# TopHat: A formal foundation for task-oriented programming

Tim Steenvoorden
Software Science
Radboud University
Nijmegen, The Netherlands
tim@cs.ru.nl

Nico Naus
Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
n.naus@uu.nl

Markus Klinik
Software Science
Radboud University
Nijmegen, The Netherlands
m.klinik@cs.ru.nl

## ABSTRACT

Software that models how people work is omnipresent in today's society. Current languages and frameworks often focus on usability by non-programmers, sacrificing flexibility and high level abstraction. Task-oriented programming (TOP) is a programming paradigm that aims to provide the desired level of abstraction while still being expressive enough to describe real world collaboration. It prescribes a declarative programming style to specify multi-user workflows. Workflows can be higher-order. They communicate through typed values on a local and global level. Such specifications can be turned into interactive applications for different platforms, supporting collaboration during execution. TOP has been around for more than a decade, in the forms of iTasks and mTasks, which are tailored for real-world usability. So far, it has not been given a formalisation which is suitable for formal reasoning.

In this paper we give a description of the TOP paradigm and then decompose its rich features into elementary language elements, which makes them suitable for formal treatment. We use the simply typed lambda-calculus, extended with pairs and references, as a base language. On top of this language, we develop TopHat, a language for modular interactive workflows. We describe TopHat by means of a layered semantics. These layers consist of multiple big-step evaluations on expressions, and two labelled transition systems, handling user inputs.

With TopHat we prepare a way to formally reason about TOP languages and programs. This approach allows for comparison with other work in the field. We have implemented the semantic rules of TopHat in Haskell, and the task layer on top of the iTasks framework. This shows that our approach is feasible, and lets us demonstrate the concepts by means of illustrative case studies. TOP has been applied in projects with the Dutch coast guard, tax office, and navy. Our work matters because formal program verification is important for mission-critical software, especially for systems with concurrency.

**ACM Reference Format:**
Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A formal foundation for task-oriented programming. In *Principles and Practice of Programming Languages 2019 (PPDP '19), October 7–9, 2019, Porto, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3354166.3354182

## 1 INTRODUCTION

Many applications these days are developed to support workflows in institutions and businesses. Take for example expense declarations, order processing, and emergency management. Some of these workflows occur on the boundary between organisations and customers, like flight bookings or tax returns. What they all have in common is that they need to interact with different people (end users, tax officers, customers, etc.) and they use information from multiple sources (input forms, databases, sensors, etc.).

### 1.1 Tasks

We call interactive units of work based on information sources *tasks*. Tasks model real world collaboration between users, are driven by work users do, and are assigned to some user. Users could be people out in the field or sitting behind their desks, as well as machines doing calculations or fetching data.

### 1.2 Task-oriented programming

Task-oriented programming (TOP) is a programming paradigm which targets the sweet spot between faithful modelling workflows and rapid prototyping of multi-user web applications supporting these workflows [23]. TOP focusses on modelling collaboration patterns. This gives rise to a user's need to interact and share information. Next to that, TOP automatically provides solutions to common development jobs like designing GUIs, connecting to databases, and servers-client communication.

Therefore, a language that supports TOP should choose the right level of abstraction to support two things. Firstly, it should provide primitive building blocks that are useful for high-level descriptions of how users collaborate, with each other and with machines. These building blocks are: *editors*, *composition*, and *shared data*. Secondly, it should be able to generate applications, including graphical user interfaces, from workflows modelled with said building blocks.

Users can work together in a number of ways, and this is reflected in TOP by task compositions. There is *sequential* composition, *parallel* composition, and *choice*. Users need to communicate in order to engage in these forms of collaboration. This is reflected in TOP by three kinds of communication mechanisms. There is data flow *alongside* control flow, where the result of a task is passed onto the next. There is data flow *across* control flow, where information is shared between multiple tasks. Finally, there is communication with the *outside* world, where information is entered into the system via input events. The end points where the outside world interacts with TOP applications are called editors. In generated applications, editors can take many forms, like input fields, selection boxes, or map widgets.

## 1.3 Utilisation

Currently, we know of two frameworks implementing TOP: iTasks and mTasks. iTasks is an implementation of TOP, in the form of a shallowly embedded domain-specific language in the lazy functional programming language Clean. It is a library that provides editors, monadic combinators, and shared data sources. iTasks uses the generic programming facilities of Clean to derive rich client and server applications from a single source. It has been used to model an incident management tool for the Dutch coast guard [15]. Also it has been used numerous times to prototype ideas for Command and Control [12, 24], and in a case study for the Dutch tax authority [25].

mTasks is a subset of iTasks, focusing on IOT devices and deployment on micro controllers. It has been used to control home thermostats and other home automation applications [13]. Both implementations currently lack formal semantics which are suited to prove properties about tasks.

## 1.4 Challenges

Both iTasks and mTasks have been designed for developing real-world applications. They are constantly being extended and improved with this goal in mind. The different variations of task combinators and the details that come with real-world requirements, make it hard to see what the essence of TOP is. Also, the tight integration of both frameworks with Clean, makes it difficult to see where the boundaries are. This makes formal reasoning about TOP programs impossible.

In this paper, we want to take a step back and look at the spirit of TOP. We do this both formally and informally. Informally in the sense that we give an intuitive description of the features that define task-oriented programming. Formally in the sense that we develop a language which formalises these features as language constructs, and we give them a semantics in the style that is common in programming language research. We separate the task layer and the underlying host language, both syntactically and semantically. Thus making explicit which properties of TOP come from the task layer, and which come from functional programming. Our challenge, therefore, is to model the properties of TOP into a language and pave the way for formal treatment of TOP programs. We give this formal language the name $\widehat{\text{TOP}}$ (TopHat).

## 1.5 Contributions

Our contributions to workflow modelling, functional programming language design, and rapid application development are as follows.

(i) We describe the essential concepts of task-oriented programming. (ii) We present a formal language for modelling declarative workflows, called $\widehat{\text{TOP}}$, embedded in a simply typed $\lambda$-calculus. It is based on the aforementioned essential TOP concepts. (iii) We develop a layered operational semantics for $\widehat{\text{TOP}}$ that is driven by user input. The semantics of the task language is clearly separated from the semantics of the underlying host language. (iv) Along with this semantics, we present the following semantic observations on tasks: the current value, whether a term is stuck, the current user interface, and the accepted inputs. (v) We prove progress and type preservation for $\widehat{\text{TOP}}$. (vi) Using both the essential concepts and the formal language, we compare TOP with related work in areas ranging from business process modelling, to process algebras and reactive programming. (vii) We implemented the whole semantic system in the functional programming language Haskell [16]. (viii) To create executable applications, we implemented the task layer of $\widehat{\text{TOP}}$ in iTasks. This also demonstrates that the former is a subset of the latter.

## 1.6 Structure

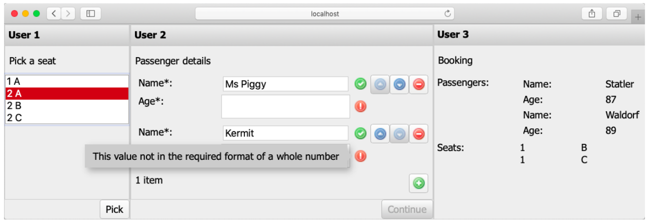
In Section 2 we demonstrate the functionality of $\widehat{\text{TOP}}$ by means of an example, Section 3 gives an overview of the essential concepts of TOP. Section 4 introduces the $\widehat{\text{TOP}}$ language syntax and Section 5 the semantics. Then in Section 6 we show that certain properties hold for the language. We take a look at related work in Section 7 and conclude in Section 8.

## 2 EXAMPLE

In this section we develop an example program to demonstrate the capabilities of $\widehat{\text{TOP}}$. The example is a small flight booking system. It demonstrates communication on all three levels: with the environment, across control flow, and alongside it. Also, it shows synchronisation and input validation.

The requirements of the application are as follows. (i) A user has to enter a list of passengers for which to book tickets. (ii) At least one of these passengers has to be an adult. (iii) After a valid list of passengers has been entered, the user has to pick seats. (iv) Only free seats may be picked. (v) Every passenger must have exactly one seat. (vi) Multiple users should be able to book tickets at the same time.

For this example we assume that the host language has lists and four functions on them: all, any, intersect, and difference. The functions all and any check if all or any elements in a list satisfy a given predicate. The functions intersect and difference compute the set-intersection and set-difference of two lists. We also make use of string equality (≡), dereferencing (!), reference assignment (:=), and expression sequencing (;). For brevity, we omit the type annotations of variable bindings.



Name*: Ms Piggy
Age*:
Name*: Kermit
This value not in the required format of a whole number

Passengers: Name: Statler Age: 87 Name: Waldorf Age: 89
Seats: 1 B
1 C

**Figure 1: Running web application of the flight booking example using a translation to iTasks. It shows three users booking a flight simultaneously. The first user entered name and age and continued picking seats. The second is entering details of two passengers. The ages are not filled in, therefore the Continue button is disabled. The message bubble shows that the age field only accepts integer values. The third user finished a booking, therefore, the first user can not pick seats 1B and 1C any more.**

*Example 2.1 (Flight booking).* We start by defining some type aliases. A passenger is a pair with name and age. A seat is a pair with a row number and a seat letter.

```
type PASSENGER = STRING × INT
type SEAT = INT × STRING
```

Choosing seats requires reading and updating shared information. The list of free seats is stored in a reference.

```
let freeSeats = ref [⟨1,"A"⟩ , ⟨1,"B"⟩ , ⟨1,"C"⟩ , ...]
```

Now we develop our workflow in a top-down manner. Our flight booking starts with an interactive task ⊠(LIST PASSENGER), where users can enter a list of passengers. A task ⊠ $\tau$ is an empty editor that asks for a value of the given type $\tau$. Passengers are valid if their name is not empty and their age is at least 0. Lists of passengers are valid if each passenger is valid, and at least one of the passengers is an adult. When the user has entered a valid list of passengers, the step after ▷ becomes enabled, and the user can proceed to picking seats. In case of an invalid list of passengers, the step is guarded by the failing task ↯ .

```
let valid = λp. not (fst p ≡ "") ∧ snd p ≥ 0 in
let adult = λp. snd p ≥ 18 in
let allValid = λps. all valid ps ∧ any adult ps in
let bookFlight = ⊠(LIST PASSENGER) ▷ λps.
  if allValid ps then chooseSeats ps else ↯
```

A selection of seats is correct if every entered seat is free.

```
let correct = λss. intersect ss !freeSeats ≡ ss in
let chooseSeats = λps. ⊠(LIST SEAT) ▷ λss.
  if correct ss ∧ length ps ≡ length ss
    then confirmBooking ps ss else ↯
```

The function confirmBooking removes the selected seats from the shared list of free seats, and displays the end result using an editor, denoted by □.

```
let confirmBooking = λps. λss.
  freeSeats := difference !freeSeats ss; □⟨ps, ss⟩
```

The main task starts three bookFlight tasks, which could be performed by three different users in parallel.

```
bookFlight ⋈ bookFlight ⋈ bookFlight
```

A screenshot of the running application is shown in Fig. 1.

All instances of the bookFlight task have access to the shared list of free seats. Rewriting the example in a language without side effects would not only be cumbersome, obfuscating the code with explicit threading of state, but it would be impossible to model the parallel execution of three bookFlight tasks. It is not known upfront which task will finish first, and thus it is not possible to thread the free seat list between the parallel tasks.

## 3 INTUITION

This section gives an overview of the core concepts of task-oriented programming.

### 3.1 Tasks model collaboration

The central objective of TOP is to *coordinate collaboration.* The basic building blocks of $\widehat{\text{TOP}}$ for expressing collaboration are task combinators. They express ways in which people can work together. Tasks can be executed after each other, at the same time, or conditionally. This motivates the combinators step, parallel, and choice.

*Example 3.1 (Breakfast).* The following program shows the different collaboration operators in the setting of making breakfast. Users have a choice (◇) whether they want tea or coffee. They always get an egg. The drink and the food are prepared in parallel (⋈). When both the drink and the food are prepared, users can step (▷) to eating the result.

```
let mkBrkfst : TASK Drink → TASK Food → TASK ⟨Drink,Food⟩
  = λmkDrink. λmkFood. mkDrink ⋈ mkFood in
mkBrkfst (mkTea ◇ mkCoffee) mkEgg ▷ enjoyMorning
```

The way the combinators are defined matches real life closely. When we want to have breakfast, we have to complete several other tasks first before we can do so. We decide what we want to have and then prepare it. We can prepare the different items we have for breakfast in parallel, but not at the same time. For example, it is impossible to scramble eggs, and put on the kettle for tea simultaneously. Instead, what is meant by parallel is that *the order in which we do tasks and the smaller tasks that they are composed of, does not matter.* Then finally, only when every item we want to have for breakfast is ready, can we sit down and enjoy it.

### 3.2 Tasks are reusable

There are three ways in which tasks are modular. First, larger tasks are composed of smaller ones. Second, tasks are first-class, they can be arguments and results of functions. Third, tasks can be values of other tasks. These aspects make it possible for programmers to model custom collaboration patterns. Example 3.1 demonstrates how tasks can be parameterised by other tasks: mkBrkfst is a collaboration pattern that always works the same way, regardless of which food and drink are being prepared.

### 3.3 Tasks are driven by user input

Input events drive evaluation of tasks. When the system receives a valid event, it applies this event to the current task, which results in a new task. In this way the system communicates with the environment. Inputs are synchronous, which means the order of execution is completely determined by the order of the inputs.

In $\widehat{\text{TOP}}$, *editors* are the basic method of communication with the environment. Editors are modelled after input widgets from graphical user interfaces. There are different editors, denoted by different box symbols. Take for example an editor holding the integer seven: □ 7. Such an editor reacts to change events, for example the values 42 or 37, which are of the same type.

The sole purpose of editors is to interact with users by remembering the last value that has been sent to them. There are no output events. As values of editors can be observed, for example by a user interface, editors facilitate both input and output. An empty editor (⊠) stands for a prompt to input data, while a filled editor (□) can be seen either as outputting a value, or as an input that comes with a default value.

*Example 3.2 (Vending machine).* This example demonstrates external communication and choice. It is a vending machine that dispenses a biscuit for one coin and a chocolate bar for two coins.

```
let vend : TASK SNACK = ⊠INT ▷ λn. if n ≡ 1 then □Biscuit
```

**else if** $n \equiv 2$ **then** □ChocolateBar **else** ↯

The editor ⊠ Int asks the user to enter an amount of money. This editor stands for a coin slot in a real machine that freely accepts and returns coins. There is a continue button that is initially disabled, due to the fact that the left hand side of the step combinator has no value. When the user has inserted exactly 1 or 2 coins, the continue button becomes enabled. When the user presses the continue button, the machine dispenses either a biscuit or a chocolate bar, depending on the amount of money. Snacks are modelled using a custom type.

## 3.4 Tasks can be observed

Several observations can be made on tasks. One of those is determining the value of a task. Not all tasks have a value, the empty editor for instance, which makes value observation partial. I.e., the value of □ 7 is 7, but the value of ⊠ Int is $\bot$.

Another observation is the set of input events a task can react to. For example, the task □ 7 can react to value events, as discussed before.

In order to render a task, we need to observe a task's user interface. This is done compositionally. User interfaces of combined tasks are composed of the user interfaces of the components. For example, of two tasks combined with a step combinator, only the left hand side is rendered. Two parallel tasks are rendered next to each other. Combining this information with the task's value and possible inputs, we can display the current state of the task, together with buttons that show the actions a user can engage in.

The final observation is to determine whether a task results in a failure, denoted by ↯. The step combinator $\triangleright$ and the choice combinator $\lozenge$ use this to prevent users from picking a failing task.

## 3.5 Tasks are never done

Tasks never terminate, they always keep reacting to events. Editors can always be changed or cleared, and step combinators move on to new tasks.

In a step $t \triangleright e$, the decision to move on from a task $t$ to its continuation $e$ is taken by $\triangleright$, not by $t$. The decision is based on a speculative evaluation of $e$. The step combinator in $t \triangleright e$ passes the value $v$ of $t$ to the continuation $e$. Steps act like $t$ as long as the step is guarded. A step is guarded if either the left task has no value, or the speculative evaluation of $e$ applied to $v$ yields the failure task ↯. Once it becomes unguarded, the step continues as the result of $e\ v$. Speculative evaluation is designed so that possible side effects are undone. The task $t \triangleright e$ additionally requires a continue event C to proceed.

Step combinators give rise to a form of internal communication. They represent data flow that *follows* control flow.

## 3.6 Tasks can share information

The step combinator is one form of internal communication, where task values are passed to continuations. Another form of internal communication is shared data. Shared data enables data flow *across* control flow, in particular between parallel tasks. Shared data sources are assignable references whose changes are immediately visible to all tasks interested in them. Users can not directly interact with shared data, a shared editor is required for that. If $x$ is a reference of type $\tau$, then ■ $x$ is an editor whose value is that of $x$.

The semantics of $\widehat{\text{TOP}}$ requires all updates to shared data and all enabled internal steps to be processed before any further communication with the environment can take place.

*Example 3.3 (Cigarette smokers).* The cigarette smokers problem by Downey [7] is a surprisingly tricky synchronisation problem. We study it here because it demonstrates the capabilities of guarded steps. The problem is stated as follows. In order to smoke a cigarette, three ingredients are required: tobacco, paper, and a match. There are three smokers, each having one of the ingredients and requiring the other two. There is an agent that randomly provides two of those. The difficulty lies in the requirement that only the smoker may proceed whose missing ingredients are present.

Downey models availability of the ingredients with a semaphore for each ingredient. The agent randomly signals two of the three. The solution proposed by Downey involves an additional mutex, three additional semaphores, three additional threads called *pushers*, and three regular Boolean variables. The job of the pushers is to record availability of their ingredient in their Boolean variable, and check availability of other resources, waking the correct smoker when appropriate.

What is important is that the implementation of what is essentially deadlock-free waiting for two semaphores requires a substantial amount of additional synchronisation, together with non-trivial conditional statements. $\widehat{\text{TOP}}$ allows a simple solution to this problem, using guarded steps. Steps can be guarded with arbitrary expressions. The parallel combinator can be used to watch two shared editors at the same time. Let match, paper, and tobacco be references to Booleans. The smokers are defined as follows.

**let** continue = $\lambda\langle x,y\rangle$ . **if** $x \wedge y$ **then** smoke **else** ↯ **in**
**let** tobaccoSmoker = (■ match ⋈ ■paper) ▷ continue **in**
**let** paperSmoker = (■ tobacco ⋈ ■match) ▷ continue **in**
**let** matchSmoker = (■ tobacco ⋈ ■paper) ▷ continue **in**
tobaccoSmoker ⋈ paperSmoker ⋈ matchSmoker

When the agent supplies two of the ingredients by setting the respective shares to True, only the step of the smoker that waits for those becomes enabled.

## 3.7 Tasks are predictable

Let $t_1$ and $t_2$ be tasks. The parallel combination $t_1 \bowtie t_2$ stands for two independent tasks carried out at the same time. This operator introduces interleaving concurrency. For the system it does not matter if the tasks are executed by two people actually in parallel, or by one person who switches between the tasks. The inputs sent to the component tasks are interleaved into a serial stream, which is sent to the parallel combinator. We assume that such a serialization is always possible. The tasks are truly independent of each other, all interleavings are permitted. The environment must prefix events to $t_1$ and $t_2$ respectively by F (first) and S (second). This unambiguously renames the inputs, removing any source of nondeterminism.

With concurrency comes the need for synchronisation, in situations where only some but not all interleavings are desired. The basic method for synchronisation in $\widehat{\text{TOP}}$ is built into the step combinator. The task $t \triangleright e$ can only continue execution when two conditions are met: Task $t$ must have a value $v$, and $e\ v$ must not evaluate to ↯. Programmers can encode arbitrary conditions in $e\ v$, which are evaluated atomically between interaction steps. This allows a variety of synchronisation problems to be solved in an intuitive and straight-forward manner.

Hoare [9] states that nondeterminism is only ever useful for *specifying* systems, never for implementing them. $\widehat{\text{TOP}}$ is meant solely for implementation and does not have any form of nondeterminism. Input events for parallel tasks are disambiguated, internal steps (▶) have a well-defined evaluation order, and internal choice (♦) is left-biased.

## 3.8 Recap

Collaboration in the real world consists of three aspects: communication, concurrency, and synchronisation. These aspects are reflected in TOP on a high level of abstraction, hiding the details of communication. For example, the cigarette smokers communicate with each other, but the programs do not explicitly mention sending or receiving events.

By focusing on collaboration instead of communication, TOP leads to specifications closer to real-world workflows which, at the same time, can be used to generate multi-user applications to support such workflows.

## 4 LANGUAGE

In this section, we present the constructs of $\widehat{\text{TOP}}$, our modular interactive workflow language. We define the host and task language, the types, and the static semantics. Then we describe the workings of each construct using examples. These constructs are formalised in Section 5.

### 4.1 Expressions

The host language is a simply typed $\lambda$-calculus, extended with some basic types and ML-style references. We use references to represent shared data sources. The grammar in Fig. 2 defines the syntax of the host language. It has abstractions, applications, variables, and constants for booleans, integers and strings. The symbol $\star$ stands for binary operators. For the result of parallel tasks we need pairs. Conditionals come in handy for defining guards. References will be used to implement shared editors. Our treatment of references closely follows the one by Pierce [22]. Creating a reference using the keyword **ref** yields a location $l$. While $x$ denotes program variables, $l$ denotes store locations. Locations are not intended to be directly manipulated by the programmer. The symbols ! and := stand for dereferencing and assignment. The unit value will be used as the result of assignments.

| | | |
|---|---|---|
| $e ::=$ | | Expressions |
| $\mid$ | $\lambda x : \tau.\ e \mid e_1\ e_2$ | – abstraction, application |
| $\mid$ | $x \mid c \mid e_1 \star e_2$ | – variable, constant, operation |
| $\mid$ | $\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \langle\rangle$ | – branch, unit |
| $\mid$ | $\langle e_1, e_2\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$ | – pair, projections |
| $\mid$ | $\mathbf{ref}\ e \mid !e \mid e_1 := e_2 \mid l$ | – references, location |
| $\mid$ | $p$ | – pretask |
| $c ::=$ | | Constants |
| $\mid$ | $B \mid I \mid S$ | – boolean, integer, string |

**Figure 2: Language grammar**

We use double quotation marks to denote strings. Integers are denoted by their decimal representation, and booleans are written True and False. We freely make use of the logic operators not, $\wedge$, and $\vee$, arithmetic operators $+$, $-$, $\times$, and the string append operator $+\!\!+$. Furthermore, we use standard comparison operations $<$, $\leq$, $\equiv$, $\not\equiv$, $\geq$, and $>$. The symbol $\star$ stands for any of those. The notation $e_1; e_2$ is an abbreviation for $(\lambda x : \textsc{Unit}\,.\ e_2)\ e_1$, where $x$ is a fresh variable. The notation $\mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2$ is an abbreviation for $(\lambda x : \tau.\ e_2)\ e_1$.

The grammar in Fig. 3 specifies the syntactic category of *pretasks*. Pretasks are tasks that have unevaluated subexpressions with respect to the host language. How expressions are evaluated will be discussed in Section 5.1. Each pretask will be discussed in more detail in the following subsections. We use open symbols (□, ⊠, ▷, ◊) for tasks that require user input, and closed symbols (■, ▶, ♦) for tasks that can be evaluated without user input.

| | | |
|---|---|---|
| $p ::=$ | | Pretasks |
| $\mid$ | $\square\, e \mid \boxtimes\, \tau \mid \blacksquare\, e$ | – 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 |
| $\mid$ | $e_1 \blacklozenge e_2 \mid e_1 \lozenge e_2$ | – choice: internal, external |

**Figure 3: Task grammar**

*Typing.* Figure 4 shows the grammar of types used by $\widehat{\text{TOP}}$. It has functions, pairs, basic types, unit, references, and tasks.

| | | |
|---|---|---|
| $\tau ::=$ | | Types |
| $\mid$ | $\tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \beta$ | – function, product, basic |
| $\mid$ | $\textsc{Unit} \mid \textsc{Ref}\ \tau \mid \textsc{Task}\ \tau$ | – unit, reference, task |
| $\beta ::=$ | | Basic types |
| $\mid$ | $\textsc{Bool} \mid \textsc{Int} \mid \textsc{String}$ | – boolean, integer, string |

**Figure 4: Type grammar**

Typing rules are of the form $\Gamma, \Sigma \vdash e : \tau$, which should be read as "in environment $\Gamma$ and store typing $\Sigma$, expression $e$ has type $\tau$". Typing rules for expressions in the host language are presented in the appendix[1]. The typing rules for pretasks are given in Fig. 5. Most typing rules lift the type of their subexpressions into the TASK-type. The typing rules for steps make sure the continuations $e_2$ are functions which accept a well-typed value from the left hand side (T-THEN, T-NEXT). References, and therefore shared editors, can only be of a basic type so they do not introduce implicit recursion (T-UPDATE).

### 4.2 Editors

Programs in $\widehat{\text{TOP}}$ model interactive workflows. Interaction means communication with end users. End users should be able to enter information into the system, change it, clear it, reenter it, and so on. To do this, we introduce the concept of *editors*. Editors are typed containers that either hold a value or are empty. Editors that have a value can be *changed*. Empty editors can be *filled*. This is depicted as a state diagram in Fig. 6 below.

Editors stand for various forms of input and output, for example widgets in a GUI, form fields on a webpage, sensors, or network connections. Consider the editor for a person's age from Example 2.1. Users can change the value until they are satisfied with it. Editors are meant to capture this constantly changing nature of user input. The user interface of an editor depends on its type. This could be

[1] https://github.com/timjs/tophat/appendix.pdf

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

$$\text{T-Edit}\ \frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \square\, e : \textsc{Task}\,\tau} \qquad \text{T-Enter}\ \frac{}{\Gamma, \Sigma \vdash \boxtimes\, \tau : \textsc{Task}\,\tau} \qquad \text{T-Update}\ \frac{\Gamma, \Sigma \vdash e : \textsc{Ref}\,\beta}{\Gamma, \Sigma \vdash \blacksquare\, e : \textsc{Task}\,\beta}$$

$$\text{T-Then}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\,\tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \textsc{Task}\,\tau_2}{\Gamma, \Sigma \vdash e_1 \blacktriangleright e_2 : \textsc{Task}\,\tau_2} \qquad \text{T-Next}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\,\tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \textsc{Task}\,\tau_2}{\Gamma, \Sigma \vdash e_1 \triangleright e_2 : \textsc{Task}\,\tau_2}$$

$$\text{T-Fail}\ \frac{}{\Gamma, \Sigma \vdash ↯ : \textsc{Task}\,\tau} \qquad \text{T-And}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\,\tau_1 \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\,\tau_2}{\Gamma, \Sigma \vdash e_1 \bowtie e_2 : \textsc{Task}\,(\tau_1 \times \tau_2)}$$

$$\text{T-Or}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\,\tau \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\,\tau}{\Gamma, \Sigma \vdash e_1 \blacklozenge e_2 : \textsc{Task}\,\tau} \qquad \text{T-Xor}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\,\tau \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\,\tau}{\Gamma, \Sigma \vdash e_1 \lozenge e_2 : \textsc{Task}\,\tau}$$

**Figure 5: Typing rules**

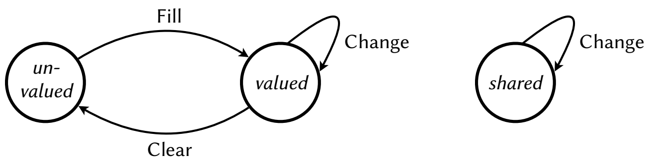


**Figure 6: Possible states of an editor and its transitions. Shared editors cannot be cleared.**

an input field for strings, a toggle switch for booleans, or even a map with a pin for locations. It could also be a parser that tries to parse a line of text to match the type of the editor.

*Valued and unvalued editors* ($\square\, e, \boxtimes\, \tau$). Editors that hold an expression $e : \tau$ have type $\textsc{Task}\,\tau$. Empty editors are annotated with a type in order to ensure type safety and type preservation during evaluation.

*Shared editors* ($\blacksquare\, e$). Shared editors watch references, lifting their value into the task domain. If $e$ is a reference $\textsc{Ref}\,\tau$, then $\blacksquare\, e$ is of type $\textsc{Task}\,\tau$. Shared editors cannot be cleared, only changed.

Changes to a shared editor are immediately visible to all shared editors watching the same reference. Imagine two users, Marco and Christopher, both watching shared editors of the same coordinates. The editors are visualised as a pin on a map. When Marco moves his pin, he updates the value of the shared editor, thereby changing the value of the reference. This change is immediately reflected on Christopher's screen: The pin changes its position on his map. This way Marco and Christopher can work together to edit the same information.

Two other important use cases for shared editors are sensors and time. Sensors can be represented as external entities that periodically update a shared editor with their current sensor value. Similarly, the current time can be stored in a shared editor (■time) which is periodically updated by a clock. The actual sensor and the clock are not modelled in $\widehat{\textsc{top}}$. We assume that they exist as external users that send update events to the system. This allows programmers to write tasks that react to sensor values or timeouts.

## 4.3 Steps

Editors represent atomic units of work. In this section we look at ways to compose smaller tasks into bigger ones. Composing tasks can be done in two ways, sequential and parallel. Parallel composition comes in two variants: combining two tasks (*and*-parallel) and choosing between two tasks (*or*-parallel). We study sequential composition first, and after that combining and choosing.

*Internal and external step* ($t \blacktriangleright e, t \triangleright e$). Sequential composition has a task $t$ on the left and a continuation $e$ on the right. External steps (▷) must be triggered by the user, while internal steps (▶) are taken automatically. The accompanying typing rules are T-Then and T-Next. According to these rules, the left hand side must be a task $t : \textsc{Task}\,\tau_1$, and the right hand side $e : \tau_1 \to \textsc{Task}\,\tau_2$ must be a function that, given the task value of $t$, calculates the task with which to continue.

Steps are guarded, which means that the step combinators can only proceed when the following conditions are met. The left hand side must have a value, only then can the right hand side calculate the successor task. The successor task must not be ↯, introduced below. This is enforced on the semantic level, as described in the next section. The internal step can proceed immediately when these conditions are met. The external step must additionally receive a continue event C.

*Example 4.1 (Conditional stepping).* Consider the following:

⊠Int ▶ $\lambda n$. **if** $n \equiv 42$ **then** □"Good" **else** □"Bad"

Initially, the step is guarded because the editor does not have a value. When users enter an integer, the program continues immediately with either □"Good" or □"Bad", depending on the input.

*Fail* (↯). Fail is a task that never has a value and never accepts input. The typing rule T-Fail states that it has type $\textsc{Task}\,\tau$ for any type $\tau$. Programmers can use ↯ to tell steps that no sensible successor task can be determined.

*Example 4.2 (Guarded stepping).* Consider this slight variation on Example 4.1:

⊠Int ▶ $\lambda n$. **if** $n \equiv 42$ **then** □"Good" **else** ↯

The user is asked to enter an integer. As long as the right hand side of ▶ evaluates to ↯, the step cannot proceed, and the user can keep editing the integer. As soon as the value of the left hand side is 42, the right hand side evaluates to something other than ↯, and the step proceeds to □"Good".

*Example 4.3 (Waiting).* With the language constructs seen so far it is possible to create a task that waits for a specified amount of time. To do this, we make use of a shared editor holding the current time (see Section 4.2), and a guarded internal step.

```
let wait : Int → Task Unit = λamount : Int.
  ■time ▶ λstart : Int.
  ■time ▶ λnow : Int.
    if now > start + amount then □⟨⟩ else ↯
```

The first step is immediately taken, resulting in *start* to be the time at the moment wait is executed. The second step is guarded until the current time is greater to the start time plus the requested *amount*.

## 4.4 Parallel

A common pattern in workflow design is splitting up work into multiple tasks that can be executed simultaneously. In $\widehat{\text{TOP}}$, all parallel branches can progress independently, driven by input events. This requires inputs to be tagged in order to reach the intended task.

There are two ways to proceed after a parallel composition. One way is to wait for all tasks to produce results and combine those, the other to pick the first available result. Both ways introduce explicit forks and implicit joins in $\widehat{\text{TOP}}$.

*Combination* ($e_1 \bowtie e_2$). A combination of two tasks is a parallel *and*. It has a value only if both branches have a value. This is reflected in the typing rule T-And, It shows that if the first task has type $\tau_1$, and the second has type $\tau_2$, their combination has the pair type $\tau_1 \times \tau_2$.

*Example 4.4 (Combining).* The task

$\boxtimes\textsc{Int} \bowtie \square\text{"Batman"} \blacktriangleright \lambda\langle n, s\rangle\,.\,\square(\text{replicate } n \text{ "Na"} +\!\!+\ s)$

can only step when both editors have values. When it steps, the continuation uses the pair to calculate the result.

*Internal and external choice* ($e_1 \blacklozenge e_2, e_1 \lozenge e_2$). Internal choice ($\blacklozenge$) is a parallel *or*. It picks the leftmost branch that has a value. Its typing rule T-Or states that both branches must have the same type $\textsc{Task}\ \tau$. For example $\boxtimes\,\textsc{Int} \blacklozenge \square\,37$ normalizes to $\square\,37$, because $\boxtimes\,\textsc{Int}$ doesn't have a value. Users can work on both branches of an internal choice simultaneously.

External choice ($\lozenge$) is different in this regard. An external choice requires users to pick a branch before continuing with it. This means users cannot work on the branches of an external choice before picking one.

*Example 4.5 (Delay).* We illustrate the use of internal and external choice by means of an example that asks users to proceed with a given task or to cancel. If the user does not make a choice within a given time frame, the program proceeds automatically. The example makes use of the task wait from Example 4.3.

**let** cancel : $\textsc{Task Unit} = \square\langle\rangle$ **in**
**let** delay : $\textsc{Int} \to \textsc{Task Unit} \to \textsc{Task Unit} = \lambda n.\ \lambda \mathit{proceed}.$
  $(\mathit{proceed} \lozenge \text{cancel}) \blacklozenge (\text{wait } n \blacktriangleright \lambda u : \textsc{Unit}.\ \mathit{proceed})$

Note that delay is higher-order. It is a task which takes another task as parameter.

## 4.5 Annotations

Tasks can be annotated with additional information. The system can use this information in various ways. Possible use cases are labels for the user interface, resource consumption information for static resource analysis, or messages for automatic end-user feedback. Annotations are not covered in this paper. Our Haskell implementation of $\widehat{\text{TOP}}$ supports annotating tasks with user IDs, so that individual tasks in a large workflow can be assigned to different users. These annotations are used to filter the user interfaces for each user so that they can only see their part of the workflow.

## 5 SEMANTICS

In this section we formalise the semantics of the language constructs described in Section 4. We organise this by following the structure of the language. Firstly, the task language is embedded in a simply typed $\lambda$-calculus. This requires a specification of the *evaluation* of terms in the host language, and how it handles the task language. Secondly, there are two ways to drive evaluation of task expressions, internally by the system itself, and externally by the user. This is done in two additional semantics, one for the internal *normalisation* of tasks, and another for the *interaction* with the end user.

The three main layers of semantics are thus evaluation, normalisation, and interaction. The semantics, together with *observations*, will be discussed in the following subsections. Figure 7 shows the relation between all semantics arrows. It also shows that there are two helper semantics, *handle* and *stride*. We use the convention that downward arrows are big-step semantics, and rightward arrows are small-step semantics.

$$\begin{tikzcd}
 & \text{stride } (\rightsquigarrow) \arrow[dl, "uses"'] & \text{handle } (\longrightarrow) \arrow[dl, "uses"'] \\
\text{evaluate } (\downarrow) & \text{normalise } (\Downarrow) \arrow[l, "uses"'] \arrow[u, "uses"'] & \text{interact } (\Longrightarrow) \arrow[l, "uses"'] \arrow[u, "uses"']
\end{tikzcd}$$

**Figure 7: Semantic functions defined in this report and their relation.**

One of our explicit goals is to keep the semantics for evaluation and normalisation separate, to not mix general purpose programming notions with workflow specific semantics. This is achieved by letting tasks be values in the host language.

## 5.1 Evaluating expressions

The host language evaluates expressions using a big-step semantics. Evaluating an expression $e$ in state $\sigma$ into a value $v$ in state $\sigma'$ is denoted by $e, \sigma \ \downarrow\ v, \sigma'$. To ease reasoning about references, we choose a call-by-value evaluation strategy.

Figure 8 shows values with respect to the evaluation semantics. Tasks are values, and the operands of task constructors are evaluated eagerly. Exceptions to this are steps and external choice, where some or all of the operands are not evaluated.

$$\begin{array}{rcll}
v & ::= & & \text{Values} \\
 & \mid & \lambda x : \tau\,.\,e \ \mid\ \langle v_1, v_2\rangle \ \mid\ \langle\rangle & \text{– abstraction, pair, unit} \\
 & \mid & c \ \mid\ l \ \mid\ t & \text{– constant, location, task} \\
t & ::= & & \text{Tasks} \\
 & \mid & \square\, v \ \mid\ \boxtimes\, \tau \ \mid\ \blacksquare\, l & \text{– editors} \\
 & \mid & t_1 \blacktriangleright e_2 \ \mid\ t_1 \rhd e_2 & \text{– steps} \\
 & \mid & \lightning \ \mid\ t_1 \bowtie t_2 & \text{– fail, combination} \\
 & \mid & t_1 \blacklozenge t_2 \ \mid\ e_1 \lozenge e_2 & \text{– choices}
\end{array}$$

**Figure 8: Value grammar**

The rules to evaluate expressions $e$ do not differ from standard work, except for the task constructs. The evaluation rules for tasks can be deduced from the value grammar. They are given in the appendix[2]. Most task constructors are strict in their arguments. Only steps keep their right hand side unevaluated to delay side effects till the moment the step is taken. The same holds for both branches of the external choice.

[2]https://github.com/timjs/tophat/appendix.pdf

## 5.2 Task observations

The normalisation and interaction semantics make use of observations on tasks. Observations are semantic functions on the syntax tree of tasks. There are four semantic functions: $\mathcal{V}$ for the current task value, $\mathcal{F}$ to determine if a task fails, $\mathcal{I}$ for the currently accepted input events, and a function for generating user interfaces. The semantics make use of $\mathcal{V}$ and $\mathcal{F}$, while $\mathcal{I}$ is used for proving safety. The function for user interfaces is not used by the semantics, but by our implementation. It is only described in passing here.

*Observable values* ($\mathcal{V}$). Task values are used by steps to calculate the successor task. Filled editors are tasks which contain values, as are shared editors. Unvalued editors do not contain values, neither does the fail task. These facts propagate through all other task constructors. The partial function $\mathcal{V}$ associates a value $v$ to tasks $t$ where possible. Its definition is given in Fig. 9.

$$
\begin{array}{lll}
\mathcal{V} : \text{Tasks} \times \text{States} \rightharpoonup \text{Values} & & \\
\mathcal{V}(\square\, v, \sigma) & = & v \\
\mathcal{V}(\boxtimes\, \tau, \sigma) & = & \bot \\
\mathcal{V}(\blacksquare\, l, \sigma) & = & \sigma(l) \\
\mathcal{V}(\lightning, \sigma) & = & \bot \\
\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) & = & \bot \\
\mathcal{V}(t_1 \triangleright e_2, \sigma) & = & \bot \\
\mathcal{V}(t_1 \bowtie t_2, \sigma) & = & \begin{cases} \langle v_1, v_2 \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
\mathcal{V}(t_1 \blacklozenge t_2, \sigma) & = & \begin{cases} v_1 & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \\ v_2 & \textbf{when } \mathcal{V}(t_1, \sigma) = \bot \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
\mathcal{V}(t_1 \lozenge t_2, \sigma) & = & \bot
\end{array}
$$

**Figure 9: Values**

Internal and external steps do not have an observable value, because calculating the value would require evaluation of the continuation. Parallel composition only has a value when both branches have values, in which case these values are paired. Internal choice has a value when one of the branches has a value. When both branches have a value, it takes the value of the left branch. External choice does not have a value because it waits for user input.

*Failing* ($\mathcal{F}$). In Section 4.3 we introduced $\lightning$ to stand for an impossible task. Combinations of tasks can also be impossible. Take for example the parallel composition of two fails ($\lightning \bowtie \lightning$). This expression is equivalent to $\lightning$, because it can not handle input and can not be further normalised. This intuition is formalised by the function $\mathcal{F}$ in Fig. 10. It determines whether a task is impossible. Such tasks are called *failing*.

$$
\begin{array}{lll}
\mathcal{F} : \text{Tasks} \times \text{States} \rightarrow \text{Booleans} & & \\
\mathcal{F}(\square\, v, \sigma) & = & \mathsf{False} \\
\mathcal{F}(\boxtimes\, \tau, \sigma) & = & \mathsf{False} \\
\mathcal{F}(\blacksquare\, l, \sigma) & = & \mathsf{False} \\
\mathcal{F}(\lightning, \sigma) & = & \mathsf{True} \\
\mathcal{F}(t_1 \blacktriangleright e_2, \sigma) & = & \mathcal{F}(t_1, \sigma) \\
\mathcal{F}(t_1 \triangleright e_2, \sigma) & = & \mathcal{F}(t_1, \sigma) \\
\mathcal{F}(t_1 \bowtie t_2, \sigma) & = & \mathcal{F}(t_1, \sigma) \wedge \mathcal{F}(t_2, \sigma) \\
\mathcal{F}(t_1 \blacklozenge t_2, \sigma) & = & \mathcal{F}(t_1, \sigma) \wedge \mathcal{F}(t_2, \sigma) \\
\mathcal{F}(e_1 \lozenge e_2, \sigma) & = & \mathcal{F}(t_1, \sigma_1') \wedge \mathcal{F}(t_2, \sigma_2') \\
 & & \quad \textbf{where } e_1, \sigma \Downarrow t_1, \sigma_1' \textbf{ and } e_2, \sigma \Downarrow t_2, \sigma_2'
\end{array}
$$

**Figure 10: Failing**

Steps whose left hand sides are failing can never proceed because of the lack of an observable value. Therefore they are itself failing. The internal choice of two failing tasks is failing. External choices let the user pick a side and only then evaluate the corresponding subexpression. To determine if an external choice is failing, it needs to peek into the future to check if both subexpressions are failing.

*User interface.* $\widehat{\text{TOP}}$ is designed such that a user interface can be generated from a task's syntax tree. A possible graphical user interface is shown in Fig. 1, where tasks are rendered as HTML pages. Editors are rendered as input fields, external choices are represented by two buttons, and parallel tasks are rendered side by side. Steps only show the interface of their left hand side. In case of an external step they are accompanied by a button. When the guard condition of a step is not fulfilled, the button is disabled.

## 5.3 Normalising tasks

The normalisation semantics is responsible for reducing expressions of type Task until they are ready to handle input. It is a big-step semantics, and makes use of evaluation of the host language. We write $e, \sigma \Downarrow t, \sigma'$ to describe that an expression $e$ in state $\sigma$ normalises to task $t$ in state $\sigma'$.

Normalisation rules are given in Fig. 11. Both rules ensure that expressions are first evaluated by the host language ( $\downarrow$ ), and then by the stride semantics ( $\rightsquigarrow$ ). These two actions are repeated until the resulting state and task stabilise.

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

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

$$
\textsc{N-Repeat}\quad \frac{e, \sigma \downarrow t, \sigma' \qquad t, \sigma' \rightsquigarrow t', \sigma'' \qquad t', \sigma'' \Downarrow t'', \sigma'''}{e, \sigma \Downarrow t'', \sigma'''}\ \sigma' \neq \sigma'' \vee t \neq t'
$$

**Figure 11: Normalisation semantics**

The striding semantics is responsible for reducing internal steps and internal choices. A stride from task $t$ in state $\sigma$ to $t'$ in state $\sigma'$ is denoted by $t, \sigma \rightsquigarrow t', \sigma'$. The rules for striding are given in Fig. 12. Tasks like editors, fail and external choice are not further reduced. For external choice and parallel there are congruence rules.

The split between striding and normalisation is due to mutable references. Consider the following example, where $\sigma = \{l \mapsto \mathsf{False}\}$.

$$(\blacksquare\, l \blacktriangleright \lambda x{:}\textsc{Bool}.\ \textbf{if } x \textbf{ then } e \textbf{ else } \lightning\ ) \bowtie (l := \mathsf{True}; \square\langle\rangle\ )$$

S-And reduces this expression in one step to

$$(\blacksquare\, l \blacktriangleright \lambda x{:}\textsc{Bool}.\ \textbf{if } x \textbf{ then } e \textbf{ else } \lightning\ ) \bowtie (\square\, \langle\rangle\ )$$

with $\sigma' = \{l \mapsto \mathsf{True}\}$. This expression is not normalised, because the left task can take a step. The issue here lies in the fact that the right task updates $l$. To overcome this problem, the N-Done and N-Repeat rules ensure that striding is applied until the state $\sigma$ becomes stable and no further normalisation can take place.

*Principles of stepping.* Stepping away from task $t_1$ can only be performed when $t_1$ has a value: $\mathcal{V}(t_1) = v_1$. Only then can a new task $t_2$ be calculated from the expression $e$. On top of that, $t_2$ must not be failing: $\neg\mathcal{F}(t_2)$. These principles lead to the stepping rules

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

*Step.*

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

S-ThenFail
$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \qquad e_2\ v_1, \sigma' \downarrow t_2, \sigma''}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow t_1' \blacktriangleright e_2, \sigma'}\ \mathcal{V}(t_1', \sigma') = v_1 \land \mathcal{F}(t_2, \sigma'')$$

S-ThenCont
$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \qquad e_2\ v_1, \sigma' \downarrow t_2, \sigma''}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow t_2, \sigma''}\ \mathcal{V}(t_1', \sigma') = v_1 \land \neg\mathcal{F}(t_2, \sigma'')$$

*Choose.*

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

S-OrRight
$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \qquad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow t_2', \sigma''}\ \mathcal{V}(t_1', \sigma') = \bot \land \mathcal{V}(t_2', \sigma'') = v_2$$

S-OrNone
$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \qquad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow t_1' \blacklozenge t_2', \sigma''}\ \mathcal{V}(t_1', \sigma') = \bot \land \mathcal{V}(t_2', \sigma'') = \bot$$

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

S-Xor
$$\frac{}{e_1 \lozenge e_2, \sigma \rightsquigarrow e_1 \lozenge e_2, \sigma}$$

*Congruence.*

S-Next
$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma'}{t_1 \triangleright e_2, \sigma \rightsquigarrow t_1' \triangleright e_2, \sigma'}$$

S-And
$$\frac{t_1, \sigma \rightsquigarrow t_1', \sigma' \qquad t_2, \sigma' \rightsquigarrow t_2', \sigma''}{t_1 \bowtie t_2, \sigma \rightsquigarrow t_1' \bowtie t_2', \sigma''}$$

**Figure 12: Striding semantics**

in Fig. 12. S-ThenStay does nothing, because the left side does not have a value. S-ThenFail covers the case that the left side has a value but the calculated successor task is failing. This rule uses the semantics of the host language to evaluate the application $e_2\ v_1$. When all required conditions are fulfilled, S-ThenCont allows stepping to the successor task.

*Principles of choosing.* Choosing between two tasks $t_1$ and $t_2$ can only be done when at least one of them has a value: $\mathcal{V}(t_1) = v_1 \lor \mathcal{V}(t_2) = v_2$. When both have a value, the left task is chosen. When none has a value, none can be chosen. These principles lead to the rules S-OrLeft, S-OrRight, and S-OrNone, which encode that the choice operator picks the leftmost task that has a value.

## 5.4 Handling user inputs

The handling semantics is the outermost layer of the stack of semantics. It is responsible for performing external steps and choices, and for changing the values of editors. The rules of the interaction semantics are given in Fig. 13. The semantics is only applicable to normalised $t$. Sending an input event $i$ to a task $t$ first handles the event and then prepares the resulting task for the next input by normalising it.

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

I-Handle
$$\frac{t, \sigma \xrightarrow{i} t', \sigma' \qquad t', \sigma' \Downarrow t'', \sigma''}{t, \sigma \overset{i}{\Longrightarrow} t'', \sigma''}$$

**Figure 13: Interaction semantics**

Inputs $i$ are formed according to the grammar in Fig. 14. F and S in an input encode the path to the task for which the input is designated. There is a function $\mathcal{I}$ which calculates the possible input events a given task expects. It takes a normalised task and a state and returns a set of inputs that can be handled. The definition of this function is listed in Fig. 15.

$$\begin{array}{rlll}
i ::= & & & \text{Inputs} \\
 & | & a \mid \mathsf{F}\, i \mid \mathsf{S}\, i & \text{– action, pass to first, pass to second} \\
a ::= & & & \text{Actions} \\
 & | & v \mid \mathsf{C} & \text{– change, continue} \\
 & | & \mathsf{L} \mid \mathsf{R} & \text{– go left, go right}
\end{array}$$

**Figure 14: Input grammar**

$$\begin{array}{lll}
\mathcal{I} : \text{Tasks} \times \text{States} \to \mathcal{P}(\text{Inputs}) & & \\
\mathcal{I}(\square\, v, \sigma) & = \{v' \mid \varnothing \vdash v' : \tau\} \cup \{\mathsf{E}\} & \textbf{where } \square\, v : \textsc{Task}\ \tau \\
\mathcal{I}(\boxtimes\, \tau, \sigma) & = \{v' \mid \varnothing \vdash v' : \tau\} & \\
\mathcal{I}(\blacksquare\, l, \sigma) & = \{v' \mid \varnothing \vdash v' : \tau\} & \textbf{where } \blacksquare\, l : \textsc{Task}\ \tau \\
\mathcal{I}(\lightning, \sigma) & = \varnothing & \\
\mathcal{I}(t_1 \blacktriangleright e_2, \sigma) & = \mathcal{I}(t_1, \sigma) & \\
\mathcal{I}(t_1 \triangleright e_2, \sigma) & = \mathcal{I}(t_1, \sigma) \cup \{\mathsf{C} \mid \mathcal{V}(t_1, \sigma) = v_1 \land & \\
 & \quad e_2\ v_1, \sigma \Downarrow t_2, \sigma' \land \neg\mathcal{F}(t_2, \sigma')\} & \\
\mathcal{I}(t_1 \bowtie t_2, \sigma) & = \{\mathsf{F}\ i \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{\mathsf{S}\ i \mid i \in \mathcal{I}(t_2, \sigma)\} & \\
\mathcal{I}(t_1 \blacklozenge t_2, \sigma) & = \{\mathsf{F}\ i \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{\mathsf{S}\ i \mid i \in \mathcal{I}(t_2, \sigma)\} & \\
\mathcal{I}(e_1 \lozenge e_2, \sigma) & = \{\mathsf{L} \mid e_1, \sigma \Downarrow t_1, \sigma' \land \neg\mathcal{F}(t_1, \sigma')\} \cup & \\
 & \quad \{\mathsf{R} \mid e_2, \sigma \Downarrow t_2, \sigma' \land \neg\mathcal{F}(t_2, \sigma')\} &
\end{array}$$

**Figure 15: Inputs**

Handling input is done by the *handling* semantics shown in Fig. 16. It is a small step semantics with labelled transitions. It takes a task $t$ in a state $\sigma$ and an input $i$, and yields a new task $t'$ in a new state $\sigma'$.

H-Change, H-Fill, H-Update: Input events $v$ are used to change the value of editors. Editors only accept values of the correct type.

H-Next: The C(ontinue) action triggers an external step. As with internal stepping, this is only possible if the left side has a value and the continuation is not failing.

H-PickLeft, H-PickRight: L and R are used to pick the left or right option of an external choice.

H-PassThen, H-PassNext: The step combinators pass all events other than C to the left side.

H-FirstAnd, H-SecondAnd, H-FirstOr, H-SecondOr: Inputs F(irst) and S(econd) are used to direct inputs to the correct branch of parallel combinations.

## 5.5 Implementation

The semantics have been implemented in the Haskell programming language [16]. We use techniques presented by Jaskelioff et al. [10], Swierstra [27], and Peyton Jones [21]. The source code can be found

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

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

H-PickLeft
$$\frac{e_1, \sigma \Downarrow t_1, \sigma'}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{L}} t_1, \sigma'}\; \neg \mathcal{F}(t_1, \sigma')$$

H-PickRight
$$\frac{e_2, \sigma \Downarrow t_2, \sigma'}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{R}} t_2, \sigma'}\; \neg \mathcal{F}(t_2, \sigma')$$

*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-PassNext
$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \triangleright e_2, \sigma \xrightarrow{i \neq \mathsf{C}} t_1' \triangleright e_2, \sigma'}$$

H-FirstAnd
$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{F}\, i} t_1' \bowtie t_2, \sigma'}$$

H-SecondAnd
$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{S}\, i} t_1 \bowtie t_2', \sigma'}$$

H-FirstOr
$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{F}\, i} t_1' \blacklozenge t_2, \sigma'}$$

H-SecondOr
$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{S}\, i} t_1 \blacklozenge t_2', \sigma'}$$

**Figure 16: Handling semantics**

on GitHub.[3] A command-line interface is part of this implementation. It prompts users to type input events, which get parsed and processed by the interaction semantics.

Also, we made an implementation of $\widehat{\text{TOP}}$ combinators on top of iTasks, so that $\widehat{\text{TOP}}$ specifications can be compiled to runnable applications. This shows that $\widehat{\text{TOP}}$ is a subset of iTasks.

## 6 PROPERTIES

In order to show our semantics is sane, we show that our evaluation, normalisation and handling semantics is type preserving. We additionally prove a progress theorem for our small-step handling semantics. We show that our failing function $\mathcal{F}$ indeed only indicates expressions that can not be normalised and that allow no further interaction. Finally, we prove that the function to compute all possible inputs $\mathcal{I}$ is sound and complete.

### 6.1 Type preservation

We show that the following three preservation Theorems hold.

Theorem 6.1 (Type preservation under evaluation). *For all expressions $e$ and states $\sigma$ such that $\Gamma, \Sigma \vdash e : \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $e, \sigma \downarrow e', \sigma'$, then $\Gamma, \Sigma \vdash e' : \tau$ and $\Gamma, \Sigma \vdash \sigma'$.*

Where $\Gamma, \Sigma \vdash \sigma$ means that for all $l \in \sigma$, it holds that $\Gamma, \Sigma \vdash \sigma(l) : \Sigma(l)$.

Theorem 6.2 (Type preservation under normalisation). *For all expressions $e$ and states $\sigma$ such that $\Gamma, \Sigma \vdash e : \textsc{Task}\, \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $e, \sigma \Downarrow e', \sigma'$, then $\Gamma, \Sigma \vdash e' : \textsc{Task}\, \tau$ and $\Gamma, \Sigma \vdash \sigma'$.*

Theorem 6.3 (Type preservation under handling). *For all expressions $e$, states $\sigma$ and inputs $i$ such that $\Gamma, \Sigma \vdash e : \textsc{Task}\, \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $e, \sigma \xrightarrow{i} e', \sigma'$, then $\Gamma, \Sigma \vdash e' : \textsc{Task}\, \tau$ and $\Gamma, \Sigma \vdash \sigma'$.*

All three Theorems are proven to be correct by induction over $e$. The full proofs are listed in the appendix[4]. From Theorem 6.3 and Theorem 6.2 we directly obtain that the driving semantics also preserves types.

### 6.2 Progress

A well-typed term of a task type is guaranteed to progress after normalisation, unless it is failing.

We define what we mean with progress in Theorem 6.4.

Theorem 6.4 (Progress under handling). *For all well typed expressions $e$ and states $\sigma$, if $e, \sigma \Downarrow e', \sigma'$, then either $\mathcal{F}(e', \sigma')$ or there exist $e''$, $\sigma''$, and $i$ such that $e', \sigma' \xrightarrow{i} e'', \sigma''$.*

Where a well typed expression $e$ means that $\Gamma, \Sigma \vdash e : \tau$ for some type $\tau$, and a well typed state means that $\Sigma \vdash \sigma$.

If an expression $e$ and state $\sigma$ are well-typed, then after normalisation, the pair $e', \sigma'$ either fails, or there exists some input $i$ that can be handled by it under the handling semantics. In order to prove this Theorem, we need to show that the failing function $\mathcal{F}$ behaves as desired.

Theorem 6.5 (Failing means no interaction possible). *For all well typed expressions $e$ and states $\sigma$, and $e, \sigma \Downarrow e', \sigma'$, we have that $\mathcal{F}(e', \sigma') = \mathsf{True}$, if and only if there is no input $i$ such that $e', \sigma' \xrightarrow{i} e'', \sigma''$ for some $e''$ and $\sigma''$.*

The Theorem above states that an expression $e$ and state $\sigma$ are failing, if, after normalisation, there exists no input that can be handled by it. We prove the theorem to be true by induction on $e'$.

We now have the ingredients to prove Theorem 6.4.

Proof. Given $\Gamma, \Sigma \vdash e : \textsc{Task}\, \tau$ and $\Sigma \vdash \sigma$ and after normalisation $e, \sigma \Downarrow e', \sigma'$, we find ourselves in either one of the following situations:

There exists an $i$ such that $e', \sigma' \xrightarrow{i} e'', \sigma''$.

There does not exist an $i$ such that $e', \sigma' \xrightarrow{i} e'', \sigma''$. In this case, we know that $\mathcal{F}(e', \sigma')$ must be true, by Theorem 6.5. □

### 6.3 Soundness and Completeness of Inputs

In order to validate the function that calculates all possible inputs $\mathcal{I}$, we want to show that the set of possible inputs it produces is both sound and complete with respect to the handle semantics. By sound we mean that all inputs in the set of possible inputs can actually be handled by the handle semantics, and by complete we mean that the set of possible inputs contains all inputs that can be handled by the handle semantics. Theorem 6.6 expresses exactly this property.

[3] https://github.com/timjs/tophat-haskell

[4] https://github.com/timjs/tophat/appendix.pdf

Theorem 6.6 (Inputs function is sound and complete). *For all well typed expressions $e$, states $\sigma$, and inputs $i$, we have that $i \in \mathcal{I}(e, \sigma)$ if and only if $e, \sigma \xrightarrow{i} e', \sigma'$.*

We prove the above theorem by induction over $e$. The proof is listed in the appendix.

## 6.4 Outlook

At this point we have specified a formal language for task-oriented programming, given its semantics, and proved its safety. The main motive to formalise this paradigm, is to be able to reason about tasks. In future work, we plan on utilising the formalisation to do so. Firstly, we would like to express properties of tasks and prove them. For example, one would like to prove that, no matter what, in Example 3.1 breakfast is always being served. Secondly, we would like to explore what it means for two tasks to be equal. One could have noticed that some operators have a monadic or applicative feeling. The combination of $\bowtie$ and $\square$ could form a (lax) monoidal functor, $\blacklozenge$ is similar to applicative choice, and $\blacktriangleright$ looks like a bind operation. We need a correct understanding of equivalence of tasks, taking the interactive setting into account, to prove this. Thirdly, we do not know yet if the more complex combinators of iTasks are expressible in the basic combinators of $\widehat{\text{TOP}}$. We implemented $\widehat{\text{TOP}}$ on top of iTasks, so we know it is a subset, but we also know iTasks can do more. A more in depth description of future work can be found in Section 8.

## 7 RELATED WORK

The work presented in this paper lies on the boundary of many areas of study. People have looked at the problem of how to model and coordinate collaboration from many different perspectives. The following subsections give an overview of related work from the many different areas.

## 7.1 TOP implementations

*iTasks.* As mentioned earlier, iTasks is an implementation of TOP. iTasks has many features, and its basic combinators are versatile and powerful. Simpler combinators are implemented by restricting the powerful ones. This is useful for everyday programming, where having lots of functionality at one's fingertips is convenient and efficient. $\widehat{\text{TOP}}$ on the other hand does not include the many different variations of the step- and parallel combinators of iTasks. To name two examples, the combinators `(>>|)` and `(||-)` are variations of step and parallel that ignore the value of the left task.

There have been two previous papers that describe semantics of iTasks, by Koopman et al. [14] and Plasmeijer et al. [23]. Both give a different semantics in the form of minimal implementations of a subset of the interface of iTasks. These semantics however do not make an explicit distinction between the host language and task language and they do not provide an explicit formal semantics. Therefore, the do not lend itself well for formal reasoning.

*mTasks.* The mTasks framework [13] is an implementation of TOP geared towards IOT devices. As $\widehat{\text{TOP}}$ its basic combinators are a subset of iTasks. They are similar to those of $\widehat{\text{TOP}}$. However, on IOT devices it is useful to continue running tasks endlessly, which is done in mTasks using a `forever` combinator. This is currently not possible in $\widehat{\text{TOP}}$.

As for iTasks, there is currently no formal semantics for mTasks.

## 7.2 Worfklow modelling

Much research has been done into workflow modelling. This work focusses on describing the collaboration between subsystems, rather than the communication between them. The systems described in the literature follow a *boxes and arrows* model of specifying workflows. Control flow, represented by arrows, usually can go unrestricted from anywhere to anywhere else in a workflow. We see TOP as the functional programming of workflows, as opposed to this *goto*-style.

*Workflow patterns.* Workflow patterns are regarded as special kind of the design patterns in software engineering. They identify recurring patterns in workflow systems, much like the combinators defined by $\widehat{\text{TOP}}$. Work by van der Aalst et al. [30] defines a comprehensive list of these pattens, and examines their availability in industry workflow software. Workflow patterns are usually described in terms of control flow graphs, and no formal specification is given, which makes comparison and formal reasoning more difficult.

*Workflow Nets & YAWL.* Workflow Nets (WFN) [28] allow for the modelling and analysis of business processes. They are graphical in nature, and clearly display how every component is related to each other. A downside of WFN is that they do not facilitate higher order constructs. Also, they are often not directly executable.

A language based on WFN that is actually directly executable is YAWL by van der Aalst and ter Hofstede [29]. It facilitates modelling and execution of dynamic workflows, with support for *and*, *or* and *xor* workflow patterns. As mentioned, YAWL programs consist of WFN, and are therefore programmed visually.

*BPEL.* BPEL [20] is another popular business process calculus. The standardised language allows for the specification of actions within business processes, using an XML format. The language is mainly used for coordinating web services. Two workflow patterns are supported; execution of services can be done sequential or in parallel. On top of that, processes can be guarded by conditionals. There is no support for higher order processes however. Processes described in BPEL can be regarded as activity graphs, and they can also be rendered as such. The specified processes in BPEL are directly executable, just like YAWL.

## 7.3 Process algebras

*Differences.* There are two main differences between TOP and process algebras. The first is a difference in scope. Process algebras focus on modelling the input/output behaviour of processes, by explicitly stating which actions are sent and received at certain points in the program. The goal of process algebras is formal reasoning about the interaction between processes. Typically, one wishes to prove properties such as deadlock-freedom, liveness, or adherence to a protocol specification.

The focus of TOP on the other hand is to model collaboration patterns, with the explicit goal of not having to specify how exactly subtasks communicate. The declarative specification of data dependencies between subtasks enables TOP to hide such details.

The second difference concerns internal communication. There are two forms of communication between tasks: Passing values to continuations and sharing data. This is different from communication in process algebras, which is based on message-passing.

*Similarities.* There are some aspects that are similar in $\widehat{\text{TOP}}$ and process algebras. Internal communication in Hoare's CSP [9] is introduced with the concealment operator. The semantics of CSP requires that all concealed actions are handled to exhaustion before any action with the environment can take place. This is somewhat similar to $\widehat{\text{TOP}}$, where all enabled internal steps must be taken until the system can react to input events again. Contrast this with Milner's CCS [18], where concealed actions are visible to the outside as $\tau$-actions, and can be interleaved with external communication.

Another similarity between $\widehat{\text{TOP}}$ and process algebras, or any system with concurrency for that matter, is the need for synchronisation. Broadly speaking, concurrency means that different parts of a program can interact with the environment independently, in an interleaved manner. Synchronisation means that only some, but not all, of the possible interleavings are desirable. The semantics of the step combinators in $\widehat{\text{TOP}}$, together with the fact that internal communication happens atomically, allows for concise and intuitive synchronisation code.

## 7.4 Reactive programming

*HipHop & Esterel.* HipHop [3, 4] is a programming language tailored to the development of synchronous reactive web systems. From a single source, both server and client applications can be generated. Programs are written in the Hop language, a Scheme dialect. Communication is based on a reactive layer embedded in Hop. The set of HipHop reactive statements is based on those of the Esterel language [2, 5]. Each reactive component starts by specifying possible input and output events. The component then proceeds as a state machine.

Input events are sent to such a machine programmatically using Hop, or are explicitly wired to events from the client. They are optionally associated with a Hop value. As Hop is a dynamic language, and HipHop uses strings to identify events, events and their possible associated values are not statically checked. Events are aggregated until the moment the machine is asked to react. The machine is executed and reacts by building a multi-set of output events. The execution of a HipHop machine is atomic. The set of inputs is not influenced by the current computations.

As with $\widehat{\text{TOP}}$, HipHop is a DSL embedded in a general purpose programming language. Another similarity is that both specifications lead to executable server and client applications from a single source. However, both HipHop and Esterel are more low level regarding their specification. Where $\widehat{\text{TOP}}$ takes tasks and collaboration as a starting point, HipHop focusses on synchronous communication and atomic execution of reactive machines.

This difference in focus shows in the way both systems define events. In HipHop programmers can define and use their own events. Inputs in $\widehat{\text{TOP}}$ are not extensible and not visible to the developer. They are a completely separate entity living on the semantic level.

Another important difference is the way in which both systems handle events. In HipHop the programmer decides when a machine should process its events. This could be just one event, or a multi-set of events that are processed simultaneously. $\widehat{\text{TOP}}$ always processes an input the moment it occurs and only handles a single event in one instance.

*Functional reactive programming.* Functional Reactive Programming (FRP) is a paradigm to describe dynamic changes of values in a declarative way. This is done by specifying networks of values, called behaviours, that can depend on each other and on external events. Behaviours can change over time, or triggered by events. When a behaviour changes, all other behaviours that depend on it are updated automatically. The underlying implementation that takes care of the updating usually can tie input devices, like mouse and keyboard, to event streams and behaviours to output facilities, like text fields. This allows for declarative specifications of applications with user interfaces.

The idea of FRP was pioneered by Elliott and Hudak [8]. In the meantime there are many variants and implementations, where reactive-banana [1], FrTime [6], and Flapjax [17] belong to the most well-known.

FRP and TOP are different systems that have different goals in mind. Whereas FRP expresses automatically updating data dependencies, TOP expresses collaboration patterns. TOP has no notion of time. Tasks cannot change spontaneous over time, while behaviours can. Only input events can change task values. The biggest conceptual difference between a workflow in TOP and a data network in FRP is that an event to a task only causes updates up until the next step, while an event in FRP propagates through the whole network.

That being said, there are some concepts that are similar in TOP and FRP. The *stepper* behaviour, for example, is associated with an event and yields the value of the most recent event. This is similar to editors in TOP. Furthermore, both systems can be used to declaratively program user interfaces, albeit in FRP the programmer has to construct the GUI elements manually, and connect inputs and outputs to the correct events and behaviours. In TOP graphical user interfaces are automatically derived.

## 7.5 Session types

Session types are a type discipline that can be used to check whether communicating programs conform to a certain protocol. Session types are expressions in some process calculus that describe the input/output behaviour of such programs. Session types are useful for programming languages where modules communicate with each other via messages, like CSP, $\pi$-calculus, or Go, to name a few. The only form of messages in TOP are input events which drive execution, but modules do not communicate using messages. Therefore, session types are not applicable to TOP in the sense used in the literature.

Formal reasoning about TOP programs is one of our future goals for $\widehat{\text{TOP}}$. The ideas and techniques of session types could be useful for specifying that a list of inputs of a certain form leads to desired task values. The details are a topic for future work.

## 8 CONCLUSION

In this paper we have identified and intuitively described the essence of task-oriented programming. We then formalised this essence by developing a domain-specific language for declarative interactive workflows, called $\widehat{\text{TOP}}$. The task language and the host language are clearly separated, to make explicit where the boundaries are. The semantics of the task layer is driven by user input. We have compared $\widehat{\text{TOP}}$ with workflow modelling languages, process algebras, functional reactive programming and session types to point out differences and similarities. Finally, we have proven type safety and progress for our language.

*Future work.* There are a couple of ways in which we would like to continue this line of work.

One of the main motivations to formalise task-oriented programming is to be able to reason about programs. In this paper we reason about the language itself, but it would be nice to prove properties about individual programs. To this end, we are very interested to see if it is possible to develop an axiomatic semantics for $\widehat{\text{TOP}}$ that allows us to do so. There are certain properties of our language that make this particularly complex: We have to deal with parallelism, user interaction, and references.

We would also like to prove whether certain programs are equivalent, for example to show that the monad laws hold for our step combinator. This requires a notion of equality, which in the presence of side effects most certainly needs some form of coalgebraic input-output conformance. We have implemented the reduction semantics of our language in Haskell, whose type system could aid in the formalisation of such proofs.

Another form of reasoning about programs is static analysis. Klinik et al. [11] have developed a cost analysis for tasks that require resources in order to be executed. This analysis was developed for a simpler task language, and could be brought over to the one developed here.

Naus and Jeuring [19] have looked at building a generic feedback system for rule-based problems. A workflow system typically is rule based, as outlined in their work. It would be interesting to fit the generic feedback system to $\widehat{\text{TOP}}$ in order to support end-users working in applications developed in this language.

Additionally, we would like to develop visualisations for $\widehat{\text{TOP}}$ language constructs. An assistive development environment integrating these visualisations and the presented textual language would aid domain experts to model workflows in a more accessible manner. A system that visualises iTask programs has been developed in the past [26].

## ACKNOWLEDGMENTS

The authors are grateful to Johan Jeuring, Sjaak Smetsers, and Andreas Vinter-Hviid for fruitful discussions, and Rinus Plasmeijer, Peter Achten and Pieter Koopman for proofreading.

This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

## REFERENCES

[1] Heinrich Apfelmus. 2019. reactive-banana. https://wiki.haskell.org/Reactive-banana. [Accessed 13-Febuary-2019].

[2] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (1992), 87–152.

[3] Gérard Berry, Cyprien Nicolas, and Manuel Serrano. 2011. HipHop: A synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients.* ACM, 49–56.

[4] Gérard Berry and Manuel Serrano. 2013. Hop and HipHop : Multitier Web Orchestration. *CoRR* abs/1312.0078 (2013).

[5] Frédéric Boussinot and Robert De Simone. 1991. The Esterel language. *Proc. IEEE* 79, 9 (1991), 1293–1304.

[6] Gregory Cooper and Shriram Krishnamurthi. 2004. *FrTime: Functional Reactive Programming in PLT Scheme.* Technical Report CS-03-20. Department of Computer Science, Brown University, Rhode Island.

[7] Allen B. Downey. 2008. *The Little Book of Semaphores.* Green Tea Press.

[8] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.* 263–273.

[9] C. A. R. Hoare. 1985. *Communicating Sequential Processes.* Prentice Hall Int'l.

[10] Mauro Jaskelioff, Neil Ghani, and Graham Hutton. 2011. Modularity and Implementation of Mathematical Operational Semantics. *Electr. Notes Theor. Comput. Sci.* 229, 5 (2011), 75–95.

[11] Markus Klinik, Jan Martin Jansen, and Rinus Plasmeijer. 2017. The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages, IFL 2017.* ACM.

[12] Bram Kool. 2017. Integrated Mission Management voor C2-ondersteuning. Bachelor's Thesis. Dutch Defence Academy, Den Helder, The Netherlands.

[13] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A task-based dsl for microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop.* 4.

[14] Pieter W. M. Koopman, Rinus Plasmeijer, and Peter Achten. 2008. An Executable and Testable Semantics for iTasks. In *Proceedings of the 20th Symposium on Implementation and Application of Functional Programming Languages, IFL 2008.*

[15] Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. 2012. Incidone: A Task-Oriented Incident Coordination Tool. In *Proceedings of ISCRAM.*

[16] Simon Marlow et al. 2010. *Haskell 2010 language report.*

[17] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA.* 1–20.

[18] Robin Milner. 1989. *Communication and concurrency.* Prentice Hall.

[19] Nico Naus and Johan Jeuring. 2016. Building a Generic Feedback System for Rule-Based Problems. In *Trends in Functional Programming - 17th International Conference, TFP 2016, College Park, MD, USA, June 8-10, 2016, Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 10447. Springer, 172–191.

[20] OASIS. 2019. Web Services Business Process Execution Language. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel. [Accessed 12-Febuary-2019].

[21] Simon Peyton Jones. 2001. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdorf Summer School 2000, Germany.*

[22] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

[23] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2012. Task-oriented programming in a pure functional language. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium, 2012.*

[24] Jurriën Stutterheim. 2017. *A Cocktail of Tools.* Ph.D. Dissertation. Radboud University, Nijmegen, The Netherlands.

[25] Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2017. Maintaining Separation of Concerns Through Task Oriented Software Development. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK.*

[26] Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. 2014. Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 8843. Springer, 122–141.

[27] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436.

[28] Wil M. P. van der Aalst. 1998. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers* 8, 1 (1998), 21–66.

[29] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. 2005. YAWL: yet another workflow language. *Inf. Syst.* 30, 4 (2005), 245–275.

[30] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. 2003. Workflow Patterns. *Distributed and Parallel Databases* 14, 1 (2003), 5–51.