A Reflection on Task-Oriented Programming

Mart Lubbers
Radboud University
Nijmegen, Netherlands
mart@cs.ru.nl

Tim Steenvoorden
Open University
Heerlen, Netherlands
tim.steenvoorden@ou.nl

Abstract

Task-oriented programming (TOP) is a declarative programming paradigm where the main building blocks are tasks. Tasks represent work and have an observable task value. Tasks are combined to form compositions of tasks. From this specification of work, a ready-for-work application can be derived automatically.

There are several implementations of task-oriented programming, for example iTask, an industry-grade TOP system for distributed web applications; and TopHat, a fully formalised task-oriented language. iTask and TopHat differ a lot in philosophy. The iTask language only has three complex super combinators from which every other combinator is derived. This makes it difficult to provide a formal semantics for them. In TopHat more complex combinators are built from a rich set of simple building blocks, core combinators. Consequently, defining a formal semantics is easier.

By definition, the super combinators of iTask are more expressive than TopHat, as they allow the programmer to use the full host language Clean to define the behaviour. Whereas in TopHat, one has to create the behaviour by combining simple core combinators. The contribution of the paper is threefold, we perform a qualitative and quantitative analysis of task combinator usage. From that, we identify gaps between iTask and TopHat. Finally we introduce a new combinator, *reflect* (◎) to bridge a gap.

CCS Concepts

• **Software and its engineering** → **Language features**; **Very high level languages**.

Keywords

Task-Oriented Programming, Functional Programming, Workflow Modelling

ACM Reference Format:
Mart Lubbers and Tim Steenvoorden. 2025. A Reflection on Task-Oriented Programming. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming (PPDP '25), September 10–11, 2025, Rende, Italy.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3756907.3756913

1 Introduction

Task-oriented programming (TOP) is a relatively new programming paradigm to model workflows [37]. It is a declarative programming paradigm where the basic building blocks are *tasks*. Tasks are an abstract representation of work and only describe *what* work needs to be done, the *how* is derived from this specification. Tasks have an *observable task value*. I.e. during the execution of a task, other tasks can observe the progress of the task and make decisions accordingly. Task values are observed by other tasks using *task combinators*. There is a rich set of task combinators that allow the composition of tasks. For example, tasks can be composed sequentially or parallel to form complex workflow systems. Besides exposing the progress of a task *following* control flow, tasks can share information *across* control flow using *Shared Data Sources* (SDSs).

There are several implementations of TOP, for example iTask, an industry-grade TOP system for distributed web applications; toppyt, a TOP implementation in Python [21, 22]; mTask, a TOP language for microprocessors that integrates with iTask [17, 26]; and TopHat, a fully formalised TOP language. The first and the last are the focus of this paper.

The iTask system is a TOP implementation that generates an interactive multi-user distributed web server that facilitates users to perform the specified work [34]. It is implemented as an embedded domain-specific language (eDSL) in the purely functional host language Clean [7, 38].[1] It has a long history and the set of task combinators changed continuously throughout the years [19]. The philosophy behind iTask is that with three super combinators, all other combinators are derived. This means that there is only one sequential super combinator (`step`), one parallel super combinator (`parallel`), and one transform combinator (`transformError`). As a consequence, deriving new combinators is relatively easy, but understanding or changing the exact semantics of the super combinators is difficult, as they define a mixture of complex behaviours. Attempts to define a semantics have been made but always only on a subset of iTask [18, 37]. There are many documented case studies in literature and it is used in industry, resulting in a relatively large codebase of real-world TOP applications.

TopHat is a TOP implementation that is fully mathematically formalised [44]. This formalisation is used to drive a symbolic execution engine on tasks which supports next-step hints generation [30, 31]. It is also the basis for reasoning about equivalence of tasks [16]. The design of task combinators in TopHat is exactly opposite of iTask. Instead of deriving all combinators from three complex super combinators, there is a rich set of core combinators from which more complex ones can be derived. Over the years, the set of core combinators of TopHat has been extended to cover more and more real-world workflow patterns. For example, allowing users to dynamically spawn more tasks [43].

Motivation. Comparing the expressive power of iTask and TopHat is difficult due to their philosophical differences and their implementation details. Still, we would like to identify the feature

[1] A guide to Clean for Haskell programmers can be found in Lubbers and Achten [27].

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7/25/09
https://doi.org/10.1145/3756907.3756913

gaps between these two TOP implementations. Many features in iTask grew over time and their theoretical usefulness is not directly apparent. Basing our knowledge on real-world usage is therefore paramount. Then, we can make a well founded decision on extending TopHat and providing formal semantics for missing features.

Contributions. The research contribution of this paper is threefold. First, we identify combinator usage of real-world TOP programs by analysing thirteen iTask's published case studies, two of iTask's internal workflow applications, and one real-world industrial application. Then, using this information, we establish the gaps between iTask and TopHat. For each of these gaps, we discuss the impact on TOP code and the necessity to formalise these features in TopHat. We conclude that task value reflection, detaching of tasks, exceptions, and advanced parallel combinator usage are features of iTask that are currently not supported by TopHat. Finally, we narrow the gap between iTask and TopHat in two ways. We identify a derived combinator *censor* ($\lfloor t \rfloor$) and we add task value reflection to TopHat by extending the language with a new *reflect* (◎) core combinator. We give its complete semantics and derive the reflect combinators seen in the real-world iTask examples from it.

Structure. The remaining of this paper is structured as follows. First, in Section 2 we introduce TOP and its concepts by means of TopHat. We discuss the language, its components and combinators, and its semantics. Next, in Section 3, we describe our method and results for analysing real-world TOP applications. For each category of combinators, we describe similarities and differences between TopHat and iTask. Then, in Section 4, we dive into the usage and semantics of reflection combinators and extend TopHat with a new combinator supporting these workflow patterns. Section 5 discusses related work and Section 6 concludes our paper.

2 Task-oriented programming in TopHat

In this section we discuss the semantics of task-oriented programs by means of TopHat [45]. TopHat is a formal specification of task-oriented programming, with a verified implementation in Idris[2] and a practical one in Haskell.[3] It specifies the semantics of basic task-oriented operations. The framework has been extended for symbolic execution of tasks [31], and next-step hint generation [30]. Also, it is the foundation of proving equivalence of task definitions [16].

2.1 Host and task languages

The TopHat language consists of two two parts: the *host language* and the *task language*. TopHat's host language is the simply typed λ-calculus with *basic types* β such as booleans, integers, and strings, and product and sum types thereof. It also contains *addresses* a, which are values on the host layer and can only be manipulated on the task layers. Most importantly, our host language has no operation for general recursion, and addresses are restricted to only contain basic types, that is, no functions nor other addresses. This means, evaluation of λ terms is pure and total.

On top of the simply typed λ-calculus, TopHat builds a task language. Its grammar is given in Figure 1. Terms v are *values* and

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

		Editors	
d	::=	$\square^{\nu}\beta \mid \boxminus^{\nu} b \mid \boxdot^{\nu} b$	– unvalued, valued, read-only
	$\mid$	$\boxplus^{\nu} a \mid \boxplus^{\nu} a$	– shared, read-only
		Tasks	
t	::=	$d \mid \blacksquare v \mid \lightning$	– editor, done, fail
	$\mid$	$v_1 \bullet t_2 \mid t_1 \blacktriangleright v_2$	– transform, step
	$\mid$	$t_1 \bowtie t_2 \mid t_1 \blacklozenge t_2$	– pair, choose
	$\mid$	$\rhd\lhd^{\nu}_{t_0} [\bar{t}]$	– pool
	$\mid$	**share** $b \mid a_1 := b_2$	– share, assign

Figure 1: Grammar of TopHat's task language.

b *basic values* of the host language. In the following paragraphs, we discuss the operators in the task language. For more details about types and expressions in the host language, we refer to previous work [42].

2.2 Editors

Editors are the endpoints of tasks, used to interact with end users. They are an abstraction over input fields or widgets, as seen in web frameworks and TOP toolkits. Editors are typed, which means that in an Int editor, users can really only fill in integers.

Editors come in multiple flavours. *Unvalued editors* $\square\beta$ do not contain a value yet. They need to be filled with a value of the appropriate type β. *Valued editors* $\boxminus b$ do contain a value b, which can be modified by users. *Read-only editors* $\boxdot b$ also contain a value b, but can only be viewed by users and not modified. We discuss editors on shared data in Section 2.4.

To distinguish editors at runtime, they are *named* by unique names ν. Running programs receive *inputs* which are basic values together with the unique name of the editor ment to receive the value. In examples, we omit names on editors if they are not needed in that context.

Example 2.1 (Editors). The simple program $t_0 = \square^{k_1}\text{String}$ is a task to fill in data of type String. Users see this as an empty input field on their screen. Typing in the the input "*Helloreader*" in this field, sends the input k_1!"*Helloreader*" to program t_0. TopHat's semantics will then rewrite program t_0 to $t_1 = \boxminus^{k_1}$"*Helloreader*". This process of entering string data can continue indefinitely.

2.3 Combinators

TopHat's combinators join smaller task into bigger ones. Combinators come in two main forms: sequential and parallel.

The main sequential operator is a *step* from task t_1 to its continuation v_2, denoted by $t_1 \blacktriangleright v_2$. Here, v_2 is a pure function which calculates the next task to be performed based on its arguments. When this calculated task happens to be the *fail* task ($\lightning$), the step is not made and we stay working on t_1.

A task to check age restrictions, could be programmed as follows.

$$\square\text{Int} \blacktriangleright \lambda n.\ \textbf{if}\ n < 18\ \textbf{then}\ \lightning\ \textbf{else}\ \square\text{FormData}$$

This task indefinitely asks to enter an age, and only continues to the next task of filling out the form data if the entered number is higher than 18.

The transform combinator $v_1 \bullet t_2$ *transforms* the observable value of a task t_2 into something new by mapping the pure function v_1 over the value of t_2. As an example, take the task *trafficLight* below.

type Light = [Green, Red]

let *trafficLight* = λ**x**.(**if x then** Green **else** Red) • □Bool

This initially does not have an observable value. It asks users to enter a boolean True or False, but returns one of the colours Green or Red by mapping a lambda over the task □Bool.

The parallel combinators also come in three forms: pair, choose, and pool. *Pairing* two task $t_1 \bowtie t_2$ let us work on both t_1 and t_2 interleaved. The observed value of both tasks is combined in a tuple if both are available, otherwise, it does not have a value. So, ⧈"*Cat*" ⋈ ⊟**True** has the observable value {"*Cat*", **True**}, but ⧈"*Cat*" ⋈ □Bool has no initial observable value.

Choosing between two tasks, $t_1 \blacklozenge t_2$, also means we can work on both tasks interleaved. However, the observed value is the value of t_1 if it is available, otherwise, it chooses the value of t_2. For example, □Int♦ ⊟ 42 immediately normalises to ⊟42, because □Int does not have an observable value. If both values are unavailable, this also does not have a value.

Both pairing and choosing are *static* combinators, i.e. the number of tasks to work on is specified at development time and tasks cannot be added or deleted at runtime. This can be achieved by using *task pools*, denoted $\triangleright\!\triangleleft^{v}_{t_0}\ [\bar{t}]$. Task pools can receive inputs from users, therefore they need to be identifiable by a name v. They are parametrised by a *template task* t_0. Upon receiving an add-input, this template task is added to the task list $[\bar{t}]$. Task in this list can also be dynamically removed by sending a remove-input.

2.4 Sharing data

Note that, till now, data could only be passed from task to task sequentially: when groups of tasks finish, resulting values can be used to calculate continuations. This is too restrictive to describe general workflow systems, where parallel workflows need to react on data from each other. Therefore, data in TOP specifications can be shared.

Shared data is introduced by **share** b, which allocates the basic value b in memory and returns its address a. Using this a, multiple tasks can watch the same data. For example, *shared editors* ⊞a watch address a, show the data at a to end users, and allow them to change it. Similarly, *read-only shared editors* ▣ a also watch an address a, but end users cannot change its data. The application itself can set addresses to any basic value using $a_1 := b_2$.

2.5 Observations

Tasks form syntax trees which can be *observed*. The most important observation on tasks is their current *value*. This is a partial function $\mathcal{V}$ from task trees to values. For example, unvalued editors like □Bool do not have a value, while valued editors like ⊟**True** have a value. Value observations are defined recursively on task trees. Notably steps never have a value, as we cannot tell what continuation it will evaluate to.

Tasks can be observed to be *failing* ($\mathcal{F}$) or not. The fail task ↯ is failing, as is the paring of fail tasks ↯ ⋈ ↯, these tasks cannot be worked on. However, the task ↯ ⋈ □Int is not failing, as the second element is a task that can be worked on. TopHat's semantics ensure steps are never made to failing tasks.

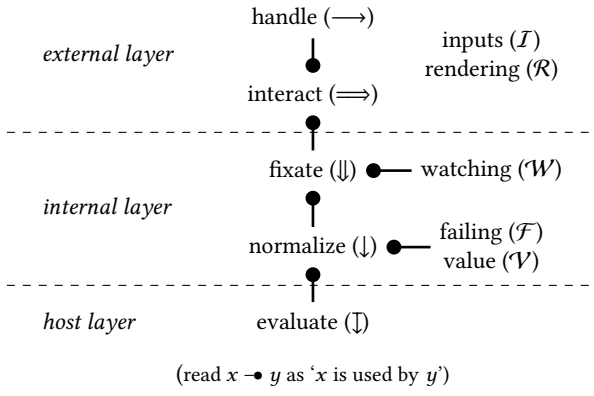


Figure 2: Overview of semantic layers, relations and functions in TopHat.

Other observations on tasks are the set of *watched addresses* ($\mathcal{W}$), and the possible *inputs* that can be send to a task ($\mathcal{I}$). Also, the *rendered* (graphical) user interface can be modelled as an observation ($\mathcal{R}$).

2.6 Semantic layers

TopHat semantics is defined in three layers: the host layer, the internal layer, and the external layer. These are depicted in Figure 2. At the bottom, there is the *host layer*, which evaluates pure lambda terms. On top of that, there are two *task layers* reasoning about the task language. The semantic arrows in the *internal layer* prepare a task for user interaction. The semantic arrows in the *external layer* do the actual handling of user inputs. At the right side of Figure 2, we can see the observations that play a role on that layer. Adding features to the task layer, should not alter the semantics of the host language and vice versa.

3 Analysing real-world TOP applications

TOP and iTask in particular have a very long history with many published case studies, examples, support applications and even some commercial use. In order to get some insight in real-world TOP combinator usage, we did a quantitative analysis of all real-world TOP applications or fragments. In this section we explain the method used to do this analysis, present our results, and for each category of combinators we describe similarities and differences between TopHat and iTask. We finalize the section with a short discussion of our findings.

3.1 Method

To gather TOP applications, we analysed little over 50 published papers on TOP, mainly about iTask but also a handful on mTask. Of those papers, we where able to recover 13 applications or example sets either from the provided artifact, by digging into the version control history of the iTask framework, or by asking the authors for a copy. Additionally, we extracted one application and an example

Table 1: List of analysed real-world applications.

name	description
conf2009	a conference management system designed to teach students iTask at the AFP 2008 school [35].
itasks22009	a set of example programs bundled with the iTask 2 system [24].
esmviz2011	a visualisation tool for testing a simplified version of iTask,written in iTask [18].
gin2012	the frontend for GiN, a graphical interactive applications to design tasks [13].
incidone2012	an incident report application for coordinating rescue situations [23].
trax2013	a single-player puzzle game designed to demonstrate the elegance en ergonomics of TOP [1].
tonic2014	the fronted for Tonic, a static and dynamic visualisation tool for iTask tasks [48].
ligretto2014	a multi-user card game designed to demonstrate the scalable vector graphics integration in iTask [2].
tasklets2015	a set of bigger examples for executing small tasks in the browser using *TaskLets* and *EditLets* [8, 9, 11].
shipadventure2017	an interactive fire-extinguishing game situated on a naval ship created to demonstrate task-oriented software development with a non-trivial case study [46].
serviceengineer2017	a case study to showcase the distributed extension of iTask. It is an application to manage and perform job allocation for service engineers [33].
taxman2018	a workflow system for entering solar panel reimbursements, designed after specifications provided by the Dutch tax office [47].
cws2023	a smart campus monitoring IOT system prototype designed after specifications given by the University of Glasgow. It is written in both iTask and mTask, the latter runs on microcontrollers [28].
admin2024	a collection of task workflows for the iTask system to administrate the server itself [37].
examples2024	a snapshot of the set of example programs bundled with the iTask system [37].
top2024	an industrial deployed GIS application developed by TOP Software B.V. of around 110 kLOC.

set from the current version of the iTask framework.[4] Finally, we included the analysis of a commercial geographic information system (GIS) application. This results in a total of 16 analysed applications, which are all listed in Table 1.

The 15 published or publicly available real-world TOP applications have a total of 30 kLOC (source lines of code). The commercial application consists of around ±110 kLOC. This demonstrates that the programs at least have quite some developer effort that was put into them [40]. We do not draw conclusions on how realistic every application is based on the number of LOC, as that metric is not necessarily a realistic measure for that purpose [3]. The commercial application and the support applications of iTask on the other hand are realistic applications by definition because they are used by many users and are constantly adapting to current needs.

In order to analyse the combinator usage, we created a program that recursively scans all modules and keeps track of the combinator usage. It does so by parsing the module with the use of the Clean compiler and looks up every identifier to see whether it is a known combinator. The list of known combinators is compiled by hand and enumerates all known task combinators and their category. The direct occurrences of the iTask super combinator `parallel` (1 in the commercial and 19 in the rest) where hand checked on a case-by-case basis to determine if the use really required to use this super combinator or could be rewritten to use simpler derived combinators. Sometimes the super combinator `parallel` is used for ergonomics and the behaviour could be expressed otherwise. The analysis was ran locally on the applications where the source code was available to us. For the commercial application we asked the company to run the tool on their source code since it is proprietary.

[4]The iTask framework is found here: https://gitlab.com/clean-and-itasks/itasks-sdk.

Table 2: Distribution of combinator categories.

category	subcategory	num	% of parallel	% of total
sequential	—	2270	—	60.76
transform	—	825	—	22.07
parallel	or	282	45.41	7.55
	and	187	30.11	5.01
	detach	67	10.79	1.79
	reflect	39	6.28	1.04
	pool	38	6.12	1.02
	rest	8	1.29	0.21
	total	621	100.00	16.62
exception	—	20	—	0.54
total	—	3736	—	100.00

3.2 Results

In total, 140 kLOC from of 395 modules was automatically analysed resulting in a total of 3736 uses of task combinators. These task combinators are divided into four categories: sequential combination (Section 3.4), transformation of tasks (Section 3.5), parallel combination (Section 3.6), and exception handling (Section 3.7). The parallel combinator is furthermore split up into six categories: disjunctive (or) and conjunctive combination (and) (Section 3.6.1), detach behaviour (Section 3.6.2), reflect behaviour (Section 3.6.3), pool behaviour (Section 3.6.4), and the rest (Section 3.6.5). Table 2 shows the results of this analysis. The following sections details on the different aspects and combinators and how they relate to TopHat's.

3.3 Representation of task values and SDSs

Before diving into the combinators, we first have to discuss the differences between task value representation in iTask and TopHat. In both systems, tasks have observable task values. However, in TopHat a task either has a value or not, in iTask a task value also has a stability. The definition for task values as used in iTask is presented in Listing 1. An unstable task value signifies that it can change in the future, a stable task value cannot but this is in no way guaranteed by the iTask system.

```
:: TaskValue a = NoValue | Value a Stability
:: Stability :== Bool
```

Listing 1: Task value data type in iTask.

Stability of tasks as used in iTask can in TopHat be mapped on the task conditions of Klijnsma and Steenvoorden [16, §5] where stable is *finished steady*, unstable is *finished unsteady* and no value is *running*. In TopHat, stability can be simulated by yielding a tuple of the real task value and a Bool representing the stability.

Furthermore, the data type for SDSs in iTask (simplified to `Shared a`) separates a read type, a write type and a parameter type [10]. Lijnse and Plasmeijer [25] introduced editors with a separate read and write type as well but comparing this to TopHat is beyond the scope of this paper. In our examples, we assume that the read type and the write type are the same parameter.

3.4 Sequential combinators

Of all combinators, the vast majority (60.76%) are sequential combinators. All sequential combinators in iTask are derived from one super combinator, `step`, which signature is shown in Listing 2.[5] This combinator has two arguments: a left-hand side, the task to execute first; and a right-hand side, a task continuation. This construction allows the right hand side to *observe* the task value of the left-hand side and act upon it. There are four different types of continuations (also listed in Listing 2):

(1) `OnValue` is used to perform the step without user intervention, just by a predicate on the task value, we call this an *internal step*;
(2) `OnAction` is only triggered when the user performs an action, e.g. click on a button, still a predicate is used to determine whether the button is enabled or not, we call this an *external step*;
(3) `OnException`, and (4) `OnAllExceptions` are used to step when a certain exception occurs (see Section 3.7).

Besides the ability to catch exceptions, there seemingly is a difference between the iTask sequential combinator and TopHat's (►). Firstly, TopHat does not explicitly support external steps, only internal ones. External steps are simulated with normal editors. Pushing a button, for example, is simulated with:

$$\square\text{Unit} \blacktriangleright \lambda\{\}.\ t_{\text{cont}}$$

[5]All types in iTask have a class collection `iTask` that contains derivable type classes for printing, storage, editors and equality. From now on we will omit these class constraints for readability of the type signatures.

```
(>>*) :: (Task a)               // Current task
         [TaskCont a (Task b)]  // Continuation list
         → Task b

:: TaskCont a b
   = OnValue ((TaskValue a) → ?b)
   | OnAction Action ((TaskValue a) → ?b)
   | ∃.e: OnException (e → b) & iTask e
   | OnAllExceptions (String → b)
```

Listing 2: Super sequential combinator in iTask.

Secondly, in iTask one can create task continuations that continue even if the left-hand side task has no value. While this is quite a pathological case because the left-hand side task only gets one normalisation step, it is possible to implement this in TopHat as well, using the fact that ■Unit always has a value:

$$(t \blacklozenge \blacksquare\text{Unit}) \blacktriangleright \lambda\_.\ t_{\text{cont}}$$

In general, it is possible to expose the no value state of tasks in TopHat, by hoisting or exposing the task value as a sum type together using the fact that ♦ is left-biased:

$$\mathit{expose}\ t = (\text{Value} \bullet t) \blacklozenge \blacksquare\text{NoValue}$$

Furthermore, the way iTask prevents or guards a step is also a bit different but related. In iTask, the *Maybe*[6] type is used whereas in TopHat, failure is an observation of the resulting task using $\mathcal{F}$, i.e. $\lightning$.

3.5 Transform combinators

Transform combinators comprise around 22.08% of the entire combinator base. These combinators allow the programmer to apply a pure function to the task value of the task. In iTask, transform combinators are all derived from the super combinator `transformError` seen in Listing 3.[7] Not only does this function allow to transform the task value, it also allows you to inject exceptions (see Section 3.7).

```
transformError ::  // | Transformation function
   ((TaskValue a) → MaybeError TaskException (TaskValue b))
   (Task a)       // Current task
   → Task b
```

Listing 3: Super transform combinator in iTask.

Other than the option to inject exceptions, the seeming difference between this super combinator and TopHat's (•) is the ability to transform no value into a value and the other way around. However, both can be expressed with TopHat as well. Using *expose* shown in Section 3.4, the existence of a task value is exposed. Then a transform function using the exposed value can be applied. The question is then how to lower the hoisted task value back into the task. Converting a NoValue to a task without a value can be done using a step. For convenience, as we will need it later, we define a new derived combinator for censor ($\lfloor t \rfloor$). Censoring a task t, means

[6]In Clean ? is the builtin strict *Maybe* type: `:: ? a = ?None | ?Just !a`
[7]The error type in Clean is defined as `:: MaybeError e a = Error e | Ok a`.

we *hide* its value for the outside world. In TopHat, we accomplish this by attempting a step to the fail task. This step is never made, because TopHat's semantics use $\lightning$ to signal an impossible step. Also, steps do not have a value, therefore censoring the value of t.

$$\lfloor t \rfloor = t \blacktriangleright \lambda\_.\ \lightning$$

Converting a hoisted task value back into a task value in general is not possible. However, it can be done in specific cases using a step (▶) in TopHat, but that freezes the value:

$$\begin{aligned} \mathit{freeze}\ t = t \blacktriangleright \lambda x.\ \textbf{case}\ x\ \textbf{of}\quad & \text{Value } x \quad \mapsto x \\ & \text{NoValue} \quad \mapsto \lightning \end{aligned}$$

3.6 Parallel combinators

16.62% of the combinators are parallel combinators and they typically appear high up in task trees. Parallel combinators perform the child tasks *at the same time*, but not necessarily truly in parallel. For example, in iTask, but also in TopHat, the user interfaces of the children are combined and presented to the user. Event handling is strictly sequential though and left biased.

All parallel combinators in iTask are implemented using the super combinator `parallel` (see Listing 4). This combinator is implemented as a function with two arguments. The first argument is an initial list of not just ordinary tasks but `ParallelTasks`. These are functions that, when given a `SharedTaskList`, produce ether a regular task (`Embedded`), or a detached task (`Detached`, see Section 3.6.2). This `SharedTaskList` can be used to observe the task values of all siblings and add, remove, or even replace tasks at will. The second argument is a list of task continuations that operate on task values of every task in the list and produce parallel tasks as continuations that are added to the task list as a new child.

Parallel tasks can use the `SharedTaskList` to manage the parallel combinator, e.g. add, remove or replace tasks. Moreover, the `SharedTaskList` is used to observe tasks. This allows for very dynamic behaviour that is similar to full control over a process table in an operating system [36].

```
:: ParallelTaskType
    = Embedded
    | Detached Bool TaskAttributes

:: ParallelTask a :== (SharedTaskList a) → Task a

parallel :: [(ParallelTaskType, ParallelTask a)]
            [TaskCont [(Int, TaskValue a)]
                      (ParallelTaskType, ParallelTask a)]
            → Task [(Int, TaskValue a)]
```

Listing 4: Super parallel combinator in iTask.

The parallel super combinator is notoriously difficult to implement and reason about [19]. Furthermore, it is quite difficult to use (see for example the implementation of `-&&-` in Listing 6) so real-world top applications almost exclusively use derived combinators that we divide into six subcategories. Most of the derived combinators are expressible in TopHat as well as we will see in the following sections.

3.6.1 and *and* or *combinators.* The simpler derived parallel combinators, disjunction (`-||-`, `-||`, and `||-`) and conjunction (`-&&-`) are used in 45.41% and 30.11% of all parallel combinators. Conjunction and disjunction are both binary combinators that run the tasks at the same time in a `parallel` and combine the task values accordingly. There are also list variants of this combinator available called `allTasks` and `anyTask`. All signatures are shown in Listing 5.

```
(-&&-)   :: (Task a) (Task b) → Task (a, b)
(-||-)   :: (Task a) (Task a) → Task a
(-|| )   :: (Task a) (Task b) → Task a
( ||-)   :: (Task a) (Task b) → Task b
allTasks :: [Task a] → Task [a]
anyTask  :: [Task a] → Task a
```

Listing 5: And and or derived combinators in iTask.

The simple *and*- and *or*-combinators have the same semantics as TopHats pair (⋈) and choose (⧫) combinators. Due to the complexity of the super combinator, deriving them is not trivial (see Listing 6). Derived combinators `allTasks` and `anyTask` can be expressed as folds over using ⋈ and ⧫ [42]. Finally, the left and right censoring variants (`-||` and `||-`) are derived using the censor combinator:

$$\begin{aligned} t_1 \texttt{-||}\ t_2 &= t_1 \ \blacklozenge\ \lfloor t_2 \rfloor \\ t_1\ \texttt{||-}\ t_2 &= \lfloor t_1 \rfloor \ \blacklozenge\ t_2 \end{aligned}$$

```
(-&&-)   :: (Task a) (Task b) → Task (a, b)
(-&&-) taska taskb = parallel
    [ (Embedded, \_ → taska @ Left)
    , (Embedded, \_ → taskb @ Right)] [] @? res
where
    res (Value [(_,Value (Left a) sa)
                ,(_,Value (Right b) sb)] _)
        = Value (a,b) (sa && sb)
    res _ = NoValue
```

Listing 6: Implementation of the and derived combinator in iTask.

3.6.2 detach *combinators.* The third biggest category of combinators is the *detach* category, spanning 10.79% of the parallel combinator category. These combinators are implemented using the `Detached` type of parallel task to graft a task tree onto another task tree. The most popular combinator is the assign combinator (`@:`, Listing 7). This *assigns* a task to a specific user but allows the original task to observe the progress: a loose coupling. So the task is *detached* and the user can see this incoming task and choose to *attach* to the task to work on it.

```
(@:) :: User (Task a) → Task a
attach :: InstanceNo Bool → Task AttachmentStatus
```

Listing 7: Assign core combinator in iTask.

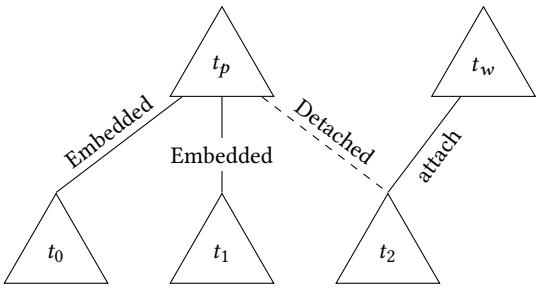


Figure 3: Visualisation of detach behaviour.

Figure 3 illustrates a parallel with a detach task. Task t_p is a parallel task, arbitrarily deep in a task tree with several embedded tasks and one detached task. Detaching the task means separating it from the task tree but it maintains a loose link that allows progress to be observed. The task is part of t_p, but its user interface is not shown and it cannot be worked on by t_p. In the figure, t_w attaches to the task to work on it.

In the iTask implementation, the parallel combinator uses an SDS to store the list of tasks. SDS's are persistent over execution and data is serialised using the *GraphCopy* library for serialising execution graphs, i.e. Graph Packing [6, 33]. Serialising tasks, i.e. execution graphs, works well but is not stable across different versions of the code or even compilations. The commercial application (**top2024**) therefore never uses detach behaviour directly but always simulates it using data. Instead of detaching a task for someone to attach to, a label is written into an SDS. The other user then knows which task to start from the label and explicitly communicates its value via a dedicated SDS.

Detaching parts of task trees and grafting it to other task trees is currently not implementable in TopHat. Finding new core combinators to implement this remains future work. Its limited form, as used by the commercial application, can be simulated similarly using labels.

3.6.3 reflect *combinators.* Then, there is a category of parallel combinators called the *reflect* combinators, with an amount of 6.28% of the total parallel combinators. These combinators utilise the `parallel` super combinator to monitor other task's progress *across* control flow. There are three reflection combinators that are often used shown in Listing 8. All expose the task value in SDSs to provide to a task function.

Feed-forward (`>&>`) exposes the left-hand side task's value in an SDS that is passed to the right-hand side task function. The task value of the right-hand side is considered the task value of the compound task. Feed-sideways does the same but considers the task value of the left-hand side as the task value of the compound task. Finally there is the feed-bidirectional combinator. This ties a knot between two tasks by exposing the one observable task value to the other and vice versa. The task value of the combined task is the pair of the individual tasks, similar to the conjunctive combinator (`-&&-` in iTask, ⋈ in TopHat). In Section 4 we provide a new TopHat core combinator to implement these combinators.

3.6.4 pool *combinators.* Pool combinators are combinators that manage a list of tasks, i.e. shrinking and growing the list by actions

```
(>&>) :: (Task a) (Shared (TaskValue a)) → Task b) → Task b
(>&^) :: (Task a) (Shared (TaskValue a)) → Task b) → Task a
(<&>) :: ((Shared (TaskValue b)) → Task a)
         ((Shared (TaskValue a)) → Task b)
            → Task (a, b)
```

Listing 8: Derived reflection combinators in iTask.

of the user. They comprise around 6.12% of all parallel combinators. The most used combinator in the pool category is the sideStep (`>^*`) combinator (Listing 9). This combinator has a similar type signature to the regular step (Listing 2) only with a different return type: `Task a` instead of `Task b`. Instead of continuing with one of the continuations, the picked continuation task is added as a child to the parallel.

```
(>^*) :: (Task a) [TaskCont a (Task b)] → Task a
```

Listing 9: Sidestep derived combinator in iTask.

When the `TaskConts` are just actions, i.e. buttons, this is implementable using the pool combinator from TopHat (⋈). Otherwise it is considered to be *rest* behaviour.

3.6.5 rest *combinators.* Then there is the *rest* category, only spanning 1.29% of parallel combinator use. This type of behaviour uses the `parallel` combinator to dynamically shrink and grow the task list without user intervention. Implementing this in TopHat is possible but many nice properties such as symbolic execution [43] will be lost.

3.7 Exception combinators

Finally, the last 0.54% of the main categories of combinators concerns exception handling. In iTask, exceptions are caught using either sequential or parallel combinators, but it is also possible to catch them using the derived combinators `try` and `catchAll`.

Exceptions are also (ab)used as a way of short circuiting execution. For example, when the left-hand side of a parallel *or* combinator throws an exception, the right-hand side of the combinator is not executed any more, the exception prevented that. This is never used directly and only found in one derived combinator, `whileUnchanged` (Listing 10), so it can be considered an implementation detail. This function has two arguments, the first argument is an SDS, the second a task function that is restarted each time the SDS changes. Internally it is implemented with an *or* combinator that is short circuited.

```
whileUnchanged :: (Shared a)    // SDS to observe
                  (a → Task b) // Task function
                  → Task b
```

Listing 10: The type of iTask's `whileUnchanged` function.

Typical use cases of exceptions are easily implementable in TopHat by encoding them as a sum type such as `MaybeError`. It remains future work to see how the short circuiting behaviour can also be implemented.

3.8 Discussion

The analysis of the large code base showed that TopHat covers the majority of real-world workflows. All real-world sequential and transform combinators can be expressed in equivalent TopHat. This already comprises 82.84% of all combinators. Of the remaining 17.16%, 2.54% is not yet expressible, containing some variants of the parallel combinator and exceptions. For the *reflect* behaviour, we have identified a core combinator that allows us to express this. The implementation and semantics are shown in the next section. We leave the *detach* and *exception* behaviour for future work.

Further work could be to split out the combinators even more. For example analysing task continuation usage as well. Alternatively, combinator distribution can be propagated, there are many derived combinators that use multiple core combinators, this could be propagated to get a more detailed view of core combinator statistics. Finally, the SDSs in iTask are richer and support combinators as well, it would be interesting to see how they compare to TopHat.

4 Reflection on task values

During the quantitative analysis of the combinator usage in real-world Top applications, we identified a gap. All different *reflect* behaviours, comprising 6.28% of the parallel combinator usage, and possibly some of the *rest* behaviour, can be expressed with a single new core combinator: *reflect*. In this section, we describe the observed *reflect* combinators with examples from the real-world applications, introduce a reflect core combinator (◎) with its semantics in TopHat, and finally show how to derive the real-world combinators from it.

4.1 Reflect derived combinators

There are two combinators that comprise almost all of the reflection behaviour seen in real-world applications. Those are feed-sideways (`>&^`) and feed-forward (`>&>`), which types we have seen in Listing 8. They expose task values into an SDS so that other tasks can observe the progress *across* control flow.

In the analysed examples we see higher-level workflows that depend on this behaviour, for example in the helper function `crud` used in **taxman2018** and **shipadventure2017**. Listing 11 shows this function. It creates a task with which one can execute create/read/update/delete (CRUD) operations on an SDS. The left-hand side of the combinator is a task for selecting the key of the entry you want to edit. The right-hand side of the combinator is an expression that creates a task from the currently selected key. All operation on the map are guarded except for *Create* for it needs no key. When the result of the database operations are not needed for the compound task, `feedSideways` is used. Changing it to `feedForward` changes the task value to the result of the operation once it is finished. The generated resulting application is shown in Figure 4.

Furthermore, `feedForward` is useful as a debugging tool when doing rapid prototyping, an often described use case of iTask [20]. Such usage has for example been seen in **tasklets2015** codebase. In this example, `feedForward` is combined with the `withSelection` function. Listing 12 shows the type of this function and an example of its usage. It results in the user interface shown in Figure 5.

Recently, a new derived reflection combinator, feed-bidirectionally (`<&>`), has been added to the iTask system. This novel combinator

```
crud :: (Shared (Map k v)) → Task k
crud sds = (enterChoiceWithShared [ChooseFromList id]
        (mapRead keys sds)
        <<@ Title "Entries in database"
    >&^ \mk→forever $ viewSharedInformation [] mk
        <<@ Title "Current selection"
    >>* [ OnAction (Action "Create") (always  createTask)
        , OnAction (Action "Read")   (hasJust readTask)
        , OnAction (Action "Update") (hasJust updateTask)
        , OnAction (Action "Delete") (hasJust deleteTask)
        ])
where
    hasJust f = ifValue isJust (f o fromJust)
    createTask = enterInformation []
        <<@ Label "Enter the key and value"
        >>? \(k, v)→upd (put k v) sds
    readTask k = ···
    ···
```

Listing 11: A simple CRUD interface to an SDS

```
withSelection :: (Task c)  // When there is no value yet
    (a → Task b)           // When there is a value
    (Shared (TaskValue a)) // The SDS reflecting the value
    → Task b

withSel :: Task Int
withSel = enterInformation [] <<@ Label "Enter a number"
    >&> withSelection (viewInformation [] "Nothing entered")
        (\v→viewInformation [] v <<@ Label "You entered")
```

Listing 12: The type of `withSelection` and an example of debugging a task value.

ties a knot between two tasks, letting them observe each other's result. This is typically used high up the tree and is due to its novelty only observed in **top2024**. The task value in that case often represents some kind of progress or some kind of status.

4.2 Reflection in TopHat

We observed that these three combinators, feed-forward, feed-sideways, and feed-bidirectionally, in iTask all *reflect* the value of a task into an SDS. That is, other tasks can see and react upon the value of the reflected task by watching this SDS or address. To accommodate this, we extend TopHat with a new core combinator, reflect:

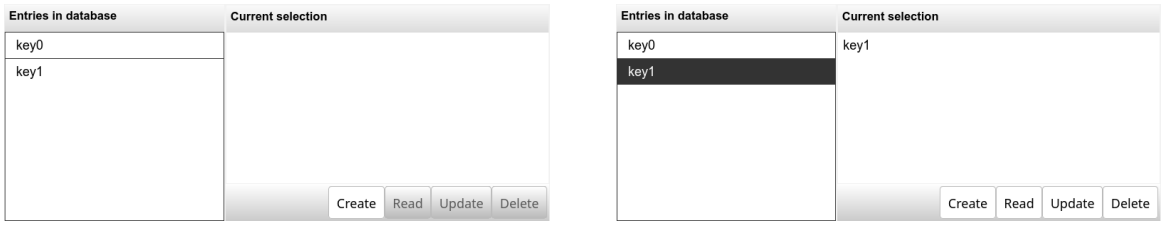
$$t ::= \ldots \mid ◎_a\ t$$

where the value of task t is reflected at address a.

The typing rule for the reflect combinator is as follows:

$$\textsc{T-Reflect}\quad \frac{\Gamma, \Sigma \vdash a : \text{Ref (Maybe } \tau) \qquad \Gamma, \Sigma \vdash t : \text{Task } \tau}{\Gamma, \Sigma \vdash\ ◎_a\ t : \text{Task } \tau}$$

Here, Ref τ is the type of references to τ, and Task τ is a task containing a value of type τ. As tasks can or cannot have a value,



(a) Without a selection, only the Create button is enabled.

(b) When selecting a key, the Read, Write and Update buttons are enabled as well.

Figure 4: The interface for the CRUD task.

(a) When there is no task value yet, it shows only the label *Nothing entered*.

(b) When there is a task value, it shows the label *You entered* and the current task value.

Figure 5: The user interface of Listing 12.

we need to take this into account and use the standard Maybe type to wrap the reflected value in the reference.

Next, we discuss the implementation of reflection in TopHat's semantics. Of the semantic arrows shown in Section 2, only the normalisation ($\downarrow$) and handling ($\longrightarrow$) semantics need to be extended. However, all observations need to be extended to support reflection on tasks.

4.2.1 Normalisation and handling. As stated in Section 2.6, normalisation is the process of preparing tasks for user input. They form a subset of tasks t. Normalising $\circledcirc_a\ t$ in state σ comprises:

(1) normalising t in σ to normalised task n', delivering a new state σ' and some dirty addresses δ';
(2) adding a to the dirty addresses, as we modified it;
(3) setting address a to the value of n', which is precisely the reflection behaviour we like to model;
(4) returning a new reflect combinator on address a, but now on the normalised task n'.

The final derivation rule for normalisation reads:

N-Reflect

$$\frac{t, \sigma \downarrow n', \sigma', \delta'}{\circledcirc_a t, \sigma \downarrow \circledcirc_a\ n', [a \mapsto v]\sigma', \delta' \cup \{a\}}\ v = \begin{cases} \text{Just } \mathcal{V}(n', \sigma') & \text{if defined} \\ \text{Nothing} & \text{otherwise} \end{cases}$$

Here we use constructors Just and Nothing to take into account that normalised task n' can or cannot have a value.

To handle inputs ι by reflections we simply pass the input on to the inner, now normalised, task n. The result is now a not-normalised task t'.

H-Reflect

$$\frac{n, \sigma \xrightarrow{\iota} t', \sigma', \delta'}{\circledcirc_a\ n, \sigma \xrightarrow{\iota} \circledcirc_a\ t', \sigma', \delta'}$$

4.2.2 Observations. Observations on reflections are mostly delegating to the same observation on the inner task. Only $\mathcal{W}$ also adds address a to the watch set. Note that all observations, except $\mathcal{F}$, actually operate on a normalised task n.

$$\begin{aligned} \mathcal{F}(\circledcirc_a\ t) &= \mathcal{F}(t) \\ \mathcal{V}(\circledcirc_a\ n, \sigma) &= \mathcal{V}(n, \sigma) \\ \mathcal{I}(\circledcirc_a\ n) &= \mathcal{I}(n) \\ \mathcal{W}(\circledcirc_a\ n) &= \mathcal{W}(n) \cup \{a\} \\ \mathcal{R}(\circledcirc_a\ n, \sigma) &= \mathcal{R}(n, \sigma) \end{aligned}$$

As the value of n is reflected at address a, one could think that the value at address a is *always* the same as the value of the reflected, normalised task n:

$$\mathcal{V}(\circledcirc_a\ n, \sigma) = \mathcal{V}(n, \sigma) = \sigma(a) \quad \text{Not true!}$$

Here $\sigma(a)$ looks up the value stored at address a in state σ. Sadly, this is not true. Take for example the following task:

$$\textbf{share}\ \text{Nothing} \blacktriangleright \lambda a.\ (\circledcirc_a\ \blacksquare 37) \bowtie (a := 42)$$

This should reflect the value 37 in a but *also* sets a to 42. Because the semantics of the pairing combinator ($\bowtie$) is defined to be *left-to-right*, normalising $a := 42$ is done *after* normalising $\circledcirc_a\ \blacksquare 37$. Therefore, the value stored in a is 42 and not 37 and the value of the left task is not reflected in the address. This could be solved by introducing read-only memory locations in the host language.

4.3 Deriving the feed combinators

With the semantics of reflection in place, we can now define the feed-forward, -sideways, and -bidirectionally combinators. For this, we need one helper on tasks, *censor* ($\lfloor t \rfloor$), which we already discussed

in Section 3.5 and repeat its definition here for convenience:

$$\lfloor t \rfloor = t \blacktriangleright \lambda\_.\ \lightning$$

Now we can define feed-forward ($\oslash$), -sideways ($\ominus$), and -bidirectionally ($\oplus$) using censor as follows:

$$\begin{aligned} t \oslash e &= \textbf{share}\ \text{Nothing} \blacktriangleright \lambda a.\ \lfloor \circledcirc_h t \rfloor \blacklozenge e\ a \\ t \ominus e &= \textbf{share}\ \text{Nothing} \blacktriangleright \lambda a.\ \circledcirc_h\ t \blacklozenge \lfloor e\ a \rfloor \\ e_1 \oplus e_2 &= (\textbf{share}\ \text{Nothing} \bowtie \textbf{share}\ \text{Nothing}) \\ &\quad \blacktriangleright \lambda\{a_1, a_2\}.\ \circledcirc_{a_1}\ (e_1\ a_2) \bowtie \circledcirc_{a_2} (e_2\ a_1) \end{aligned}$$

To feed-forward $t \oslash e$ or feed-sideways $t \ominus e$, we start by sharing the value Nothing at address a, this address is then used by the reflect combinator to reflect the value of task t. However, in case of feed-forward, we are not interested in the result of this task, but in the result of the task computed by $e\ a$. Therefore, we censor the left-hand side of the choice operator. This results in the right-hand side at the moment it produces a value. In case of feed-sideways, this is exactly the other way around, we censor the right-hand side to result only the left-hand side reflected task.

In case of feed-bidirectionally $e_1 \oplus e_2$, we start sharing at two addresses, a_1 and a_2. We reflect the value of e_1 in a_1, and of e_2 in a_2. Note we pass the addresses reflecting *the other* task to each expressions.

5 Related work

In the quantitative and qualitative analysis of this paper, around 50 TOP related papers and case studies where extracted. We are quite confident that, barring at most one or two papers, this is the entire body of research on TOP. However, there is some other related work.

Functional reactive programming (FRP) is a programming style that at first glance looks similar to TOP. The term was first coined by Elliott and Hudak [12] as a technique to declaratively specify animations. Since then, many implementations can be found in other domains [5], including workflow modelling [39]

Where TOP focusses on collaboration, FRP is focussed on data dependencies and behaviours of them. Similar to TOP, vanilla FRP can give no guarantees on memory usage, but extensions such as arrowised FRP [32] or modal FRP [4].

Workflow modelling in programming languages is a well-trodden field. With TOP, interactive collaborative systems are programmed using a high abstraction level making it very suitable to model workflows. van der Aalst et al. [49] and Russell et al. [41] provide an overview of languages and frameworks and defines a benchmark.

Constraint-based approaches to model business processes, such as Declare [29] and Ampersand [15] follow a different approach to TOP. Their semantics can be characterized with logic-based formalisms like relation algebras. Using these formalisms, one describes rules to keep information in the system consistent. Instead of specifying a workflow, it is automatically derived from this specification. Alternatively, timed Dynamic Condition Response (DCR) graphs [14] are a way of specifying workflows which allow formal verification of both safety and liveness properties.

6 Conclusions

We analysed sixteen real-world TOP applications. Thirteen came from literature, two are internal to iTASK and one is a commercial GIS application. Together, they total 140 kLOC defined in a total of 395 modules. From the combinator usage, several gaps are identified between TOP and iTASK.

We find that over 96.50% of combinator usage of real-world TOP programs can be expressed in TOPHAT. However, as most of these examples make use of recursion, they can only be expressed in an extended version of TOPHAT without termination assurance. In practice, this means the examples can be executed, but TOPHAT's symbolic execution engine is not guaranteed to terminate.

We observed that parallel combinators are used way less than sequential combinators (±16% versus ±60%). Furthermore, we see that parallel combinators typically appear higher up in the task tree, giving some indication that they are used to describe workflow higher up, i.e. are more important. With the new reflect combinator, we capture an extra 6% of all parallel combinator uses in real-world application.

6.1 Future work

Detaching tasks, i.e. separating tasks from their task tree and allowing other task trees to take over the task, is something that is available in iTASK but not in TOPHAT. It would be interesting to see if and what core combinator we would need in order to express this behaviour as well.

SDSs in iTASK are much richer and support combinators as well, it would be interesting to see how they compare to TOPHAT.

Finally, in iTASK, there is an exception mechanism available. For example, a task may throw an exception, this exception bubbles up and is caught by a sequential combinator. All real-world usage of exceptions can easily be simulated by using an `Either` or `Error` type but one. The exception mechanism is used to short-circuit derived combinators such as `whileUnchanged`. Figuring out how to shortcircuit combinators in TOPHAT is future work.

References

[1] Peter Achten. 2013. Why Functional Programming Matters to Me. In *The Beauty of Functional Code*, Peter Achten and Pieter Koopman (Eds.). Vol. 8106. Springer Berlin Heidelberg, Berlin, Heidelberg, 79–96. doi:10.1007/978-3-642-40355-2_7

[2] Peter Achten, Jurriën Stutterheim, László Domoszlai, and Rinus Plasmeijer. 2014. Task oriented programming with purely compositional interactive scalable vector graphics. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, New York, NY, USA, 7.

[3] Kalev Alpernas, Yotam M. Y. Feldman, and Hila Peleg. 2020. The wonderful wizard of LoC: paying attention to the man behind the curtain of lines-of-code metrics. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 146–156. doi:10.1145/3426428.3426921 event-place: Virtual, USA.

[4] Patrick Bahr. 2022. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming* 32 (2022), e15. doi:10.1017/S0956796822000132

[5] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4 (Aug. 2013), 1–34. doi:10.1145/2501654.2501666 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[6] Jost Berthold. 2011. Orthogonal Serialisation for Haskell. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–53.

[7] Tom H. Brus, Marko C.J.D. van Eekelen, Maarten O. van Leer, and Marinus J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional*

Programming Languages and Computer Architecture, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

[8] László Domoszlai and Tamás Kozsik. 2013. Clean Up the Web! In *The Beauty of Functional Code: Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*, Peter Achten and Pieter Koopman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 133–150. doi:10.1007/978-3-642-40355-2_10

[9] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Editlets: type-based, client-side editors for iTasks. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/2746325.2746331 event-place: Boston, MA, USA.

[10] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. ACM, New York, NY, USA, 1–11. doi:10.1145/2746325.2746333 event-place: Boston, MA, USA.

[11] László Domoszlai and Rinus Plasmeijer. 2015. Tasklets: Client-Side Evaluation for iTask3. In *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, Viktória Zsók, Zoltán Horváth, and Lehel Csató (Eds.). Springer International Publishing, Cham, 428–445. doi:10.1007/978-3-319-15940-9_11

[12] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, New York, NY, USA, 263–273.

[13] Jeroen Henrix, Rinus Plasmeijer, and Peter Achten. 2012. GiN: A Graphical Language and Tool for Defining iTask Workflows. In *Trends in Functional Programming*, Ricardo Peña and Rex Page (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–178.

[14] Thomas T. Hildebrandt, Håkon Normann, Morten Marquard, Søren Debois, and Tijs Slaats. 2021. Decision Modelling in Timed Dynamic Condition Response Graphs with Data. In *Business Process Management Workshops - BPM 2021 International Workshops, Rome, Italy, September 6-10, 2021, Revised Selected Papers (Lecture Notes in Business Information Processing, Vol. 436)*, Andrea Marrella and Barbara Weber (Eds.). Springer, 362–374. doi:10.1007/978-3-030-94343-1_28

[15] Stef Joosten. 2018. Relation Algebra as programming language using the Ampersand compiler. *J. Log. Algebraic Methods Program.* 100 (2018), 113–129. doi:10.1016/J.JLAMP.2018.04.002

[16] Tosca Klijnsma and Tim Steenvoorden. 2022. Semantic Equivalence of Task-Oriented Programs in TopHat. In *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13401)*, Wouter Swierstra and Nicolas Wu (Eds.). Springer, Cham, 100–125. doi:10.1007/978-3-031-21314-4_6

[17] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. ACM Press, Vienna, Austria, 1–11. doi:10.1145/3183895.3183902

[18] Pieter Koopman, Rinus Plasmeijer, and Peter Achten. 2011. An Executable and Testable Semantics for iTasks. In *Implementation and Application of Functional Languages*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 212–232.

[19] Bas Lijnse. 2013. Evolution of a Parallel Task Combinator. In *The Beauty of Functional Code: Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*, Peter Achten and Pieter Koopman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–210. doi:10.1007/978-3-642-40355-2_14

[20] Bas Lijnse. 2013. *TOP to the rescue: task-oriented programming for incident response applications*. UB Nijmegen, Nijmegen. OCLC: 833851220.

[21] Bas Lijnse. 2022. Modeling Real World Crisis Management Plans with C2Sketch.. In *ISCRAM 2022 Conference Proceedings – 19th International Conference on Information Systems for Crisis Response and Management*. ISCRAM, Brussels, Belgium, 404–413. event-place: Tarbes, France.

[22] Bas Lijnse. 2024. Toppyt. https://gitlab.com/baslijnse/toppyt

[23] Bas Lijnse, Jan Martin Jansen, Rinus Plasmeijer, and others. 2012. Incidone: A task-oriented incident coordination tool. In *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM*, Vol. 12. ISCRAM, Brussels, Belgium, 1–5.

[24] Bas Lijnse and Rinus Plasmeijer. 2009. iTasks 2: iTasks for End-users. In *International Symposium on Implementation and Application of Functional Languages*. Springer, Cham, 36–54.

[25] Bas Lijnse and Rinus Plasmeijer. 2021. Typed Directional Composable Editors in iTasks. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20)*. Association for Computing Machinery, New York, NY, USA, 115–126. doi:10.1145/3462172.3462197 event-place: Canterbury, United Kingdom.

[26] Mart Lubbers. 2023. *Orchestrating the Internet of Things with Task-Oriented Programming*. PhD Thesis. Radboud University Press, Nijmegen. doi:10.54195/9789493296114.

[27] Mart Lubbers and Peter Achten. 2024. Clean for Haskell Programmers. https://arxiv.org/abs/2411.00037 _eprint: 2411.00037.

[28] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2023. Could Tierless Languages Reduce IoT Development Grief? *ACM Trans. Internet Things* 4, 1 (Feb. 2023), 1–35. doi:10.1145/3572901 Place: New York, NY, USA Publisher: ACM.

[29] Marco Montali, Federico Chesani, Paola Mello, and Fabrizio Maria Maggi. 2013. Towards data-aware constraints in declare. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, Sung Y. Shin and José Carlos Maldonado (Eds.). ACM, 1391–1396. doi:10.1145/2480362.2480624

[30] Nico Naus and Tim Steenvoorden. 2020. Generating Next Step Hints for Task Oriented Programs Using Symbolic Execution. In *Trends in Functional Programming - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12222)*, Aleksander Byrski and John Hughes (Eds.). Springer, Cham, 47–68. doi:10.1007/978-3-030-57761-2_3

[31] Nico Naus, Tim Steenvoorden, and Markus Klinik. 2019. A symbolic execution semantics for TopHat. In *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*, Jurriën Stutterheim and Wei-Ngan Chin (Eds.). ACM, New York, USA, 1–11. doi:10.1145/3412932.3412933

[32] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '02*. ACM Press, Pittsburgh, Pennsylvania, 51–64. doi:10.1145/581690.581695

[33] Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages*. ACM, New York, NY, USA, 7. event-place: Bristol, UK.

[34] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. *ACM SIGPLAN Notices* 42, 9 (2007), 141–152.

[35] Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, and Thomas van Noort. 2009. An iTask Case Study: A Conference Management System. In *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–329. doi:10.1007/978-3-642-04652-0_7

[36] Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, Thomas van Noort, and John van Groningen. 2011. iTasks for a Change: Type-safe Run-time Change in Dynamically Evolving Workflows. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*. ACM, New York, NY, USA, 151–160. doi:10.1145/1929501.1929528 event-place: Austin, Texas, USA.

[37] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP '12)*. ACM, New York, NY, USA, 195–206. doi:10.1145/2370776.2370801 event-place: Leuven, Belgium.

[38] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2021. *Clean Language Report version 3.1*. Technical Report. Institute for Computing and Information Sciences, Nijmegen. 127 pages.

[39] Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 55–68. doi:10.1145/2661136.2661140 event-place: Portland, Oregon, USA.

[40] Jarrett Rosenberg. 1997. Some misconceptions about lines of code. In *Proceedings Fourth International Software Metrics Symposium*. IEEE, New York, NY, USA, 137–142. doi:10.1109/METRIC.1997.637174 event-place: Albuquerque, NM, USA.

[41] Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and Nataliya Mulyar. 2006. *Workflow Control-Flow Patterns: A Revised View*. Technical Report BPM-06-22. BPM Center. http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf

[42] Tim Steenvoorden. 2022. *TopHat: Task-oriented programming with style*. Ph. D. Dissertation. Radboud University, Nijmegen, the Netherlands.

[43] Tim Steenvoorden and Nico Naus. 2024. Dynamic TopHat: Start and Stop Tasks at Runtime. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages (IFL '23)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3652561.3652574 event-place: Braga, Portugal.

[44] Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A Formal Foundation for Task-Oriented Programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP '19)*. ACM, New York, NY, USA, 1–13. doi:10.1145/3354166.3354182 event-place: Porto, Portugal.

[45] Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A formal foundation for task-oriented programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, New York,

USA, 17:1–17:13. doi:10.1145/3354166.3354182

[46] Jurriën Stutterheim. 2017. *A Cocktail of Tools: Domain-Specific Languages for Task-Oriented Software Development*. UB Nijmegen, Nijmegen. https://hdl.handle.net/2066/181589

[47] Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2018. Maintaining Separation of Concerns Through Task Oriented Software Development. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Vol. 10788. Springer International Publishing, Cham, 19–38. doi:10.1007/978-3-319-89719-6

[48] Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. 2014. Tonic: An infrastructure to graphically represent the definition and behaviour of tasks. In *International Symposium on Trends in Functional Programming*. Springer, Cham, 122–141.

[49] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, Bartosz. Kiepuszewski, and Alistair P. Barros. 2003. Workflow Patterns. *Distributed and Parallel Databases* 14, 1 (July 2003), 5–51. doi:10.1023/A:1022883727209