($\widehat{TOP}$) [4].

$\widehat{TOP}$ is a TOP language that is mathematically formalised. In $\widehat{TOP}$, the task layer is separated from the host language syntactically and semantically to give a better overview of the TOP paradigm [4]. Operational semantics for the $\widehat{TOP}$ was fully formalised and its soundness and correctness was proven [4].

1.1 IoT and mTask

The number of devices connected to the internet is continuously growing, and it is predicted that the number of such devices will reach 83 billion by the year 2024 [5]. This network of connected devices is called Internet of Things (IoT), and it entails a distributed network of devices that interact with each other, end users and the real world.

An IoT device is used as an umbrella term for all kinds of devices from smartphones to microcontrollers, among which the computing capacity can differ tenfolds. Thus, programming languages and technologies that are used for programming applications for computers and servers are often not suitable for programming resource-restricted IoT devices, which require more light-weight approaches. This is the place where mTask comes in handy and helps to create distributed and low-cost TOP applications [6].

The mTask language is used for programming IoT devices and makes use of the TOP paradigm. It is currently still in development, but already used for programming micro controllers [7]. However, its formal foundation was never fully defined and there exists a knowledge gap in the formalisation of the language. Thus, defining a formal foundation of mTask language is the purpose of this bachelor thesis.

1.2 Research and motivation

This bachelor thesis focuses on overviewing the operational semantics of $\widehat{TOP}$ and creating an operational semantics for mTask. For the overview of the $\widehat{TOP}$ semantics, the article *TopHat: A formal foundation for task-oriented programming* written by Tim Steenvoorden, Nico Naus and Marcus Klinik [4] was used. The semantics of mTask are defined according to the TopHat operational semantics structure described above with focus on mTask's needs and peculiarities. The difference between the semantics of $\widehat{TOP}$ and mTask becomes clear when $\widehat{TOP}$ semantics is tried to be applied to the mTask language, and the changes that are made to it indicate how much the semantics of the languages differ.

The research question that is central to this thesis is: *Can we define the semantics of mTask based on semantics of TopHat ($\widehat{TOP}$)?* The way rules for types and semantics are defined, the different levels of semantics

and other key ideas of how to create semantics are taken from the $\widehat{TOP}$ approach described in the foundation paper [4] and applied on mTask.

Moreover, a formal definition of the mTask language has never been created before and its formalisation can help with the further development and improvement of the language.

1.3 Thesis structure

The following Chapter 2 introduces TOP paradigm and its languages. It describes what the notions of tasks and combinators are and provides an overview of three languages implementing the TOP paradigm: iTask, mTask and $\widehat{TOP}$. Chapter 3 describes the semantics, what types of semantics exist and provides a thorough description of the $\widehat{TOP}$ language and its operational semantics on the different levels: evaluation, normalisation and interaction. Chapter 4 focuses on the syntax, typing rules and semantics of mTask. The chapter fully focuses on the research question of this paper and describes the mTask language and its host language, an enriched λ-calculus, constructs and defines the semantics of mTask embedded in its host language. The last chapter concludes the work, discusses the related work and provides an overview of what can be done in the future to make the formal foundation of mTask completer.

Chapter 2

Task-Oriented Programming

TOP is a programming paradigm that was developed for the construction of interactive, distributed and multi-user applications. The TOP approach was designed originally for internet forms, though, its architecture makes it easy to create applications for a wide range of equipment: from computers to IoT devices. TOP provides solutions for common jobs such as GUI designing, connecting databases, and client-server communication. Moreover, TOP applications are designed to be multi-user applications [2]. Thus, TOP allows users to interact and share information using collaboration patterns.

TOP extends the host language programming paradigm with the definition of *tasks*. A *task* is an abstract description of a persistent unit of work that has an observable typed value. Tasks can be independent or rely on the results of other tasks. To interconnect tasks with each other, *combinators* are used. A program is a set of tasks that are combined using combinators. When a TOP framework executes the task, it has an opaque persistent state and related tasks can observe the current value of the executed task in a controlled way [8]. The observable value of a task can be in two different states:

1. **No value**: the task has no value of the specified type. The task could have made some progress, but the value might not be calculated yet or the task might have some internal value which is unrelated to the observable value.

2. **Typed value**: the task has achieved a value of the specified type.

Tasks can communicate using task values. However, in some collaboration patterns they need to share data using Shared Data Sources (SDSs) [7]. An SDS is a typed abstract interface over any data. It can represent different data sources such as a file, chunk of memory, database or any other data source that can be read, written and updated [8].

There are multiple ways users and systems cooperate with each other, and this is reflected in the TOP using different task combinators [4]. Gener-

ally, there are two types of collaboration that are needed for more complex patterns.

1. **Sequential composition**: tasks are executed one after another in a sequence defined by the program. Often, the results of the previously executed tasks are used in the following tasks, but this is not necessarily the case.

2. **Parallel composition**: a set of tasks is executed in parallel and task values of these tasks can be accessed from the tasks in this set.

2.1 iTask

The iTask system is a TOP framework for programming distributed collaborative web applications that are suitable for modelling collaboration in many domains [9]. iTask is a combinator language [2] and the TOP paradigm grew from it. The iTask system is implemented as an Embedded Domain Specific Language (EDSL) in Clean [2]. A compiled iTask program, which embeds a TOP specification, is a multi-user distributed webserver that hosts a *ready-for-work* user interface. The graphical user interface, serialization and communication are automatically generated by iTask system.

Tasks in iTask are the first-class citizens, which means that they can be both function arguments and function results. A task is implemented as an event-driven stateful rewrite function, so when an event occurs, the function is executed with the current state of the system with the event as an argument. Such an execution changes the state of the system and results in either a value or an exception. The resulting value includes the task value, an update for the user interface and a modified state. The current state of the task is the structure of the tasks and their combinators [3].

iTask task values are the same as in TOP, but typed values can be divided into the following states:

1. **Unstable value:** the task has a value of the correct type, but it can change after handling an event to a different value or even to no observable value.

2. **Stable value**: the task has achieved a final result. If the value is observed later, then it remains unchanged.

States of task values can change and the transitions of these states are illustrated by the state diagram in Figure 2.1.

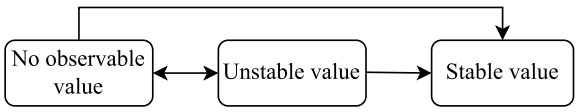

Figure 2.1: Task value state transition diagram.

2.2 mTask

Similarly to iTask, mTask is an EDSL designed for a specific domain and embedded within Clean. The mTask language is a TOP language designed to program low-cost and resource-restricted systems such as IoT systems [6].

IoT applications have a layered architecture: each layer is responsible for a certain functionality, for example, data collection or representation. Such an architecture makes the creation of these systems quite complex, but combined with iTask, mTask solves this problem and allows to create an entire IoT stack from a single specification. The mTask language is used to program low-cost and resource restricted devices, whereas iTask is used for programming multi-user and distributed applications [10]. The mTask language is simpler and smaller compared to iTask due to its narrow domain, which makes it more light-weight and programs can be run on devices with 2KiB of memory and on a slow 16 MHz processor [11]. Despite the differences, these languages are closely integrated and allow to share data with each other using SDSs and run mTask programs as iTask tasks. Their implementations are done in Clean allowing the use of the same programming language and easier integration. Tasks in mTask and iTask have the notion of stability for their values that can be stable or unstable. But compared to iTask, tasks in mTask can only be *second-class* citizens: the tasks can be the result of a function, but they cannot be arguments of the functions.

The mTask tasks are not evaluated in the host language, but the IoT application running on the server transforms the task at run time into byte code code and sends it to the IoT device. The run-time system, running on this device, interprets the byte code and the resulting task is executed by this system.

A more thorough description of the mTask language is done in Chapter 4.

2.3 TopHat

$\widehat{TOP}$ is the first TOP language that was fully mathematically formalised. The iTask language is practical in nature, but it is hard to reason about, thus, $\widehat{TOP}$ was created to fill this gap [1]. Nonetheless, $\widehat{TOP}$ can be used to create practical solutions to the programming problems.

$\widehat{TOP}$ is a programming language that consists of two parts: the host language and the task language. The task language is embedded in a simply-typed λ-calculus, i.e. the host language. $\widehat{TOP}$ programs are called tasks and they consist of basic elements called *editors* that are combined using *combinators* [12]. The $\widehat{TOP}$ language does not allow recursion.

Editors are basic tasks that communicate with the outside world. They are an abstraction over widgets in a Graphic User Interface (GUI): their values are changed by users in a similar way as to how widgets are manipulated in a GUI. The appearance of an editor, when generated from the task specification, is influenced by its type. There are three different types of editors in $\widehat{TOP}$ and we introduce them along with their evaluated semantical representation:

1. **Valued** editor ($\square$ v): the editor holds the value of the certain type and users can only change its value, not the type.

2. **Unvalued** editor ($\boxtimes$ τ): editor does not have a value, but can receive a value of the certain type τ, turning it into the valued editor.

3. **Shared** editor ($\blacksquare$ l): editor refers to a stored location, which is an SDS. The observable value is the value stored in this location, which can be changed by a new value received from the user.

The states of an editor and their transitions are depicted in Figure 2.2.

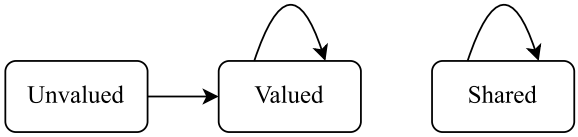

Figure 2.2: Editor value state transition diagram.

Before describing combinators, it is important to mention another basic task. **Fail** ($\lightning$) indicates a task that never has a value and never accepts input. It does not belong to the category of editors or combinators, but it plays a crucial role in indicating tasks that can fail in certain states.

Programmers can select a collaboration model that suits the users' goals best and, depending on that, there are different ways to pass task values between tasks. Data flow can be alongside control flow, when a task value is passed to the next sequential task; or it can be across the control flow, when data is shared between simultaneous working tasks. There is also a communication with the outside world (user), when data is entered into the system through input events like "continue" button or user input. The combinators are used to combine smaller tasks into larger tasks depending

on the collaboration type. The following combinators are available in $\widehat{TOP}$, where t indicates an evaluated task and e is an unevaluated expression:

1. **Sequential combinators**:
   (a) **Step** ($t \blacktriangleright e$): as soon as the task t has a task value, it is passed to the expression e. The expression is a function that takes the task value as an argument and results in a new task. This combinator is guarded and the successor task must not be the fail task $\lightning$. If it is a failing task, the combinator continues waiting for a new task value from the task t until the guard is satisfied.
   (b) **User step** ($t \rhd e$): the workflow is similar to the step combinator, but in order to send the task value to the expression, the user needs to send a "continue" event.
2. **Parallel combinators**:
   (a) **Pair** ($t_1 \bowtie t_2$): *and* composition of two tasks that are worked on in parallel. The task value of this combinator is a pair of both task values.
   (b) **Choose** ($t_1 \blacklozenge t_2$): *or* composition of two tasks that are worked on in parallel. It picks the leftmost branch that has a value of awaited type and yields it as the result, the other branch is deleted.

Task evaluation is always driven by the input events in $\widehat{TOP}$. Tasks are typed, hence accepted inputs should be of a certain type as well. When the system receives an event, it passes this event to the running task, which then is reduced to a new task. Evaluation of the tasks that happen between interaction steps happens atomically with regard to inputs.

Tasks are modular: larger tasks can be composed of smaller ones; they are *first-class* which means that tasks can be used in functions as arguments and as a result. This way, custom collaboration patterns can be modelled.

Chapter 3

Semantics

Formal semantics specify the meaning or behaviour of software or hardware programs. Semantics describe the process of program execution: starting from the input state and ending in the output state, tracking the steps that the computer takes while executing the program. Semantics are used to reveal ambiguities or any possible complexities in defining documents and to create the basis for implementation, analysis and verification [13].

3.1 Formal semantics

There are three major approaches in formal semantics: denotational semantics, axiomatic semantics and operational semantics.

Denotational semantics describe what the effect of the program execution is. The effect of the program is the relationship between initial and final states and it is achieved through the definition of a semantic function for each syntactic category which maps them to mathematical objects.

Axiomatic semantics allows us to prove partial correctness properties of the program using a logical system (assertions).

Operational semantics describe how a program is interpreted as a sequence of computational steps. The main interest is *how* the effect of this computational sequence is produced. This semantics gives an abstraction of how the program is executed on the machine and does not provide any further information about the use of registers, addresses and other memory components for variables. It is independent from machine architecture and the ways the program is actually implemented [13].

3.2 TopHat semantics

The sections below describe the syntax and operational semantics of TopHat ($\widehat{TOP}$) language, which are used as the basis for the research. Soundness

and completeness of $\widehat{TOP}$ are out of scope, so they are only briefly mentioned in the last section. The syntax and semantics of $\widehat{TOP}$ are described in the paper *TopHat: A formal foundation for task-oriented programming* written by Tim Steenvoorden, Nico Naus and Marcus Klinik [4] is used in this work. Tim Steenvoorden's unpublished doctorate dissertation about the development and formalization of $\widehat{TOP}$ is a successor to this and other papers published about $\widehat{TOP}$. Tim Steenvoorden's thesis contains the most recent $\widehat{TOP}$ syntax and semantics, but due to the fact that it is still a work in progress, the decision was made to use the most stable version of the $\widehat{TOP}$ semantics. And thus, the *TopHat: A formal foundation for task-oriented programming* paper is used.

3.2.1 Syntax of TopHat

The host language is a simply typed λ-calculus that was extended with some basic types (boolean, integer and string) and ML style references. Figure 3.1 shows the syntax of the host language and $\widehat{TOP}$.

Expressions and constants define the syntax of the host language. The star symbol represents binary operators that are enumerated in a separate syntactic category. Pairs are used for defining parallelized task results, conditionals help with defining guards. The keyword **ref** yields a location l and helps to implement shared editors. The variable x is a program variable, the symbols ! and := represent dereferencing and assignment. A unit is used as the result of an assignment.

Pretasks are tasks with unevaluated subexpressions from the host language. There are open symbols ($\square, \boxtimes, \triangleright$) that indicate tasks that require user input and closed symbols ($\blacksquare, \blacktriangleright, \blacklozenge, \bowtie$) used for the tasks that are evaluated without interaction with the user. The shared editor accepts the user input in handling semantics, but its value changes at the specified location l. Expression e in the shared editor pretask is evaluated to the location l in the evaluation semantics described in Section 3.2.2.

The external choice combinator ($\Diamond$) was left out of the description of the syntax and semantics of the language, because it is not used in the latest versions of the language due to its needlessness, and it was discovered that its semantics contained errors. Moreover, mTask does not contain any similar combinator, thus, omitting the external choice semantics will not influence the semantics of mTask.

In addition, the filling of the combination combinator ($\bowtie$) was changed in this thesis, because it should be closed symbol as no user input is required. In the foundation paper for $\widehat{TOP}$ [4] the symbol $\bowtie$ was used for this combinator, but this appears to be a mistake. It was changed by Tim Steenvorden in his doctorate dissertation [1] to the filled symbol, which is used in this bachelor thesis.

$e ::=$		*Expressions*
	$\lambda x : \tau.e \mid e_1\ e_2$	abstraction, application
$\mid$	$x \mid c \mid e_1 \star e_1$	variable, constant, operation
$\mid$	**if** e_1 **then** e_2 **else** $e_3 \mid \langle\rangle$	branch, unit
$\mid$	$\langle e_1,\ e_2\rangle \mid \text{fst}\ e \mid \text{snd}\ e$	pair, projections
$\mid$	**ref** $e \mid !e \mid e_1 := e_2 \mid l$	references, location
$\mid$	p	pretask
$c ::=$	$B \mid I \mid S$	*Constants*: boolean, integer, string
$\star ::=$		*Binary operations*
	$< \mid \leq \mid \equiv \mid \not\equiv \mid \geq \mid >$	equational
$\mid$	$+ \mid - \mid \times$	numerical
$\mid$	$\wedge \mid \vee$	conjunction, disjunction
$\mid$	$+\!\!+$	append
$p ::=$		*Pretasks*
	$\square\ e \mid \boxtimes\,\tau \mid \blacksquare\ e$	editors: valued, unvalued, shared
$\mid$	$e_1 \blacktriangleright e_2 \mid e_1 \triangleright e_2$	steps: internal, external
$\mid$	$\lightning \mid e_1 \bowtie e_2$	fail, combination
$\mid$	$e_1 \blacklozenge e_2$	choice: internal
$\tau ::=$		*Types*
	$\tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \beta$	function, product, basic
$\mid$	$\texttt{Unit} \mid \texttt{Ref}\ \tau \mid \texttt{Task}\ \tau$	unit, reference, task
$\beta ::=$	$\texttt{Bool} \mid \texttt{Int} \mid \texttt{String}$	*Basic types*: boolean, integer, string

Figure 3.1: Syntax of expressions, pretasks and types in TopHat.

Typing rules have the form $\Gamma, \Sigma \vdash e : \tau$ and are read as *in environment* Γ *and store typing* Σ *the expression* e *has type* τ. The typing rules in $\widehat{TOP}$ are shown in Figure 3.2. All the typing rules described in the formal foundation paper [4] are present in the figure except for constants, binary operators and units, which typing rules are trivial and thus were omitted in the paper and in Figure 3.2, in consequence.

$$\boxed{\Gamma, \Sigma \vdash e : \tau}$$

Pretasks

T-Edit
$$\frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \square\, e : \texttt{Task}\ \tau}$$

T-Enter
$$\frac{}{\Gamma, \Sigma \vdash \boxtimes \tau : \texttt{Task}\ \tau}$$

T-Update
$$\frac{\Gamma, \Sigma \vdash e : \texttt{Ref}\ \beta}{\Gamma, \Sigma \vdash \blacksquare\, e : \texttt{Task}\ \beta}$$

T-Then
$$\frac{\Gamma, \Sigma \vdash e_1 : \texttt{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \texttt{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \blacktriangleright e_2 : \texttt{Task}\ \tau_2}$$

T-Next
$$\frac{\Gamma, \Sigma \vdash e_1 : \texttt{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \texttt{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \triangleright e_2 : \texttt{Task}\ \tau_2}$$

T-Fail
$$\frac{}{\Gamma, \Sigma \vdash \lightning : \texttt{Task}\ \tau}$$

T-Or
$$\frac{\Gamma, \Sigma \vdash e_1 : \texttt{Task}\ \tau \quad \Gamma, \Sigma \vdash e_2 : \texttt{Task}\ \tau}{\Gamma, \Sigma \vdash e_1 \blacklozenge e_2 : \texttt{Task}\ \tau}$$

T-And
$$\frac{\Gamma, \Sigma \vdash e_1 : \texttt{Task}\ \tau_1 \qquad \Gamma, \Sigma \vdash e_2 : \texttt{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \bowtie e_2 : \texttt{Task}\ (\tau_1 \times \tau_2)}$$

Expressions

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

T-Loc
$$\frac{\Sigma(l) = \beta}{\Gamma, \Sigma \vdash l : \texttt{Ref}\ \beta}$$

T-Ref
$$\frac{\Gamma, \Sigma \vdash e : \beta}{\Gamma, \Sigma \vdash \textbf{ref}\ e : \texttt{Ref}\ \beta}$$

T-Deref
$$\frac{\Gamma, \Sigma \vdash e : \texttt{Ref}\ \beta}{\Gamma, \Sigma \vdash !e : \beta}$$

T-Assign
$$\frac{\Gamma, \Sigma \vdash e_1 : \texttt{Ref}\ \beta \quad \Gamma, \Sigma \vdash e_2 : \beta}{\Gamma, \Sigma \vdash e_1 := e_2 : \texttt{Unit}}$$

T-Pair
$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2}{\Gamma, \Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

T-App
$$\frac{\Gamma[x : \tau_1], \Sigma \vdash e : \tau_2}{\Gamma, \Sigma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$$

T-Abs
$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma, \Sigma \vdash e_2 : \tau_1}{\Gamma, \Sigma \vdash e_1\ e_2 : \tau_2}$$

T-If
$$\frac{\Gamma, \Sigma \vdash e_1 : \texttt{Bool} \quad \Gamma, \Sigma \vdash e_2 : \tau \quad \Gamma, \Sigma \vdash e_3 : \tau}{\Gamma, \Sigma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}$$

Figure 3.2: Typing rules in TopHat.

3.2.2 Semantics of TopHat

Since the $\widehat{TOP}$'s task language is embedded in a simply-typed λ-calculus, it requires the specification of the terms *evaluation* in the host language and the way it handles the task language. It can be done in two ways, internally by the system and externally by the user. As a result, two additional semantics are used: *normalisation* (internal normalisation of the task) and *interaction* (external interaction with the user).

The authors use rightward arrows to represent small-step semantics and downward arrows for big-step semantics. The three resulting layers of semantics are evaluation ($\downarrow$), normalisation ($\Downarrow$) and interaction ($\Rightarrow$). There are two helper semantics: stride ($\rightsquigarrow$), used for normalisation, and handle ($\rightarrow$), used for interaction. Figure 3.3 shows how different semantics are related.

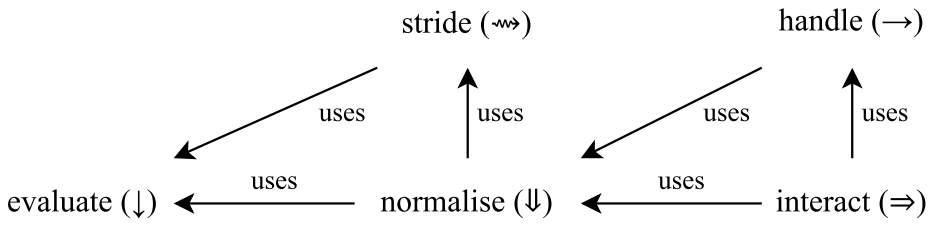

Figure 3.3: Relations of semantics in TopHat. The image was copied from the paper of $\widehat{TOP}$ formal foundation [4].

Figure 3.4 provides an example of how different layers of semantics are connected using simple program. The interaction semantics handles inputs only for normalised tasks. Thus, the first semantics used in the program is normalisation. The normalisation semantics is a big-step semantics that consists of evaluation, the big-step reduction of expressions, and stride, the internal reduction of tasks. Only when the task is normalised, the input is handled by the handling semantics and the resulting task is normalised again for the possible next input.

$$
\begin{array}{ll}
 & \Box(5+1) \blacktriangleright \lambda x.\textbf{if } x = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
\downarrow & \Box 6 \blacktriangleright \lambda x.\textbf{if } x = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Normalisation I} : \textit{evaluate} \\
\rightsquigarrow & \Box 6 \blacktriangleright \lambda x.\textbf{if } x = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Normalisation I} : \textit{stride step} \\
\downarrow & \textbf{if } 6 = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Stride step} : \textit{evaluate I} \\
\downarrow & \textbf{if } \texttt{False} \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Stride step} : \textit{evaluate II} \\
\downarrow & \lightning \qquad \textit{Stride step} : \textit{evaluate III} \\
\xrightarrow{42} & \Box 42 \blacktriangleright \lambda x.\textbf{if } x = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Interaction} : \textit{ handle} \\
\downarrow & \Box 42 \blacktriangleright \lambda x.\textbf{if } x = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Normalisation II} : \textit{evaluate} \\
\rightsquigarrow & \Box 42 \blacktriangleright \lambda x.\textbf{if } x = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Normalisation II} : \textit{stride step} \\
\downarrow & \textbf{if } 42 = 42 \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Stride step} : \textit{evaluate I} \\
\downarrow & \textbf{if } \texttt{True} \textbf{ then } (\Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int}) \textbf{ else } \lightning \\
 & \qquad \textit{Stride step} : \textit{evaluate II} \\
\downarrow & \Box(7 \times 6) \blacklozenge \boxtimes \texttt{Int} \quad \textit{Stride step} : \textit{evaluate III} \\
\downarrow & \Box 42 \blacklozenge \boxtimes \texttt{Int} \qquad \textit{Stride step} : \textit{evaluate IV} \\
\downarrow & \Box 42 \blacklozenge \boxtimes \texttt{Int} \qquad \textit{Normalisation III} : \textit{evaluate} \\
\rightsquigarrow & \Box 42 \qquad\qquad\qquad \textit{Normalisation III} : \textit{stride}
\end{array}
$$

Figure 3.4: Semantics example for TopHat.

Evaluation

Big-step semantics and a call-by-value strategy are used for evaluation. The rule $e, \sigma \downarrow v, \sigma'$ means that *the expression e in state σ is evaluated to a value v in state σ'*. Figure 3.5 shows the value grammar in evaluation semantics of $\widehat{TOP}$ Tasks are values, and their constructors' operands are evaluated eagerly.

$v ::=$		*Values*
|	$\lambda x : \tau.e \mid \langle v_1,\ v_2 \rangle \mid \langle\ \rangle$	abstraction, pair, unit
|	$c \mid l \mid t$	constant, location, task
$t ::=$		*Tasks*
|	$\square\, v \mid \boxtimes \tau \mid \blacksquare\, l$	editors
|	$t_1 \blacktriangleright e_2 \mid t_1 \rhd e_2$	steps
|	$\lightning \mid t_1 \bowtie t_2 \mid t_1 \blacklozenge t_2$	fail, combination, choice

Figure 3.5: Evaluation value grammar in TopHat.

The evaluation rules for expressions e are standard, except for the task constructs. Task evaluation rules can be deduced from the value grammar. The steps' right-hand sides stay unevaluated, because their evaluation requires the task value of the left-hand side [4]. All the evaluation rules that were provided in the formal foundation paper [4] are given in Figure 3.6.

$$\boxed{e, \sigma \downarrow v, \sigma'}$$

E-App

$$\frac{e_1, \sigma \downarrow \lambda x \ : \ \tau.e_1', \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma'' \quad e_1'[x \mapsto v_2], \sigma'' \downarrow v_1, \sigma'''}{e_1 e_2, \sigma \downarrow v_1, \sigma'''}$$

E-IfTrue

$$\frac{e_1, \sigma \downarrow \texttt{True}, \sigma' \quad e_2, \sigma' \downarrow v, \sigma''}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \sigma \downarrow v, \sigma''}$$

E-IfFalse

$$\frac{e_1, \sigma \downarrow \texttt{False}, \sigma' \quad e_3, \sigma' \downarrow v, \sigma''}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \sigma \downarrow v, \sigma''}$$

E-Ref

$$\frac{e, \sigma \downarrow v, \sigma' \quad l \notin Dom(\sigma')^1}{\textbf{ref } e, \sigma \downarrow v, \sigma'[l \mapsto v]}$$

E-Deref

$$\frac{e, \sigma \downarrow l, \sigma'}{!e, \sigma \downarrow \sigma'(l), \sigma'}$$

E-Assign

$$\frac{e_1, \sigma \downarrow l, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{e_1 := e_2, \sigma \downarrow \langle\rangle, \sigma''[l \mapsto v_2]}$$

E-Value

$$\frac{}{v, \sigma \downarrow v, \sigma}$$

E-Edit

$$\frac{e, \sigma \downarrow v, \sigma'}{\square\, e, \sigma \downarrow \square\, v, \sigma'}$$

E-Enter

$$\frac{}{\boxtimes\, \tau, \sigma \downarrow \boxtimes\, \tau, \sigma}$$

E-Update

$$\frac{e, \sigma \downarrow l, \sigma'}{\blacksquare\, e, \sigma \downarrow \blacksquare\, l, \sigma'}$$

E-Then

$$\frac{e_1, \sigma \downarrow t_1, \sigma'}{e_1 \blacktriangleright e_2, \sigma \downarrow t_1 \blacktriangleright e_2, \sigma'}$$

E-Next

$$\frac{e_1, \sigma \downarrow t_1, \sigma'}{e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'}$$

E-Fail

$$\frac{}{\lightning, \sigma \downarrow \lightning, \sigma}$$

E-Pair

$$\frac{e_1, \sigma \downarrow v_1, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{\langle e_1, e_2 \rangle, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma''}$$

E-Or

$$\frac{e_1, \sigma \downarrow t_1, \sigma' \quad e_2, \sigma' \downarrow t_2, \sigma''}{e_1 \blacklozenge e_2, \sigma \downarrow t_1 \blacklozenge t_2, \sigma''}$$

E-And

$$\frac{e_1, \sigma \downarrow t_1, \sigma' \quad e_2, \sigma' \downarrow t_2, \sigma''}{e_1 \bowtie e_2, \sigma \downarrow t_1 \bowtie t_2, \sigma''}$$

Figure 3.6: Evaluation rules in TopHat.

[1] Location l is not in the domain of σ'

Task observations

Observations on tasks are used in both normalisation and interaction semantics. Observations are functions on the syntax tree of the task. The partial function *observable value* $\mathcal{V}$ associates a value v to tasks t where possible. The function *failing* $\mathcal{F}$ determines whether task t is failing: if the function with task t in state σ returns `True`, then the task is failing. The definitions of these functions are given in Figure 3.7.

In the observable value function, $\perp$ represents no value and $\sigma(l)$ is a value of a shared editor at location l in state σ. Both functions use evaluated expressions as their arguments.

Striding semantics

The striding semantics is mainly used to rewrite the combinators into their new reduced form. It is used in normalisation semantics described in Section 3.2.2. In striding semantics, the rule $t, \sigma \rightsquigarrow t', \sigma'$ means that *task* t *in state* σ *reduces to task* t' *in state* σ'. Tasks like editors and fail are reduced to themselves. The striding rules are shown in Figure 3.8.

$$\mathcal{V} : Tasks \times States \rightharpoonup Values$$

$$\begin{aligned}
\mathcal{V}(\square\, v, \sigma) &= v \\
\mathcal{V}(\boxtimes\, \tau, \sigma) &= \bot \\
\mathcal{V}(\blacksquare\, l, \sigma) &= \sigma(l) \\
\mathcal{V}(\lightning, \sigma) &= \bot \\
\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) &= \bot \\
\mathcal{V}(t_1 \rhd e_2, \sigma) &= \bot \\
\mathcal{V}(t_1 \bowtie t_2, \sigma) &= \begin{cases} \langle v_1,\ v_2 \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
\mathcal{V}(t_1 \blacklozenge t_2, \sigma) &= \begin{cases} v_1 & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \\ v_2 & \textbf{when } \mathcal{V}(t_1, \sigma) = \bot \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases}
\end{aligned}$$

$$\mathcal{F} : Tasks \times States \to Booleans$$

$$\begin{aligned}
\mathcal{F}(\square\, v, \sigma) &= \texttt{False} \\
\mathcal{F}(\boxtimes\, \tau, \sigma) &= \texttt{False} \\
\mathcal{F}(\blacksquare\, l, \sigma) &= \texttt{False} \\
\mathcal{F}(\lightning, \sigma) &= \texttt{True} \\
\mathcal{F}(t_1 \blacktriangleright e_2, \sigma) &= \mathcal{F}(t_1, \sigma) \\
\mathcal{F}(t_1 \rhd e_2, \sigma) &= \mathcal{F}(t_1, \sigma) \\
\mathcal{F}(t_1 \bowtie t_2, \sigma) &= \mathcal{F}(t_1, \sigma) \wedge \mathcal{F}(t_2, \sigma) \\
\mathcal{F}(t_1 \blacklozenge t_2, \sigma) &= \mathcal{F}(t_1, \sigma) \wedge \mathcal{F}(t_2, \sigma)
\end{aligned}$$

Figure 3.7: Observable values and failing in $\widehat{TOP}$.

$$\boxed{t, \sigma \rightsquigarrow t', \sigma'}$$

Step

S-ThenStay

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma'}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow t_1' \blacktriangleright e_2, \sigma'} \qquad \mathcal{V}(t_1', \sigma') = \bot$$

S-ThenFail

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad e_2\ v_1, \sigma' \downarrow t_2, \sigma''}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow t_1' \blacktriangleright e_2, \sigma'} \qquad \mathcal{V}(t_1', \sigma') = v_1 \wedge \mathcal{F}(t_2, \sigma'')$$

S-ThenCont

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad e_2\ v_1, \sigma' \downarrow t_2, \sigma''}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow t_2, \sigma''} \qquad \mathcal{V}(t_1', \sigma') = v_1 \wedge \neg\mathcal{F}(t_2, \sigma'')$$

Choose

S-OrLeft

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma'}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow t_1', \sigma'} \qquad \mathcal{V}(t_1', \sigma') = v_1$$

S-OrRight

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow t_2', \sigma''} \qquad \mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = v_2$$

S-OrNone

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow t_1' \blacklozenge t_2', \sigma''} \qquad \mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = \bot$$

Ready

S-Edit

$$\frac{}{\square\ v, \sigma \rightsquigarrow \square\ v, \sigma}$$

S-Fill

$$\frac{}{\boxtimes\tau, \sigma \rightsquigarrow \boxtimes\tau, \sigma}$$

S-Update

$$\frac{}{\blacksquare\ l, \sigma \rightsquigarrow \blacksquare\ l, \sigma}$$

S-Fail

$$\frac{}{\lightning, \sigma \rightsquigarrow \lightning, \sigma}$$

Congruence

S-Next

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma'}{t_1 \rhd e_2, \sigma \rightsquigarrow t_1' \rhd e_2, \sigma}$$

S-And

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \bowtie t_2, \sigma \rightsquigarrow t_1' \bowtie t_2', \sigma}$$

Figure 3.8: Striding semantics in $\widehat{TOP}$.

Normalising tasks

The normalisation semantics reduce expressions of the type `Task` until they can handle an input. It is a big-step semantics and it makes use of the host language. The rule $e, \sigma \Downarrow t, \sigma'$ means that *the expression* e *in state* σ *normalises to task* t *in state* σ'.

The normalisation rules are shown in Figure 3.9. All rules ensure that expressions are first evaluated by the host language $\downarrow$ and then reduced by the stride semantics $\rightsquigarrow$. These actions are repeated in the rule `N-Repeat` to ensure that the final state and the task stabilise.

$$\boxed{e, \sigma \Downarrow t, \sigma'}$$

`N-Done`

$$\frac{e, \sigma \downarrow t, \sigma' \quad t, \sigma' \rightsquigarrow t', \sigma''}{e, \sigma \Downarrow t, \sigma'} \qquad \sigma' = \sigma'' \wedge t = t'$$

`N-Repeat`

$$\frac{e, \sigma \downarrow t, \sigma' \quad t, \sigma' \rightsquigarrow t', \sigma'' \quad t', \sigma'' \Downarrow t'', \sigma'''}{e, \sigma \Downarrow t'', \sigma'''} \qquad \sigma' \neq \sigma'' \wedge t \neq t'$$

Figure 3.9: Normalisation semantics in $\widehat{TOP}$.

Handling and interaction semantics

The handling semantics is the outermost layer of the semantics' stack. It is responsible for external steps and for changing editors' values. Interaction semantics is defined by the use of handling semantics. Only normalised task t are used in interaction semantics. When the input event i occurs, it is handled for the task t, resulting in a new task t' that is then prepared (using normalising semantics) for the next input as shown in Figure 3.10.

$$\boxed{t, \sigma \overset{i}{\Rightarrow} t', \sigma'}$$

`I-Handle`

$$\frac{t, \sigma \overset{i}{\rightarrow} t', \sigma' \quad t', \sigma' \Downarrow t'', \sigma''}{t, \sigma \overset{i}{\Rightarrow} t'', \sigma''}$$

Figure 3.10: Interaction semantics in $\widehat{TOP}$.

The input grammar is described in Figure 3.11. `F` (first) and `S` (second) encode the paths to the task to which input should be passed. Actions can be either a value or a "continue" action.

$i ::=$	$a \mid \texttt{F}\ i \mid \texttt{S}\ i$	*Input* : action, pass to first, pass to second
$a ::=$	$v \mid \texttt{C}$	*Action* : change, continue

Figure 3.11: Input grammar in $\widehat{TOP}$.

The function $\mathcal{I}$ takes a normalised task and a state as an input and based on them calculates a set of input events that the given task expects. The function is used in the proofs of soundness of $\widehat{TOP}$, which are not covered in this thesis. Thus, the function $\mathcal{I}$ was added to Appendix A.1.

Input handling is done using the handling semantics that is described in Figure 3.12. It is a small-step semantics where transitions are labelled. A task t in a state σ and an input i are taken to yield a new task t' in a new state σ'.

$$\boxed{t, \sigma \xrightarrow{i} t', \sigma'}$$

Editing

H-Change

$$\frac{}{\square\, v, \sigma \xrightarrow{v'} \square\, v', \sigma}\; v, v' : \tau$$

H-Fill

$$\frac{}{\boxtimes\, \tau, \sigma \xrightarrow{v'} \square\, v', \sigma}\; v' : \tau$$

H-Update

$$\frac{}{\blacksquare\, l, \sigma \xrightarrow{v'} \blacksquare\, l, \sigma[l \mapsto v']}\; \sigma(l), v' : \tau$$

Continuing

H-Next

$$\frac{e_2\ v_1, \sigma \Downarrow t_2, \sigma'}{t_1 \rhd e_2, \sigma \xrightarrow{\mathtt{C}} t_2, \sigma}\; \mathcal{V}(t_1, \sigma) = v_1 \wedge \neg \mathcal{F}(t_2, \sigma')$$

Passing

H-PassThen

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \blacktriangleright e_2, \sigma \xrightarrow{i} t_1' \blacktriangleright e_2, \sigma'}$$

H-FirstAnd

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathtt{F}\ i} t_1' \bowtie t_2, \sigma'}$$

H-FirstOr

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathtt{F}\ i} t_1' \blacklozenge t_2, \sigma'}$$

H-PassNext

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \triangleright e_2, \sigma \xrightarrow{i \neq \mathtt{C}} t_1' \triangleright e_2, \sigma'}$$

H-SecondAnd

$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathtt{S}\ i} t_1 \bowtie t_2', \sigma'}$$

H-SecondOr

$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathtt{S}\ i} t_1 \blacklozenge t_2', \sigma'}$$

Figure 3.12: Handling semantics in $\widehat{TOP}$.

3.2.3 Reference to Soundness and completeness of inputs

To prove that the semantics of $\widehat{TOP}$ is sound, the authors showed that evaluation, normalisation and handling are type preserving. They showed that the failing function $\mathcal{F}$ indicated steps that cannot be normalised correctly and proved that the function computing all possible inputs is sound and complete.

These proofs do not influence the creation of the mTask semantics and prove only how well the semantics of $\widehat{TOP}$ is defined. These proofs can later be repeated for the mTask semantics, but soundness and completeness is outside of the scope of this thesis. Therefore, proofs are left out of this description of $\widehat{TOP}$ semantics. All the theorems and proofs are accessible in the paper of TopHat's formal foundation [4].

Chapter 4

mTask semantics

This chapter focuses on the mTask language's syntax and semantics. The mTask language is a TOP language that has an implementation in Clean and is designed for programming IoT devices. The syntax, types and typing rules are described in Section 4.1. The section focuses on the functions and expressions in the host language—an enriched λ-calculus—and pretasks in mTask. Section 4.2 describes the task language constructs in more detail. Section 4.3 covers evaluation, normalisation, striding, interaction and handling semantics in the form of operational semantics of mTask and provides the rules for all the language constructs.

4.1 Syntax

In this section, the mTask constructs are described. For formalisation, mTask is embedded in the λ-calculus, which is extended with some constructs. Before defining the semantics, it is important to show the syntax of both the host and mTask language. In addition, the types used in the language and typing rules for all constructs are provided at the end of this section.

4.1.1 Constants and binary operators

Expressions contain constants and binary operators, the syntax for which is defined in Figure 4.1. Constants are members of one of the following sets: booleans, integers, real numbers. Binary operators are operators used in expressions and they describe operations on two values, such as comparison, numerical or logic operators.

$c ::=$	$B \mid I \mid R$	*Constant*
		boolean, integer, real number
$\star ::=$		*Binary operator*
	$< \mid <= \mid ==$	less, less or equal, equal
$\mid$	$!= \mid >= \mid >$	not equal, more or equal, more
$\mid$	$+ \mid - \mid * \mid /$	numerical
$\mid$	$\wedge \mid \vee$	conjunction, disjunction

Figure 4.1: Syntax of constants and binary operators in mTask.

4.1.2 Expressions

The task language is embedded in a simply-typed λ-calculus, which is extended with some constructs that are described below.

One such construct is Λ, which defines the body of the function, or sets the initial value of SDS, or sets the physical location to the sensor task. SDSs and sensors are described in Section 4.2.2 and Section 4.2.1 respectively. Compared to $\widehat{TOP}$, mTask makes sure that functions, locations and sensors are only defined at the top level and are always fully applied. Thus, they are separated into their own category—$Main$. In the definition of the function, x_{f} is the function itself and $\overline{x}_n$ are arguments, where n is a natural positive number indicating the number of the arguments. Sensors are assigned some abstract location μ. It is an abstract description of the physical location of the sensor (for example, a pin on the micro controller). Locations are assigned with some expression e that holds the initial value of the location.

Moreover, the host language consists of variables, constants, applications, unary and binary operators, conditionals, pretasks, locations (used in SDS) and sensors. Tuple and their projections are used as results of observations on tasks. An application is used to apply x on the parameters $\overline{e}_n$, where n is a positive natural number representing the number of arguments.

Λ ::=	**let** $x_\mathrm{f}\ \overline{x}_n = e$ **in** Λ	$Main$: function
|	**sds** $l = e$ **in** Λ	SDS
|	**sns** $s = \mu$ **in** Λ	sensor
|	e	
e ::=		*Expression*
|	$x \mid c \mid x_\mathrm{f}\,\overline{e}_n$	variable, constant, application
|	$\neg\, e \mid e_1 \star e_1 \mid \langle\,\rangle$	negation, binary operation, unit
|	**if** e_1 **then** e_2 **else** e_3	condition
|	$\langle e_1, e_2 \rangle$ | **fst** e | **snd** e	2-tuple, projections
|	$p \mid l \mid s$	pretask, location, sensor

Figure 4.2: Syntax of function, SDS and sensor definitions and expressions in mTask.

4.1.3 Pretasks

Pretasks are constructs of the language that describe tasks with unevaluated subexpressions with respect to the host language. When a pretask and all its subexpressions are evaluated, it is called a task. Evaluation is a part of the evaluation semantics and is discussed in Section 4.3.1.

The task language includes sensors, which are discussed in detail in Section 4.2.1. The language also includes a fail pretask, which indicates a task that never has a value and never accepts input. An important aspect of the fail pretask is that the implementation of mTask does not have such a construct and tasks cannot be programmed to end in a fail state. However, fail is important in the semantics and is used to indicate states of the program that should never be reached. Thus, fail is added to the pretask category.

Three pretasks—**getSds**, **setSds** and **updSds**—are used to read the value, set a new value or change the value of a specified SDS. In $\widehat{TOP}$, referencing, value assignment and dereferencing of SDS are part of the host language syntax. In contrast, mTask has **getSds**, **setSds** and **updSds** in its task language.

For describing the formal foundation of $\widehat{TOP}$, the paper *TopHat: A formal foundation for task-oriented programming* was taken [4]. It did not contain changes that were made later to the $\widehat{TOP}$ syntax and semantics. The newest version of $\widehat{TOP}$ formal foundation is described in Tim Steenvoorden's unpublished doctorate dissertation [1], and it contains some interesting editors and combinators that are worth mentioning. For example, Tim Steenvoorden introduced the valued read-only editor ($\boxdot\, e$), which has an analogue in mTask in the form of **rtrn** and **unstable** pretasks. How-

ever, in mTask **rtrn** and **unstable** can have tasks as their values, while the read-only editor can only have values. Both **rtrn** and **unstable** emit the resulting value of the expression, which is stable or unstable respectively. $\widehat{TOP}$ does not have a notion of stability.

Moreover, Tim Steenvoorden introduced the repeat combinator ($\curvearrowleft t$) that constantly repeats the given evaluated task t. The mTask language has a similar **rpeat** e pretask. When expression e is evaluated into task t, we get the task **rpeat** t. Defining semantics for this task turned out to be a complicated task.

The mTask language also contains a **delay** e pretask, that makes the system wait for a time specified by the expression e. This task uses the notion of changing time, modelling of which makes creation of language semantics way harder. It was decided to leave this task out of scope due to time constraints.

The pretask language also includes combinators; it has two parallel and one sequential combinator. When a task value is observed, the parallel combinator && returns both task results combined into a 2-tuple (pair), while || yields only one of the task results. The parallel || combinator is similar to the choice (⧫) combinator from $\widehat{TOP}$. The sequential step combinator is the only sequential combinator in mTask. It is similar to the internal step (▶) combinator from $\widehat{TOP}$, but there are major differences in the evaluation semantics, described in Section 4.3.1. The mTask step combinator uses continuations that calculate the right-hand side of the step combinator depending on the task value of the left-hand side. All mentioned combinators are described in more detail in Section 4.2.4.

$p ::=$		*Pretask*
	$\circledast\, s \mid \lightning$	sensor, fail
|	**getSds** l | **setSds** $l\ e$ | **updSds** $l\ (\lambda x.e)$	get, set, update SDS
|	**rtrn** e | **unstable** e	return stable, unstable
|	e_1 && e_2 | e_1 || e_2	parallel: and, or
|	$e_1 \gg\!* \ [\overline{c_l},]$	sequential step
$c_l ::=$	**OnValue** $(\lambda x.e_1)\ (\lambda x.e_2)$	*Continuation* : valued
|	**OnStable** $(\lambda x.e_1)\ (\lambda x.e_2)$	stable
|	**OnUnstable** $(\lambda x.e_1)\ (\lambda x.e_2)$	unstable

Figure 4.3: Syntax of pretasks and continuations in mTask.

4.1.4 Types and typing rules

Figure 4.4 shows the grammar of types used in mTask. The type β contains basic types: boolean, integer, real. The type τ describes types of pair, unit

and basic type β. The type ρ describes the function result, which can be either $Value$ or $Task$. The type f is used to describe a function that takes n number of arguments ($\overline{\tau}_n$) of the same type τ and returns the type ρ. The type *sds* is used to indicate SDS and the type *sens* is used for sensors.

$\beta ::=$	`Bool` | `Int` | `Real`	*Basic type* : boolean, integer, Real
$\tau ::=$	$\tau_1 \times \tau_2 \mid \langle\rangle \mid \beta$	$Value$: pair, unit, basic type
$\psi ::=$	`MTask` τ	$Task$
$\rho ::=$	$\tau \mid \psi$	*Function result*: value, task
$f ::=$	$\overline{\tau}_n \rightarrow \rho$	*Function*
$sds ::=$	`SDS` τ	*Shared Data Source*
$sens ::=$	`Sensor` τ	*Sensor*

Figure 4.4: Types in mTask.

All typing rules are of the form $\Gamma, \Sigma \vdash e : \tau$ and, as in $\widehat{TOP}$, are read as *in the environment* Γ *and store typing* Σ*, expression* e *has type* τ. Environment Γ is used to store types of variables and functions, and store typing Σ is used to store the types of locations and sensors.

Figure 4.5 contains typing rules for all main constructs and expressions. Environment Γ and store location Σ are actively used for storing types of variables, functions and values that are used in both typing categories. The notion $\Sigma[l : \texttt{SDS}\ \tau]$ means that the location l of the type `SDS` τ is stored in the store location Σ. Similar notions are used for sensors in Σ and variables and functions in Γ. The stored types are used in expressions to check that the type is known for the environment Γ or store location Σ. The notion $\mu : \tau$ is a guard that checks that the abstract sensor location is of the type τ. Moreover, $\star : \tau_1\ \tau_1 \rightarrow \tau_2$ is also a guard that checks that the star-function takes two arguments of the type τ_1 and returns a result of the type τ_2.

The bar above the arguments in the rule `T-App` means that there are n arguments of the types $\tau_1 ... \tau_n$.

Function, location and sensor definition

T-LetFunc

$$\frac{\Gamma[\overline{x}_n : \overline{\tau}_n, x_f : \overline{\tau}_n \to \rho_1], \Sigma \vdash e : \rho_1 \quad \Gamma[x_f : \overline{\tau}_n \to \rho_1], \Sigma \ \vdash \ \Lambda : \rho_2}{\Gamma, \Sigma \vdash \textbf{let}\ x_\mathrm{f}\ \overline{x}_n = e\ \textbf{in}\ \Lambda\ :\ \rho_2}$$

T-LetLoc

$$\frac{\Gamma, \Sigma \vdash e : \tau \quad \Gamma, \Sigma \vdash l : \texttt{SDS}\ \tau \quad \Gamma, \Sigma[l : \texttt{SDS}\ \tau] \ \vdash \ \Lambda : \rho}{\Gamma, \Sigma \vdash \textbf{sds}\ l = e\ \textbf{in}\ \Lambda\ :\ \rho}$$

T-LetSensor

$$\frac{\mu : \tau \quad \Gamma, \Sigma \vdash s : \texttt{Sensor}\ \tau \quad \Gamma, \Sigma[s : \texttt{Sensor}\ \tau] \ \vdash \ \Lambda : \rho}{\Gamma, \Sigma \vdash \textbf{sns}\ s = \mu\ \textbf{in}\ \Lambda\ :\ \rho}$$

Expressions

T-Var

$$\frac{x : \tau \in \Gamma}{\Gamma, \Sigma \vdash x : \tau}$$

T-Const

$$\frac{c \in \beta}{\Gamma, \Sigma \vdash c : \beta}$$

T-Unit

$$\frac{}{\Gamma, \Sigma \vdash \langle\,\rangle : \texttt{Unit}}$$

T-Pair

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2}{\Gamma, \Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

T-Loc

$$\frac{l : \texttt{SDS}\ \tau \in \Sigma}{\Gamma, \Sigma \vdash l : \texttt{SDS}\ \tau}$$

T-Sensor

$$\frac{s : \texttt{Sensor}\ \tau \in \Sigma}{\Gamma, \Sigma \vdash s : \texttt{Sensor}\ \tau}$$

T-Neg

$$\frac{\Gamma, \Sigma \vdash e : \texttt{Bool}}{\Gamma, \Sigma \vdash \neg e : \texttt{Bool}}$$

T-If

$$\frac{\Gamma, \Sigma \vdash e_1 : \texttt{Bool} \quad \Gamma, \Sigma \vdash e_2 : \rho \quad \Gamma, \Sigma \vdash e_3 : \rho}{\Gamma, \Sigma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \rho}$$

T-App

$$\frac{\Gamma, \Sigma \vdash \overline{e}_n : \overline{\tau}_n \quad x_\mathrm{f} : \overline{\tau}_n \to \rho \in \Gamma}{\Gamma, \Sigma \vdash x_\mathrm{f}\ \overline{e}_n : \rho}$$

T-Binary

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \quad \star : \tau_1\ \tau_1 \to \tau_2}{\Gamma, \Sigma \vdash e_1 \star e_2 : \tau_2}$$

Figure 4.5: Definitions and expressions typing rules in mTask.

Figure 4.6 describes the typing rules for continuations. All continuations have the same typing rules, thus, symbol $\ggg$ indicates all of them: **OnValue**, **OnStable** or **OnUnstable**. Continuations receive an argument of the type τ_1 and they contain two functions: the first calculates if the continuation is matched and returns the `Bool` value and the second one calculates the new task of type `MTask` τ_2 from the argument it received.

`T-Continuation`

$$\frac{\Gamma[x : \tau_1], \Sigma \vdash e_1 : \texttt{Bool} \quad \Gamma[x : \tau_1], \Sigma \vdash e_2 : \texttt{MTask } \tau_2}{\Gamma, \Sigma \vdash \ggg (\lambda x.e_1)\ (\lambda x.e_2) : \tau_1 \rightarrow \texttt{MTask } \tau_2}$$

Figure 4.6: Continuations typing rules in mTask.

Figure 4.7 shows the typing rules of pretasks. Most of the pretasks are of the type `MTask` τ, except for the parallel combinator of the type `MTask` $(\tau_1 \times \tau_2)$. The figure includes typing rules for sensors, fail tasks, all SDS constructs, return, unstable return and combinators.

Step combinator uses list notation for continuations and $[\overline{c_l},]$ represents the list of such continuations. The rule `T-Step` shows that all of these continuations have the type $\tau_1 \rightarrow$ `MTask` τ_2, which corresponds to the `T-Continuation` rule.

$$\boxed{\Gamma, \Sigma \vdash e : \tau}$$

Sensor, fail, SDS

$$\texttt{T-Sensor}\ \frac{\Gamma, \Sigma \vdash s : \texttt{Sensor}\ \tau}{\Gamma, \Sigma \vdash \circledast\, s : \texttt{MTask}\ \tau} \qquad \texttt{T-Fail}\ \frac{}{\Gamma, \Sigma \vdash \lightning : \texttt{MTask}\ \tau} \qquad \texttt{T-Get}\ \frac{\Gamma, \Sigma \vdash l : \texttt{SDS}\ \tau}{\Gamma, \Sigma \vdash \textbf{getSds}\ l : \texttt{MTask}\ \tau}$$

$$\texttt{T-Set}\ \frac{\Gamma, \Sigma \vdash l : \texttt{SDS}\ \tau \quad \Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \textbf{setSds}\ l\ e : \texttt{MTask}\ \tau} \qquad \texttt{T-Update}\ \frac{\Gamma, \Sigma \vdash l : \texttt{SDS}\ \tau \quad \Gamma[x : \tau], \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \textbf{updSds}\ l\ (\lambda x.e) : \texttt{MTask}\ \tau}$$

Return, unstable

$$\texttt{T-Return}\ \frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \textbf{rtrn}\ e : \texttt{MTask}\ \tau} \qquad \texttt{T-Unstable}\ \frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \textbf{unstable}\ e : \texttt{MTask}\ \tau}$$

Parallel and step combinator

$$\texttt{T-And}\ \frac{\Gamma, \Sigma \vdash e_1 : \texttt{MTask}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \texttt{MTask}\ \tau_2}{\Gamma, \Sigma \vdash e_1\ \&\!\&\ e_2 : \texttt{MTask}\ (\tau_1 \times \tau_2)}$$

$$\texttt{T-Or}\ \frac{\Gamma, \Sigma \vdash e_1 : \texttt{MTask}\ \tau \quad \Gamma, \Sigma \vdash e_2 : \texttt{MTask}\ \tau}{\Gamma, \Sigma \vdash e_1\ ||\ e_2 : \texttt{MTask}\ \tau}$$

$$\texttt{T-Step}\ \frac{\Gamma, \Sigma \vdash e_1 : \texttt{MTask}\ \tau_1 \quad \Gamma, \Sigma \vdash \overline{c}_l : \tau_1 \to \texttt{MTask}\ \tau_2}{\Gamma, \Sigma \vdash e_1\ \gg\!*\ [\overline{c_l},] : \texttt{MTask}\ \tau_2}$$

Figure 4.7: Pretasks typing rules in mTask.

4.2 Task language

This section provides a more detailed overview of the pretasks that the task language consists of.

4.2.1 Sensors

The domain of mTask lies in programming IoT systems. The IoT system inputs are different sensors. Sensors are, for example, temperature, humidity, IR distance, etc. sensors connected to the micro processor. Their outputs are processed in the system, which changes according to the value read from the sensor.

Sensors ($\circledast s$) as language constructs are needed for semantics of the language, but they do not exist as a single sensor task in the mTask implementation in Clean. A sensor task includes the location, where the sensor's value can be read from. Every sensor is of the type `Sensor` τ, where τ is the type of the values that are stored at the location it is pointing to.

The values at the sensor's location cannot be changed by any of the tasks, only observed. It comes from the nature of the sensor; its values are changed due to the change in the environment it is located in, and they can only be read. Thus, in the semantics, the values at the referenced location can only be changed by the system: whenever a new value is received, it is updated at the location by the system. The processing of a sensor's input is done by the handling semantics described in Section 4.3.6.

4.2.2 Shared Data Source

In mTask there are tasks that read, set or update the value of SDS. All of these tasks are of the type `MTask` τ, but their arguments differ.

The task **getSds** reads the value of SDS at the location l and, when observed, returns the read value. It is similar to $\widehat{TOP}$ shared editor with the only difference that it does not show any UI and, thus, the value cannot be updated by a user.

The task **setSds** takes two arguments and sets a value of SDS. Its first argument should be the location of the SDS and the second argument is the value that the location should be set to.

The task **updSds** updates the value of the specified SDS. It takes two arguments: the first is the location of the SDS, and the second is a function that changes the SDS's current value to the new one. The task **updSds** can be replaced by a sequence of reading SDS's value, a function that changes the read value and writing a new SDS's value to the location in $\widehat{TOP}$. However, it is not possible in mTask, because normalisation does not reduce the task to its final form, but strides it once. Thus, tasks that write the value to SDS can interrupt the process of updating the value using the three aforementioned

tasks and make the updated value invalid. Execution of **updSds** is atomic and ensures that the SDS value is not changed by another task between reading and writing.

4.2.3 Return, unstable

The tasks **rtrn** and **unstable** are described in Tim Steenvoorden's thesis [1] with read-only editor ($\boxdot\ e$), but they are not mentioned in the $\widehat{TOP}$ foundation paper [4] used for this thesis. Moreover, both mTask returns use the notion of stability that is specific to the language.

The task **rtrn** e returns the stable value of expression e. It takes an expression e of type τ and yields a value of the `MTask` τ type. The task **unstable** does the same as **rtrn** with the difference that the yielded value is unstable.

4.2.4 Combinators and continuations

There are two types of combinators used in mTask: sequential ($\gg\!*$) and parallel (&& and ||).

Even though there is only one type of sequential combinator—step ($\gg\!*$), it is possible to derive different types of workflow from it with the use of continuations:

- **OnValue** continuation: when the left-hand side of the step combinator yields a value and the predicate function evaluates to `True`, then the combinator proceeds to the right-hand side. If the left-hand side emits no value, the combinator continues waiting for it.
- **OnStable** continuation: is similar to the first combinator, but the value has to be stable.
- **OnUnstable** continuation: is similar to the first combinator, but the value has to be unstable.

There are two parallel combinators: conjunction (&&) and disjunction (||). The combinators are named after *and* and *or* logic functions. The conjunction combinator waits until both tasks have task values and, when observed, yields both task values as a pair. The disjunction combinator selects the left-most value: if the left-hand side does not have a stable value, the combinator, when observed, yields the right-hand side's stable value, and in any other case, the left-hand side value is selected. The observation function $\mathcal{V}$, defined in Section 4.3.2, describes how the task value is selected when the task is observed.

4.3 Semantics

All language constructs defined in Section 4.1 are used in this section to formalise the semantics of the mTask language.

The semantics are divided into two big-step semantics, which are evaluation ($\downarrow$) and normalisation ($\Downarrow$), and three small-step semantics being striding ($\rightsquigarrow$), handling ($\rightarrow$) and interaction ($\Rightarrow$). Similarly to the $\widehat{TOP}$ semantics described in Section 3.2.2 and Figure 3.3, normalisation semantics uses evaluation semantics, which evaluates tasks and expressions, and stride to normalise the evaluated constructs. Moreover, interaction semantics uses normalised expressions and tasks to handle user input on the layer of handling semantics or continue task normalisation.

Figure 4.8 provides an example of how different layers of semantics are connected using simple program. Tasks t_1 and t_2 in the parallel *or* combinator are considered to be some abstract tasks that do not have any observable values. Thus, the last stride rewrites the combinator into itself.

The following sections describe all layers of the semantics mentioned above and provide the semantic rules for language constructs.

$$
\begin{array}{lll}
 & \textbf{sns}\ s = 4\ \textbf{in} & \\
 & \quad (\circledast\ s \gg\!* [\textbf{OnValue}\ (\lambda x.x = 42)\ (\lambda x.t_1 \mid\mid t_2)]) & \\
\downarrow & \circledast\ 4 \gg\!* [\textbf{OnValue}\ (\lambda x.x = 42)\ (\lambda x.t_1 \mid\mid t_2)] & \\
 & \qquad \mathit{Normalisation\ I} : \mathit{evaluation\ I} & \\
\downarrow & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Normalisation\ I} : \mathit{evaluation\ II} & \\
\rightsquigarrow & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Normalisation\ I} : \mathit{stride\ step\ with\ no\ task\ value} & \\
\xrightarrow{4\ 35} & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Interaction\ I} : \mathit{handle} & \\
\downarrow & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Normalisation\ II} : \mathit{evaluate} & \\
\rightsquigarrow & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Normalisation\ II} : \mathit{stride\ step} & \\
\downarrow & \textbf{if}\ 35 = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Stride\ step} : \mathit{evaluate\ I} & \\
\downarrow & \textbf{if}\ \texttt{False}\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Stride\ step} : \mathit{evaluate\ II} & \\
\downarrow & \lightning \qquad \mathit{Stride\ step} : \mathit{evaluate\ III} & \\
\xrightarrow{4\ 42} & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Interaction\ II} : \mathit{handle} & \\
\downarrow & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Normalisation\ III} : \mathit{evaluate} & \\
\rightsquigarrow & \circledast\ 4 \gg\!* \lambda s\ v.\textbf{if}\ v = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Normalisation\ III} : \mathit{stride\ step} & \\
\downarrow & \textbf{if}\ 42 = 42\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Stride\ step} : \mathit{evaluate\ I} & \\
\downarrow & \textbf{if}\ \texttt{True}\ \textbf{then}\ (t_1 \mid\mid t_2)\ \textbf{else}\ \lightning & \\
 & \qquad \mathit{Stride\ step} : \mathit{evaluate\ II} & \\
\downarrow & t_1 \mid\mid t_2 \qquad \mathit{Stride\ step} : \mathit{evaluate\ III} & \\
\downarrow & t_1 \mid\mid t_2 \qquad \mathit{Normalisation\ IV} : \mathit{evaluate} & \\
\rightsquigarrow & t_1 \mid\mid t_2 \qquad \mathit{Normalisation\ IV} : \mathit{stride}\ (\mathit{no\ task\ values}) &
\end{array}
$$

Figure 4.8: Semantics example for mTask.

4.3.1 Evaluation

Evaluation is a big-step semantics, which is used for pretask and expression evaluation. The rule structure in this semantics is the same as in $\widehat{TOP}$: $e, \sigma \downarrow v, \sigma'$; and it means *the expression e in state σ is evaluated to a value v in state σ'*. Values can be abstractions, pairs, units, constants, locations or tasks. Pretasks always evaluate to tasks, while expressions evaluate to values described in the value grammar. The grammar of evaluation values is shown in Figure 4.9.

$v ::=$		*Values*
	$\lambda\overline{x}_n : \overline{\tau}_n.e \mid \langle e_1,\ e_2 \rangle \mid \langle\ \rangle$	abstraction, pair, unit
$\mid$	$c \mid l \mid t \mid \mu$	constant, location, task, sensor
$t ::=$	$\circledast\, \mu \mid \lightning$	*Tasks*: sensor, fail
$\mid$	**getSds** l | **setSds** $l\ v$	get, set SDS
$\mid$	**updSds** $l\ (\lambda x.e)$	update SDS
$\mid$	**rtrn** v | **unstable** v	return stable, unstable
$\mid$	$t_1\ \&\&\ t_2 \mid t_1 \mid\mid t_2$	parallel: and, or
$\mid$	$t_1\ \gg\!*\ (\lambda x\ y.e)$	sequential step

Figure 4.9: Value grammar in mTask.

Figure 4.10 shows the evaluation of function, location and sensor definitions and host language expressions. The bar above the variables and values indicates that expressions have several elements of the same syntactic category, and n is a natural positive number indicating the number of such elements.

The notion $\sigma[l \mapsto v_1]$ means that the value of SDS at the location l is set to the value v_1 in state σ. The substitution method is used for sensors or functions when they are replaced with their definitions. For example, $\Lambda[\mu\ /\ s]$ means that all the occurrences of s in Λ are replaced with the sensor's physical location μ [14]. The substitution method is used in `E-LetFunc`, `E-LetSensor` and `E-App`.

`E-LetFunc` sets the definition of the function x_f, where x_f is the name of the function and $\overline{x}_n$ are the arguments. Since the expression e can have a recursive call of the function x_f, an additional substitution was introduced above the line. In case e contains a recursive call of the function, it replaces all mentions of x_f with the new let-expression which sets the body of the function to be e again [14, Sec. 2.3.3]. When the let-expression is evaluated

within the expression e, then the rule `E-LetFunc` is used again on this let-expression.

`E-App` evaluation happens when the function x described in syntax of the host language in Section 4.1.2 is replaced with the lambda-expression by an `E-LetFunc` rule application. Thus, its form is different compared to the one described in language syntax.

`E-LetFunc`

$$\frac{\Lambda[\lambda\overline{x}_n.e[\textbf{let } x_\text{f}\ \overline{x}_n = e \textbf{ in } x_\text{f}\ \overline{x}_n \ /\ x_\text{f}] \ /\ x_\text{f}], \sigma \downarrow v, \sigma'}{\textbf{let } x_\text{f}\ \overline{x}_n = e \textbf{ in } \Lambda, \sigma \downarrow v, \sigma'}$$

`E-LetLoc`

$$\frac{e, \sigma \downarrow v_1, \sigma' \quad \Lambda, \sigma'[l \mapsto v_1] \downarrow v_2, \sigma''}{\textbf{sds } l = e \textbf{ in } \Lambda, \sigma \downarrow v_2, \sigma''}$$

`E-LetSensor`

$$\frac{\Lambda[\mu \ /\ s], \sigma \downarrow v, \sigma'}{\textbf{sns } s = \mu \textbf{ in } \Lambda, \sigma \downarrow v, \sigma'}$$

`E-App`

$$\frac{\overline{e}_n, \sigma \downarrow \overline{v}_n, \sigma' \quad e[\overline{v}_n \ /\ \overline{x}_n], \sigma' \downarrow v, \sigma''}{(\lambda\overline{x}_n.e)\ \overline{e}_n, \sigma \downarrow v, \sigma''}$$

`E-Binary`

$$\frac{e_1, \sigma \downarrow v_1, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma'' \quad v_3 = v_1 \star v_2}{e_1 \star e_2, \sigma \downarrow v_3, \sigma''}$$

`E-Pair`

$$\frac{e_1, \sigma \downarrow v_1, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{\langle e_1, e_2 \rangle, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma''}$$

`E-Neg`

$$\frac{e, \sigma \downarrow v_1, \sigma' \quad v_2 = \neg v_1}{\neg e, \sigma \downarrow v_2, \sigma'}$$

`E-Const`

$$\frac{}{c, \sigma \downarrow v, \sigma'}$$

`E-Unit`

$$\frac{}{\langle\ \rangle, \sigma \downarrow \langle\ \rangle, \sigma}$$

`E-Value`

$$\frac{}{v, \sigma \downarrow v, \sigma}$$

`E-Loc`

$$\frac{}{l, \sigma \downarrow l, \sigma}$$

`E-Sensor`

$$\frac{}{\mu, \sigma \downarrow \mu, \sigma}$$

`E-IfTrue`

$$\frac{e_1, \sigma \downarrow \texttt{True}, \sigma' \quad e_2, \sigma' \downarrow v, \sigma''}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \sigma \downarrow v, \sigma''}$$

`E-IfFalse`

$$\frac{e_1, \sigma \downarrow \texttt{False}, \sigma' \quad e_3, \sigma' \downarrow v, \sigma''}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \sigma \downarrow v, \sigma''}$$

Figure 4.10: Evaluation semantics of expressions in mTask.

The step combinator ($\gg\!*$) uses a continuation function for evaluation of the right-hand side of the combinator. The function is provided in Figure 4.11. It takes the list of continuations and the two abstract names of the variables that is not yet in the domain of σ. The second *Name* argument v is the task value and the first *Name* argument s is the value's stability. The substitution method is used in the function and the argument x in functions e_1 and e_2 is replaced with an actual value.

$$\mathcal{C} : [Continuation] \times Name \times Name \rightarrow Expression$$

$$\begin{aligned}
\mathcal{C}([], s, v) &= \lightning \\
\mathcal{C}(\textbf{OnValue}\ (\lambda x.e_1)\ (\lambda x.e_2) : c_r, s, v) &= \textbf{if}\ e_1[v/x] \\
&\quad\ \textbf{then}\ e_2[v/x] \\
&\quad\ \textbf{else}\ \mathcal{C}(c_r, s, v) \\
\mathcal{C}(\textbf{OnStable}\ (\lambda x.e_1)\ (\lambda x.e_2) : c_r, s, v) &= \textbf{if}\ s \wedge e_1[v/x] \\
&\quad\ \textbf{then}\ e_2[v/x] \\
&\quad\ \textbf{else}\ \mathcal{C}(c_r, s, v) \\
\mathcal{C}(\textbf{OnUnstable}\ (\lambda x.e_1)\ (\lambda x.e_2) : c_r, s, v) &= \textbf{if}\ \neg s \wedge e_1[v/x] \\
&\quad\ \textbf{then}\ e_2[v/x] \\
&\quad\ \textbf{else}\ \mathcal{C}(c_r, s, v)
\end{aligned}$$

Figure 4.11: Continuation function in mTask.

Tasks are evaluated pretasks. The expressions that these pretasks contain are evaluated, which allows to normalise them in the corresponding semantics described in Section 4.3.3. For example, the second argument of **setSds** pretask is an expression, but as soon as evaluation happens, its argument becomes an actual value v. Later, it is used in striding semantics, shown in Section 4.3.4, to update the SDS's value at location l.

Figure 4.12 provides the rules for evaluation of pretasks into tasks. All sensors, SDS locations and functions are always defined at the top level, which means that their definition evaluation happens first. All sensors have their actual location value μ by the time of their evaluation. SDS's default value is set by `E-LetLoc`, so whenever the value is read by the task, it is always there. Thus, location l will have its default value by the time SDS related functions are evaluated and normalised.

`E-Step` evaluates its left-hand side into the task, whilst the continuation list is transformed into the expression e_2 using the continuation function shown in Figure 4.11.

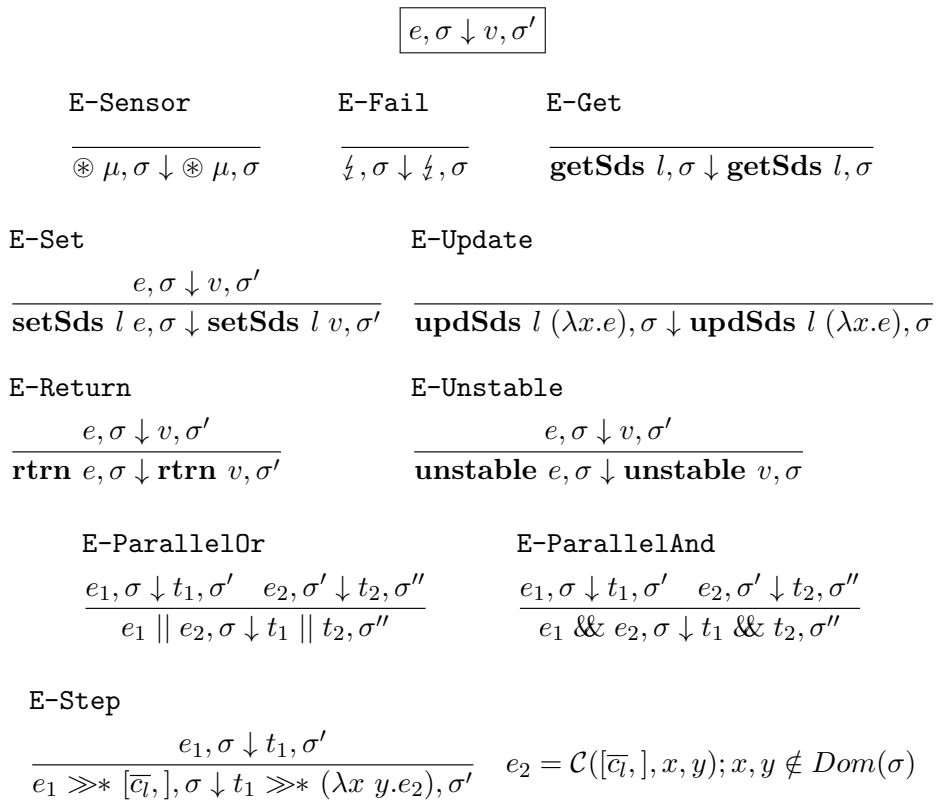

Figure 4.12: Evaluation semantics of pretasks in mTask.

4.3.2 Task fail and observation

The striding semantics, described in Section 4.3.4, makes use of functions that observe a task. These functions are semantic functions on the syntax tree of tasks.

The fail function $\mathcal{F}$ is the total function that determines if the task is failing. It takes a task as its argument and returns a boolean value as the result. It returns `True` whenever a task contains a fail $\lightning$ and `False` in any other case. Compared to the $\widehat{TOP}$, the mTask fail function does not use the state as an argument, because the result is solely calculated based on the given task's current form. When a task that includes sub tasks is checked, the sub tasks are checked and their values are used to calculate the failing of the initial task. The fail function is shown in Figure 4.13.

$$\mathcal{F} : Task \rightarrow Bool$$

$$\begin{aligned}
\mathcal{F}(\circledast\ \mu) &= \texttt{False} \\
\mathcal{F}(\lightning) &= \texttt{True} \\
\mathcal{F}(\textbf{getSds}\ l) &= \texttt{False} \\
\mathcal{F}(\textbf{setSds}\ l\ v) &= \texttt{False} \\
\mathcal{F}(\textbf{updSds}\ l\ (\lambda x.e)) &= \texttt{False} \\
\mathcal{F}(\textbf{rtrn}\ v) &= \texttt{False} \\
\mathcal{F}(\textbf{unstable}\ v) &= \texttt{False} \\
\mathcal{F}(t_1\ \&\&\ t_2) &= \mathcal{F}(t_1) \vee \mathcal{F}(t_2) \\
\mathcal{F}(t_1\ ||\ t_2) &= \mathcal{F}(t_1) \wedge \mathcal{F}(t_2) \\
\mathcal{F}(t_1 \gg\!* \ (\lambda x\ y.e_2)) &= \mathcal{F}(t_1)
\end{aligned}$$

Figure 4.13: Fail function in mTask.

The observable value function $\mathcal{V}$ is a partial function that checks the value of a tasks. The function returns two values as a pair: the first value indicates the stability of the task's value and the second value of the pair is the task value itself. When the observed task has no value in the given state σ, it returns the empty value $\perp$.

Figure 4.14 defines the function for all tasks. The notions $\sigma(\mu)$ and $\sigma(l)$ are functions that return the value of the sensor at the location μ and the value of the SDS at the location l in the current state σ respectively. Sometimes the value of a sensor has not been updated even once and it yields no value and $\perp$ is returned in this case.

The observations of the fail task and step combinator return an empty value. The observation on the tasks is only done in the process of normali-

sation of the step combinator shown in Figure 4.17. The tasks **setSds** and **updSds** are always reduced to **rtrn** v by the time their values are observed. Thus, observations of these tasks are empty values.

$$\mathcal{V} : Task \times State \rightharpoonup Bool \times Value$$

$$\begin{aligned}
\mathcal{V}(\circledast\ \mu, \sigma) = &\begin{cases} \langle \texttt{False}, \sigma(\mu) \rangle & \textbf{when } \sigma(\mu) \neq \bot \\ \bot & \textbf{otherwise} \end{cases} \\
\mathcal{V}(\lightning, \sigma) = &\ \bot \\
\mathcal{V}(\textbf{getSds } l, \sigma) = &\ \langle \texttt{False}, \sigma(l) \rangle \\
\mathcal{V}(\textbf{setSds } l\ v, \sigma) = &\ \bot \\
\mathcal{V}(\textbf{updSds } l\ (\lambda x.e), \sigma) = &\ \bot \\
\mathcal{V}(\textbf{rtrn } v, \sigma) = &\ \langle \texttt{True}, v \rangle \\
\mathcal{V}(\textbf{unstable } v, \sigma) = &\ \langle \texttt{False}, v \rangle \\
\mathcal{V}(t_1\ \&\&\ t_2, \sigma) = &\begin{cases} \langle\ b_1 \wedge b_2, \langle v_1, v_2 \rangle\ \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = \langle b_1, v_1 \rangle \\ & \quad \wedge\ \ \mathcal{V}(t_2, \sigma) = \langle b_2, v_2 \rangle \\ \bot & \textbf{otherwise} \end{cases} \\
\mathcal{V}(t_1\ ||\ t_2, \sigma) = &\begin{cases} \mathcal{V}(t_1, \sigma) & \textbf{when } \mathcal{V}(t_2, \sigma) = \bot \\ \mathcal{V}(t_2, \sigma) & \textbf{when } \mathcal{V}(t_1, \sigma) = \bot \\ \langle \texttt{True}, v_1 \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = \langle \texttt{True}, v_1 \rangle \\ \langle \texttt{True}, v_2 \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = \langle \texttt{False}, v_1 \rangle \\ & \quad \wedge\ \ \mathcal{V}(t_2, \sigma) = \langle \texttt{True}, v_2 \rangle \\ \langle \texttt{False}, v_1 \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = \langle \texttt{False}, v_1 \rangle \\ & \quad \wedge\ \ \mathcal{V}(t_2, \sigma) = \langle \texttt{False}, v_2 \rangle \end{cases} \\
\mathcal{V}(t_1 \gg\!* (\lambda x\ y.e), \sigma) = &\ \bot
\end{aligned}$$

Figure 4.14: Observable values function in mTask.

4.3.3 Normalisation

The normalisation semantics is used to reduce tasks and combinators once. Same as in TopHat, it is a big-step semantics. The semantics rule $e, \sigma \Downarrow t, \sigma'$ means that *the expression e in state σ can be normalised to the task t in state σ′*.

However, this semantics is very different compared to normalisation semantics in $\widehat{TOP}$. Tasks in $\widehat{TOP}$ are normalised until they reach a fully reduced form. Meanwhile, in mTask, the tasks are fully evaluated, but they are strided only once. This allows the semantics to accept the input on tasks that are strided once and create a system that is more reactive to the input. Normalisation semantics also majorly influences the interaction semantics described in Section 4.18. Figure 4.15 shows the rule of normalisation semantics.

$$\boxed{e, \sigma \Downarrow t, \sigma'}$$

`N-Reduce`

$$\frac{e, \sigma \downarrow t, \sigma' \quad t, \sigma' \rightsquigarrow t', \sigma''}{e, \sigma \Downarrow t', \sigma'}$$

Figure 4.15: Normalisation semantics in mTask.

4.3.4 Striding

The striding semantics is a small-step semantics used for reducing tasks and combinators. This semantics is a part of normalisation semantics described in Section 4.3.3. All rules are of the form $t, \sigma \rightsquigarrow t', \sigma'$, which means that *task* t *in state* σ *is reduced to task* t' *in state* σ'.

Tasks like sensor, fail, reading SDS, return and unstable are reduced to themselves. The task **setSds** is reduced to a stable return carrying the value v—a new value of the SDS at the location l set to it in state σ'. The task **updSds** also reduces to a stable return. It reads the value of the specified SDS in state σ, processes it and updates the value at the specified location l with the processed value, which is returned as the stable return. The notion $\sigma(l \mapsto v)$ means that the value of the SDS at the location l is updated to value v in state σ. Figure 4.16 shows the striding semantics rule for tasks.

$$\boxed{t, \sigma \rightsquigarrow t', \sigma'}$$

`S-Sensor`

$$\frac{}{\circledast\ \mu, \sigma \rightsquigarrow \circledast\ \mu, \sigma}$$

`S-Fail`

$$\frac{}{\lightning, \sigma \rightsquigarrow \lightning, \sigma}$$

`S-Update`

$$\frac{v_1 = \sigma(l) \quad e[x \mapsto v_1], \sigma \downarrow v_2, \sigma'}{\textbf{updSds}\ l\ (\lambda x.e), \rightsquigarrow \textbf{rtrn}\ v_2, \sigma'(l \mapsto v_2)}$$

`S-Set`

$$\frac{}{\textbf{setSds}\ l\ v, \sigma \rightsquigarrow \textbf{rtrn}\ v, \sigma'(l \mapsto v)}$$

`S-Get`

$$\frac{}{\textbf{getSds}\ l, \sigma \rightsquigarrow \textbf{getSds}\ l, \sigma}$$

`S-Return`

$$\frac{}{\textbf{rtrn}\ v, \sigma \rightsquigarrow \textbf{rtrn}\ v, \sigma}$$

`S-Unstable`

$$\frac{}{\textbf{unstable}\ v, \sigma \rightsquigarrow \textbf{unstable}\ v, \sigma}$$

Figure 4.16: Striding semantics of tasks in mTask.

Figure 4.17 shows rules of striding semantics for combinators. Both parallel combinators reduce their subtasks, and the result of the reduction is still the combinator or, in case of `S-OrLeft` and `S-OrRight`, one of the tasks. In $\widehat{TOP}$, *or* parallel combinator reduces to the left-most task with the value, mTask has similar stride rule for this combinator, but the task values should be stable in order to reduce to one of the tasks.

The step combinator observes the value of the left-hand side task and, depending on the value, the expression resulting from the continuations is used for the reduction of the combinator.

$$\boxed{t, \sigma \rightsquigarrow t', \sigma'}$$

Conjunctive parallel combinator.

S-And

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \;\&\&\; t_2, \sigma \rightsquigarrow t_1' \;\&\&\; t_2', \sigma''}$$

Disjunctive parallel combinator.

S-OrLeft

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma'}{t_1 \;||\; t_2, \sigma \rightsquigarrow t_1', \sigma''} \qquad \mathcal{V}(t_1', \sigma') = \langle \texttt{True}, v \rangle$$

S-OrRight

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \;||\; t_2, \sigma \rightsquigarrow t_2', \sigma''} \quad (\mathcal{V}(t_1', \sigma') = \bot \vee \mathcal{V}(t_1', \sigma') = \langle \texttt{False}, v \rangle) \wedge \mathcal{V}(t_2', \sigma'') = \langle \texttt{True}, v \rangle$$

S-OrNone

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \;||\; t_2, \sigma \rightsquigarrow t_1' \;||\; t_2', \sigma''} \quad (\mathcal{V}(t_1', \sigma') = \bot \vee \mathcal{V}(t_1', \sigma') = \langle \texttt{False}, v \rangle) \wedge (\mathcal{V}(t_2', \sigma'') = \langle \texttt{False}, v \rangle \vee \mathcal{V}(t_2', \sigma'') = \bot)$$

Step.

S-StepStay

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma'}{t_1 \gg\!* \; (\lambda x \; y.e), \sigma \rightsquigarrow t_1' \gg\!* \; (\lambda x \; y.e), \sigma'} \quad \mathcal{V}(t_1', \sigma') = \bot$$

S-StepFail

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad (\lambda x \; y.e) \; s \; v, \sigma' \downarrow t_2, \sigma''}{t_1 \gg\!* \; (\lambda x \; y.e), \sigma \rightsquigarrow t_1' \gg\!* \; (\lambda x \; y.e), \sigma''} \quad \mathcal{V}(t_1', \sigma') = \langle s, v \rangle \wedge \mathcal{F}(t_2)$$

S-StepCont

$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \quad (\lambda x \; y.e) \; s \; v, \sigma' \downarrow t_2, \sigma''}{t_1 \gg\!* \; (\lambda x \; y.e), \sigma \rightsquigarrow t_2, \sigma''} \quad \mathcal{V}(t_1', \sigma') = \langle s, v \rangle \wedge \neg \mathcal{F}(t_2)$$

Figure 4.17: Striding semantics of combinators in mTask.

4.3.5 Interaction

The interaction semantics is a small-step semantics that handles the interaction between the system and the tasks. The interaction semantics makes sure that the input, in case there is any, is handled by the handling semantics and that the resulting task is normalised by the normalisation semantics. In case there is no input, the interaction semantics continues with task normalisation. The normalisation semantics is described in Section 4.3.3 and the handling semantics is described in Section 4.3.6.

The difference in normalisation semantics also affects interaction semantics in mTask. $\widehat{TOP}$ has input driven semantics, and the interaction semantics has only one rule to handle the input and then fully normalises the task. In mTask there can also be no input, and thus, extra rules were added to continue with the task normalisation when no input is sent by the system. All rules accept input i, which can either be nothing $\varnothing$ or value for the sensor $\mu\ v$. The semantics' rules are of the form $t, \sigma \overset{i}{\Rightarrow} t', \sigma'$ and mean *sending an input i to the task t in state σ handles the input and prepares the resulting task t' in the state σ'*. Figure 4.18 shows the interaction semantics rules.

$$\boxed{t, \sigma \overset{i}{\Rightarrow} t', \sigma'}$$

`I-HandleInput`

$$\frac{t, \sigma \xrightarrow{\mu\ v} t', \sigma' \quad t', \sigma' \Downarrow t'', \sigma'' \quad t'', \sigma'' \overset{i}{\Rightarrow} t''', \sigma'''}{t, \sigma \overset{\mu\ v}{\Longrightarrow} t''', \sigma'''}$$

`I-HandleRepeat`

$$\frac{t, \sigma \Downarrow t', \sigma' \quad t', \sigma' \overset{i}{\Rightarrow} t'', \sigma''}{t, \sigma \overset{\varnothing}{\Longrightarrow} t'', \sigma''} \quad \textbf{where } t \neq t' \vee \sigma \neq \sigma'$$

`I-HandleDone`

$$\frac{t, \sigma \Downarrow t', \sigma'}{t, \sigma \overset{\varnothing}{\Longrightarrow} t'', \sigma''} \quad \textbf{where } t = t' \wedge \sigma = \sigma'$$

Figure 4.18: Interaction semantics in mTask.

The language makes use of different inputs, and it is important to provide the grammar for them. Input can be nothing $\varnothing$, which indicates that there was no system input, or the value v that has to be written to the location μ. The type of value v and the type of input expected at the location μ are

expected to be the same. Figure 4.19 shows the possible inputs in mTask.

$$i ::= \qquad \mu\ v \mid \varnothing \qquad \qquad \mathit{Input} : \text{value, nothing}$$

Figure 4.19: Input grammar in mTask.

Function $\mathcal{I}$ calculates what input events the tasks and combinators expect to receive. This function is used to prove that handling semantics is sound, meaning that all the inputs in the set of possible inputs can be handled by it. As soundness of the semantics stays out of the scope of this thesis, the function $\mathcal{I}$ is listed in Appendix A.2.

4.3.6 Handling

The handling semantics is responsible for handling the inputs. The mTask inputs are values that physically connected sensors yield. In mTask, the way inputs are handled differs a lot from what $\widehat{TOP}$ does. The sensor task in mTask holds the value of its location, which, for example, can be pin on the micro controller. When a sensor's value is updated by the system, the value at the location μ changes, and not the value of the sensor task itself. In comparison, editors in $\widehat{TOP}$ hold the values, and it is important to pass the values all the way to the editors traversing the whole tree.

The mentioned differences influence the handling semantics in mTask. There is no need to traverse the whole semantical tree to change the value of the sensor. It can be done in the state σ using the notation $\sigma(\mu \mapsto v)$, where the value v is set to the location μ in state σ. The form of the rule for handling semantics is the following:

$$\boxed{t, \sigma \xrightarrow{\mu\ v} t, \sigma(\mu \mapsto v)}$$

The meaning of the rule is *the task t in state σ takes a sensor location μ and an input value v, writes the value to the location and yields the task t in the state σ*. The rule considers that the input received is of the same type as the input expected. The task t in this rule can be any evaluated task.

4.4 TopHat and mTask differences

$\widehat{TOP}$'s and mTask's differences start from their syntax. In $\widehat{TOP}$, functions are defined in the expression category using λ-expressions, while in mTask, functions are always defined at the top level using let-expressions. SDSs and sensors also use the same syntactic category for their definitions. The notion of stability is absent in $\widehat{TOP}$, but the mTask continuations handle it well in the step combinator. Moreover, stability plays an important role in the *or* combinator, because none of the tasks are chosen if both of them have unstable value (or no value). All these differences in syntax mentioned above influence the types, typing and semantic rules.

However, the unexpected, biggest and the most interesting difference between the languages is the structure of normalisation and interaction semantics. $\widehat{TOP}$ was originally developed as a mathematical formalisation of the TOP paradigm, while mTask is a real-world implementation of the TOP paradigm in the domain of programming IoT devices. $\widehat{TOP}$'s evaluation is developed to be input driven and, whenever input is received and handled, a task is fully normalised. The mTask language is tightly integrated with the changing world, which means that even if the system still works on the reduction of the program, it should be ready to react to the changing environment. Thus, tasks are reduced only once before interaction semantics are checked for new input. If there is no input, the interaction semantics continues with normalisation, if there is an input, it is handled first and then the task is normalised. This allows the language to react to the inputs as fast as possible, without waiting for the full reduction of the task.

Moreover, sensors in mTask turned out to be nothing similar to the valued and unvalued editors in $\widehat{TOP}$ despite the initial view. Sensors are similar to the shared editors and they just hold the location of the sensor and not the value itself. Since there are no tasks that hold values, it was possible to simplify the handling semantics to one rule, which accepts the location μ and the value v as the input and updates the value at the specified location with the new one.

In the case of mTask, the name "interaction semantics" can be changed to something different that better represents the process of accepting the input and normalisation. But the name was kept with regard to the research question, because the name "interaction semantics" is used in $\widehat{TOP}$ and comparison of languages is an important aspect of the thesis.

Chapter 5

Discussion and conclusion

In the previous chapters, the mTask syntax and operational semantics were created on the basis of the $\widehat{TOP}$ semantics introduced earlier. The mTask language was syntactically described and mathematically formalised using five different semantics: evaluation, normalisation, striding, interaction and handling; each responsible for a certain part of language processing.

5.1 Discussion and future work

The scope of this work was description of the mTask language syntax and creation of its semantics and the goal was reached. However, there is still room for improving the formalisation of the language.

5.1.1 Delay and repeat

When describing the language semantics, two tasks—**delay** e and **rpeat** e—were left out of the scope of this thesis. The tasks added a lot of complexity to the thesis and were not formalised due to the time constraints. Formal definition of these tasks should be considered as a future work extending the described semantics of mTask.

5.1.2 Type preservation

To show that the semantics of the language is correct, it is important to start with proving that evaluation, normalisation and handling semantics preserve the types. All the semantics mentioned above should preserve the types of expressions and tasks according to the typing rules we have created in Section 4.1.4. Such proofs could be done in the future and they will show whether the types are preserved according to the typing rules on all the layers of semantics.

5.1.3 Soundness and completeness

To validate that the function $\mathcal{I}$ indeed calculates all the possible inputs, it is important to show that the set of resulting inputs is sound and complete in handling semantics. All the possible inputs should be handled by the corresponding semantics to prove that the resulting from the function $\mathcal{I}$ inputs are sound. Moreover, all the inputs should be complete, which means that the function $\mathcal{I}$ produces all the possible inputs that can be handled by the handling semantics in mTask. Showing that the function is sound and complete will bring mTask closer to being fully mathematically defined and the proves will show that semantics work correctly.

5.1.4 Verification

The work done in this thesis was done solely on paper and lacks computer verification. In addition to the proofs mentioned above, computer verification adds confidence that the semantics of the language was done correctly and verifies that the proofs for type preservation, soundness and completeness are correct.

5.2 Conclusion

In conclusion, it was possible to create semantics of mTask using $\widehat{TOP}$ formalisation as the basis. However, it required a lot of changes, which was not possible to predict initially.

$\widehat{TOP}$ and mTask are both simple TOP systems. We hoped that it would be easy to make a semantics for mTask based on the $\widehat{TOP}$ semantics, but it appeared to be quite challenging.

The main difference that introduced the complexity of defining mTask semantics was the difference in the drivers. The $\widehat{TOP}$ semantics are input driven, which means that whenever input is received, it is handled, the program is evaluated and normalised until the task reaches its fully reduced form using the striding semantics. This is not the case for mTask, where input is not the event that the system relies on. When there is no system input received by the mTask program, the semantics still continues with the normalisation process. The normalisation semantics in mTask reduce tasks only once and after that the input presence is checked again. The way mTask is implemented makes it possible to program quite reactive IoT systems and, arguably, schedule tasks more fairly. Such input handling and task normalisation majorly influences normalisation and interaction semantics in current work. In addition, there is a high chance that such differences will heavily influence future work on semantics expansion and its proofs.

Another major difference between the two languages are the inputs of the system. $\widehat{TOP}$ uses user input from the abstract GUI, which is saved

to the editors or used for continuation of the tasks. In mTask, input comes from the physical sensors at the specified location. In the syntax, sensors are references to specific locations. This way, their values are not stored locally in the sensor construct, but are stored in the environment. This implements differences in the way these values are handled by the handling semantics. This difference was somewhat unexpected, but it only simplified the handling semantics and a general rule was introduced that covers handling input the same way for all the tasks.

In addition, $\widehat{TOP}$ does not have a notion of stability of values, unlike mTask. Stability mainly influences the semantics of disjunctive parallel combinator and step combinator. The striding semantics of the first one remained similar to the striding semantics of the choice combinator in $\widehat{TOP}$. However, the step combinator makes use of continuations to handle the task value from the left-hand side and to calculate the task for the right-hand side. The helper function to transform continuations into an expression was introduced for the evaluation semantics. Nothing similar is used in $\widehat{TOP}$ and the step combinator is one of the hardest concepts introduced in the mTask semantics. However, the need to handle stability of the values was known from the beginning, and thus, did not introduce any additional unexpected complexity.

In short, mTask formalisation turned out to be quite different from the $\widehat{TOP}$, but it was still possible to use the $\widehat{TOP}$ syntax and semantics as a rough template for describing the syntax and semantics of mTask. Even though there were some unexpected differences between the languages' structures, the research of this bachelor thesis succeeded and the semantics for the mTask was described.

Bibliography

[1] T. Steenvoorden, "Tophat task-oriented programming with style." Draft PhD thesis.

[2] P. Achten, P. Koopman, and R. Plasmeijer, "An introduction to task oriented programming," in *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers* (V. Zsók, Z. Horváth, and L. Csató, eds.), pp. 187–245, Cham: Springer International Publishing, 2015.

[3] S. Michels, "Building itask applications: A gui paradigm based on workflows," Master's thesis, Radboud University, 2010.

[4] T. Steenvoorden, N. Naus, and M. Klinik, "Tophat: A formal foundation for task-oriented programming," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[5] M. Rothmuller and S. Barker, "Iot the internet of transformation 2020," Apr 2020.

[6] M. Lubbers, P. Koopman, and R. Plasmeijer, "Interpreting task oriented programs on tiny computers," in *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, IFL '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[7] M. Lubbers, P. Koopman, and R. Plasmeijer, "Task oriented programming and the internet of things," in *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, IFL 2018, (New York, NY, USA), p. 83–94, Association for Computing Machinery, 2018.

[8] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman, "Task-oriented programming in a pure functional language," in *Proceedings of*

the 14th Symposium on Principles and Practice of Declarative Programming, PPDP '12, (New York, NY, USA), p. 195–206, Association for Computing Machinery, 2012.

[9] "Itasks." `https://clean.cs.ru.nl/ITasks`. Accessed April 3, 2022 [Online].

[10] M. Lubbers, P. Koopman, A. Ramsingh, J. Singer, and P. Trinder, "Tiered versus tierless iot stacks: Comparing smart campus software architectures," in *Proceedings of the 10th International Conference on the Internet of Things*, IoT '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[11] S. Crooijmans, "Reducing the power consumption of iot devices in task-oriented programming," Master's thesis, Radboud University, 2021.

[12] N. Naus, T. Steenvoorden, and M. Klinik, "A symbolic execution semantics for tophat," in *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, IFL '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[13] H. R. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*. USA: John Wiley & Sons, Inc., 1992.

[14] M. Grant, Z. Palmer, and S. Smith, *Principles of Programming Languages Version 1.0.3*. USA: Scott F. Smith, 2021.

Glossary

$\widehat{TOP}$ TopHat. 1–3, 5, 6, 9–14, 16, 18, 21–26, 28–31, 35–37, 39, 43, 45, 46, 48–53, 57

EDSL Embedded Domain Specific Language. 8, 9

GUI Graphic User Interface. 10, 52

IoT Internet of Things. 1, 4, 5, 7, 9, 27, 35, 50, 52

SDS Shared Data Source. 2, 7, 9, 10, 28–31, 33–36, 39, 41, 43, 46, 50

TOP Task-Oriented Programming. 1, 4–9, 27, 50, 52

Appendix A

Appendix

A.1 Possible inputs function in TopHat

$$\mathcal{I} : Tasks \times States \to \mathcal{P}(Inputs)$$

$$\begin{aligned}
\mathcal{I}(\square\ v, \sigma) &= \{v' \mid \varnothing \vdash v' : \tau\} \cup \{\texttt{E}\} \quad \textbf{where } \square\ v : \texttt{Task } \tau \\
\mathcal{I}(\boxtimes\ \tau, \sigma) &= \{v' \mid \varnothing \vdash v' : \tau\} \\
\mathcal{I}(\blacksquare\ l, \sigma) &= \{v' \mid \varnothing \vdash v' : \tau\} \quad \textbf{where } \blacksquare\ l : \texttt{Task } \tau \\
\mathcal{I}(\lightning, \sigma) &= \varnothing \\
\mathcal{I}(t_1 \blacktriangleright e_2, \sigma) &= \mathcal{I}(t_1, \sigma) \\
\mathcal{I}(t_1 \triangleright e_2, \sigma) &= \mathcal{I}(t_1, \sigma) \cup \{C \mid \mathcal{V}(t_1, \sigma) = v_1 \wedge \\
&\qquad e_2\ v_1, \sigma \Downarrow t_2, \sigma' \wedge \neg\mathcal{F}(t_2, \sigma')\} \\
\mathcal{I}(t_1 \bowtie t_2, \sigma) &= \{\texttt{F}\ \ i \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{\texttt{S}\ \ i \mid i \in \mathcal{I}(t_2, \sigma)\} \\
\mathcal{I}(t_1 \blacklozenge t_2, \sigma) &= \{\texttt{F}\ \ i \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{\texttt{S}\ \ i \mid i \in \mathcal{I}(t_2, \sigma)\}
\end{aligned}$$

Figure A.1: Inputs functions in TopHat.

A.2 Possible inputs function in mTask

$$\mathcal{I} : Task \times State \to \mathcal{P}(Input)$$

$$\begin{aligned}
\mathcal{I}(\circledast\ \mu) &= \{v \mid \varnothing \vdash v : \tau\} \quad \textbf{where } \circledast\ \mu : \texttt{MTask } \tau \\
\mathcal{I}(\lightning) &= \varnothing \\
\mathcal{I}(\textbf{getSds } l) &= \varnothing \\
\mathcal{I}(\textbf{rtrn } v) &= \varnothing \\
\mathcal{I}(\textbf{unstable } v) &= \varnothing \\
\mathcal{I}(t_1 \mathbin{\&\&} t_2) &= \varnothing \\
\mathcal{I}(t_1 \mid\mid t_2) &= \varnothing \\
\mathcal{I}(t_1 \gg\!* e_2) &= \varnothing
\end{aligned}$$

Figure A.2: Possible inputs function in mTask.