Dynamic TopHat: Start and Stop Tasks at Runtime (draft)

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

Nico Naus
nico.naus@ou.nl
Open University of The Netherlands
Heerlen, The Netherlands

ABSTRACT

TopHat is a mathematically formalised language for Task-Oriented Programming (top). It allows developers to specify workflows and business processes in a formal language, reason about their equality and use symbolic execution to verify their correctness. TopHat can run a specification to support collaborators during the execution of a workflow. However, it can only do so for a statically specified amount of work. That is, the number of tasks running in parallel is always predefined by the developer. In contrast, other top engines like iTasks and mTasks act like an operating system, starting and stopping tasks at will.

To capture this dynamic nature of workflow systems, we introduce Dynamic TopHat: a moderate extension to the TopHat calculus which allows end users to initialise and kill tasks at runtime. Although this is a restricted version of the dynamic tasks lists found in iTasks, where the system itself can initialise new tasks, we show that all common use cases of this feature are still expressible in Dynamic TopHat. Also, our proposed solution does not compromise the formal reasoning properties of TopHat. TopHat's metatheory is formalised in the dependently typed programming language Idris and it's symbolic execution engine is implemented in Haskell.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Software prototyping**; **Software verification**.

KEYWORDS

Task oriented programming, formal semantics, symbolic execution, functional programming

ACM Reference Format:
Tim Steenvoorden and Nico Naus. 2023. Dynamic TopHat: Start and Stop Tasks at Runtime (draft). In *IFL '23: The 35th Symposium on Implementation and Application of Functional Languages, August 29–31, 2023, Braga, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 TOPHAT

1.1 Language

Here we describe the TopHat language as presented in [Steenvoorden 2022]. This slightly deviates from other presentations in [Naus and Steenvoorden 2020; Naus et al. 2019; Steenvoorden et al. 2019]. The TopHat language consists of two layers: the expression layer and the task layer. The expression layer is a simply typed λ-calculus with primitive types, pairs, lists and references. We use pattern matching to project the fields of pairs. The grammar of the task layer is given in Fig. 1. Tasks can be editors or combinators.

d ::=			Editors
	|	$\square^v \beta$	– unvalued editor
	|	$\boxminus^v b$	– valued editor
	|	$\boxcircle^v b$	– valued read-only editor
	|	$\boxplus^v h$	– shared editor
	|	$\boxoplus^v h$	– shared read-only editor
t ::=			Tasks
	|	d	– editor
	|	$e_1 \bullet t_2$	– transform
	|	$t_1 \bowtie t_2$	– pair
	|	$\blacksquare v$	– lift
	|	$t_1 \blacklozenge t_2$	– choose
	|	$\lightning$	– fail
	|	$t_1 \blacktriangleright e_2$	– step
	|	$\textbf{share}\ v$	– share
	|	$h_1 := v_2$	– assign

Figure 1: Task grammar

Editors are the interaction points of a task program. They can be thought of as an abstraction over widgets or form input fields. Editors are type safe, they only accept values of their specified type. The type of an editor can also be used to (generically) render a user interface. For example, an editor of type Bool could be rendered as a check box, while end users could interact with an editor of type Location by putting a pin on a map.

Combinators unite smaller tasks into bigger ones. The pair combinator ($t_1 \bowtie t_2$) for example, pairs two tasks so that both can be worked on concurrently. The result thereof is a pair containing both the results of t_1 and t_2. The choose combinator ($t_1 \blacklozenge t_2$) on the other hand, results in either the result of t_1 or t_2, depending on which task first has a result. The most important combinator is the step combinator. The task $t_1 \blacktriangleright e_2$ decides when to transition from task t_1 to the task calculated by its continuation e_2.

Example 1.1 (Vending machine). The following example demonstrates the use of external communication and choice. We have a vending machine that dispenses a biscuit for one coin and a chocolate bar for two coins.

$$
\begin{array}{l}
\textbf{let } \text{vend} : \textsc{Task Snack} = \square\textsc{Int} \triangleright \lambda n. \\
\quad \textbf{case } n \textbf{ of} \\
\qquad 1 \mapsto \boxcircle\text{Biscuit} \\
\qquad 2 \mapsto \boxcircle\text{Chocolate} \\
\qquad \_ \mapsto \lightning
\end{array}
$$

The editor $\square\textsc{Int}$ 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 on the machine, which sends a continue event to the select combinator ($\triangleright$). The button is initially disabled, due to the fact that the editor 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. Other cases result in the failure task $\lightning$, which signals a path that cannot be followed. The vending machine stays in the state to accept and return money. Snacks are modelled using a custom type.

Fig. 2 shows the grammar of types. The typing rules of all tasks can be found in Fig. 3. Note that editors and references can only contain values of basic types β. In particular, editors and references cannot contain functions or tasks themselves. The reason for this restriction is twofold. First, prohibiting references to functions keeps our language total. This is necessary for the symbolic execution to not enter an infinite loop. Second, restricting editors to only contain basic values guarantees one can present these values to end users (print them) and receive new inputs from end users (parse them). Arbitrary tasks and functions do not have this property.

$$
\begin{array}{rcll}
\tau & ::= & & \text{Types} \\
 & | & \tau_1 \longrightarrow \tau_2 & \text{– function type} \\
 & | & \textsc{Ref } \tau & \text{– reference type} \\
 & | & \textsc{Task } \tau & \text{– task type} \\
 & | & \textsc{Unit} & \text{– unit type} \\
 & | & \tau_1 \times \tau_2 & \text{– product type} \\
 & | & \textsc{List } \tau & \text{– list type} \\
 & | & \pi & \text{– primitive type} \\
\pi & ::= & & \text{Primitive types} \\
 & | & \textsc{Bool} & \text{– boolean type} \\
 & | & \textsc{Int} & \text{– integer type} \\
 & | & \textsc{String} & \text{– string type} \\
\beta & ::= & & \text{Basic types} \\
 & | & \textsc{Unit} & \text{– unit type} \\
 & | & \beta_1 \times \beta_2 & \text{– product type} \\
 & | & \textsc{List } \beta & \text{– list type} \\
 & | & \pi & \text{– primitive type}
\end{array}
$$

Figure 2: Typing grammar

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

$$
\textsc{T-Enter}\ \frac{}{\Gamma, \Sigma \vdash \square^{\nu}\beta : \textsc{Task } \beta}
\qquad
\textsc{T-Update}\ \frac{\Gamma, \Sigma \vdash e : \beta}{\Gamma, \Sigma \vdash \boxminus^{\nu} e : \textsc{Task } \beta}
\qquad
\textsc{T-View}\ \frac{\Gamma, \Sigma \vdash e : \beta}{\Gamma, \Sigma \vdash \boxcircle^{\nu} e : \textsc{Task } \beta}
$$

$$
\textsc{T-Change}\ \frac{\Gamma, \Sigma \vdash e : \textsc{Ref } \beta}{\Gamma, \Sigma \vdash \boxplus^{\nu} e : \textsc{Task } \beta}
\qquad
\textsc{T-Watch}\ \frac{\Gamma, \Sigma \vdash e : \textsc{Ref } \beta}{\Gamma, \Sigma \vdash \boxed{\oplus}^{\nu} e : \textsc{Task } \beta}
$$

$$
\textsc{T-Trans}\ \frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \longrightarrow \tau_2 \qquad \Gamma, \Sigma \vdash e_2 : \textsc{Task } \tau_1}{\Gamma, \Sigma \vdash e_1 \bullet e_2 : \textsc{Task } \tau_2}
$$

$$
\textsc{T-Pair}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task } \tau_1 \qquad \Gamma, \Sigma \vdash e_2 : \textsc{Task } \tau_2}{\Gamma, \Sigma \vdash e_1 \bowtie e_2 : \textsc{Task } (\tau_1 \times \tau_2)}
\qquad
\textsc{T-Lift}\ \frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \blacksquare e : \textsc{Task } \tau}
$$

$$
\textsc{T-Choose}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task } \tau \qquad \Gamma, \Sigma \vdash e_2 : \textsc{Task } \tau}{\Gamma, \Sigma \vdash e_1 \blacklozenge e_2 : \textsc{Task } \tau}
\qquad
\textsc{T-Fail}\ \frac{}{\Gamma, \Sigma \vdash \lightning : \textsc{Task } \tau}
$$

$$
\textsc{T-Step}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task } \tau_1 \qquad \Gamma, \Sigma \vdash e_2 : \tau_1 \longrightarrow \textsc{Task } \tau_2}{\Gamma, \Sigma \vdash e_1 \blacktriangleright e_2 : \textsc{Task } \tau_2}
$$

$$
\textsc{T-Share}\ \frac{\Gamma, \Sigma \vdash e : \beta}{\Gamma, \Sigma \vdash \textbf{share } e : \textsc{Task } (\textsc{Ref } \beta)}
\qquad
\textsc{T-Assign}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Ref } \beta \qquad \Gamma, \Sigma \vdash e_2 : \beta}{\Gamma, \Sigma \vdash e_1 := e_2 : \textsc{Task Unit}}
$$

Figure 3: Typing rules

1.2 Semantics

TopHat semantics consist of three layers: the host layer, the internal layer, and the external layer.

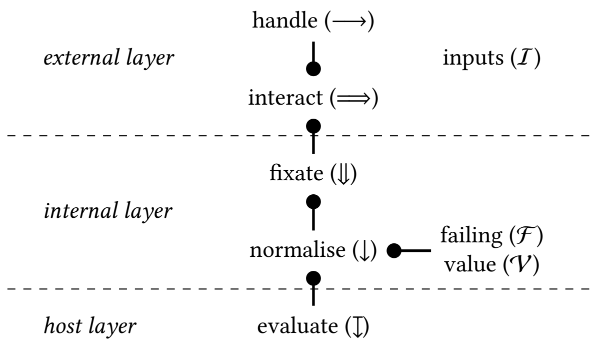

Figure 4: Semantic functions and their relation (read x •– y as 'x makes use of y')

- The *host layer* is responsible for evaluating expressions in our host language, the simply typed λ-calculus with basic types and references. This layer only contains big-step evaluation relation (↧).

- The *internal layer* only operates on tasks. It normalises tasks so that they can be presented to end users to interact with. This layer contains the big-step normalisation ($\downarrow$) and fixation ($\Downarrow$) relations.
- The *external layer* takes inputs from end users and rewrites tasks based on this input. This layer contains two labelled small-step handling ($\xrightarrow{\iota}$) and interaction ($\xRightarrow{\iota}$) relations.

Next to these semantic relations, TopHat uses *observations* on tasks to decide which semantic rule will fire. These observations include, amongst others, the value of a task, if a task is failing or not, and the set of possible inputs a task can handle. Fig. 4 shows an overview of all semantic relations and observations used in each layer.

2 DYNAMIC TOPHAT

2.1 Language

Although it is possible in TopHat to start multiple tasks in parallel, the amount of these task should be specified by the programmer at develop time.

Example 2.1 (Vending machine with shared stock). Take this slightly more complex vending machine, where two buyers can purchase a Snack at the same time from a shared inventory.

```
let vend : Task Snack = λss.
  □Int ▶ λn.
  case n of
    1 ↦ decrease Biscuit ss ▶ λ_.
          ◙Biscuit
    2 ↦ decrease Chocolate ss ▶ λ_.
          ◙Chocolate
    _ ↦ ↯
let machine : Task (Snack × Snack) =
  share [(Biscuit, 24), (Chocolate, 16)] ▶ λss.
  vend ss ⧓ vend ss
```

Here decrease is a function that decreases the amount of the given snack by one in the shared list of snacks. Note that this task results in a pair containing the two bought Snacks.

Dynamic TopHat is a moderate extension to the TopHat base language where we add support for *task pools*. A task pool is a new combinator at the task level of the TopHat language. The combinator ($\bowtie^{k}_{t_0}[\,]$) is parametrised by an unique key k, a template task t_0 and (initially) an empty task list $[\,]$. Each time end users send an *init* action to this pool (k ! $\circledast$), the template task is placed in the list of current running tasks, in this case resulting in $\bowtie^{k}_{t_0}[t_0]$. This way, an arbitrary amount of tasks can be started at runtime. Tasks in the task list of a pool can also be killed by sending a *kill* action to the pool (k ! $\ominus 1$). In this case the first task in the pool will be killed. The syntactic extensions to TopHat are shown in Fig. 5.

Example 2.2 (Dynamic number of vending machines). Using the same vend task from Example 2.1, we can now create a vending machine where end users can start an arbitrary amount of vending tasks.

```
let machine : Task (List Snack) =
  share [(Biscuit, 24), (Chocolate, 16)] ▶ λss.
  ⋈ vend ss []
```

Sending $\circledast$ to above task creates a new vend task which will ask for an amount of money and will result in a Biscuit or Chocolate. Note that this machine, contrary to the one in Example 2.1, will run indefinitely.

$$
\begin{array}{rcll}
t & ::= & \ldots & \text{Tasks} \\
 & | & \bowtie^{v}_{t_0}[\bar{t}] & \text{– pool} \\
n & ::= & \ldots & \text{Normalised tasks} \\
 & | & \bowtie^{v}_{t_0}[\bar{n}] & \text{– pool} \\
\alpha & ::= & \ldots & \text{Actions} \\
 & | & \circledast & \text{– init task in pool} \\
 & | & \ominus j & \text{– kill task in pool}
\end{array}
$$

Figure 5: Grammar for Dynamic TopHat

2.2 Semantics

Static and dynamic semantics of task pools are given in Fig. 6, and extended observations in Fig. 7.

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

T-Pool
$$\frac{\Gamma, \Sigma \vdash t_0 : \textsc{Task}\ \tau \qquad \text{for each } t_i \in \bar{t} \quad \Gamma, \Sigma \vdash t_i : \textsc{Task}\ \tau}{\Gamma, \Sigma \vdash \bowtie_{t_0}[\bar{t}] : \textsc{Task}\ (\textsc{List}\ \tau)}$$

$$\boxed{t, \sigma \downarrow n', \sigma', \delta'}$$

N-Pool
$$\frac{t_1, \sigma \downarrow n_1, \sigma_1, \delta_1 \qquad \ldots \qquad t_j, \sigma_{j-1} \downarrow n_j, \sigma', \delta_j}{\bowtie^{k}_{t_0}[\bar{t}], \sigma \downarrow \bowtie^{k}_{t_0}[\bar{n}], \sigma', \bigcup_j \delta_j}$$

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

H-Init
$$\frac{}{\bowtie^{k}_{t_0}[\bar{n}], \sigma \xrightarrow{k!\circledast} \bowtie^{k}_{t_0}[\bar{n}, t_0], \sigma, \varnothing}$$

H-Kill
$$\frac{}{\bowtie^{k}_{t_0}[n_1, \ldots, n_i, \ldots, n_j], \sigma \xrightarrow{k!\ominus i} \bowtie^{k}_{t_0}[n_1, \ldots, n_j], \sigma, \varnothing}$$

H-Pool
$$\frac{n_1, \sigma \xrightarrow{k'!\alpha} n'_1, \sigma'_1, \delta'_1 \qquad \ldots \qquad n_j, \sigma_{j-1} \xrightarrow{k'!\alpha} n'_j, \sigma', \delta'_j}{\bowtie^{k}_{t_0}[\bar{n}], \sigma \xrightarrow{k'!\alpha} \bowtie^{k}_{t_0}[\overline{n'}], \sigma', \bigcup_j \delta'_j}$$

Figure 6: Semantic rules for Dynamic TopHat

$$\begin{aligned}
&\mathcal{V} : \text{Normalised task} \times \text{State} \rightharpoonup \text{Value}\\
&\mathcal{V}(\bowtie^{k}_{t_0}[\overline{n}], \sigma) \;=\; [\mathcal{V} n_i \mid n_i \in \overline{n}]\\
&\mathcal{F} : \text{Task} \longrightarrow \text{Boolean}\\
&\mathcal{F}(\bowtie^{v}_{t_0}[\overline{t}]) \;=\; \mathsf{False}\\
&\mathcal{I} : \text{Normalised task} \longrightarrow \mathcal{P}(\text{Input})\\
&\mathcal{I}(\bowtie^{k}_{t_0}[\overline{n}]) \;=\; \{\mathcal{I}(n_i) \mid n_i \in \overline{n}\}\\
&\qquad\qquad \cup \{k \,!\, \circledast\} \cup \{k \,!\, \ominus i \mid i \in 0 \,..\, \#\overline{n}\}
\end{aligned}$$

Figure 7: Observations for Dynamic TopHat

2.3 Meta theory

Propositions from [Steenvoorden 2022] and [Klijnsma 2020] that still hold are the following:

- Valued tasks are static
- Steady tasks stay steady

However, because of the dynamic nature of task pools, the following propositions need alteration:

- Static tasks stay static
- Finished tasks stay finished

This is because task pools can always receive the initialise input ($\circledast$), so tasks containing a task pool will never get to a steady state!

We will discuss the impact on equational reasoning on tasks more thoroughly...

3 DISCUSSION

One could think of an alternative implementation of task pools, using already existing TopHat language constructs. A possible solution could be by using a shared reference to a list of running tasks, and creating custom tasks to remove a task by its index from this list or append the template task to this list.

```
let kill = λts. □Int ► λi. remove i ts
let init = λt. λts. □Unit ► λ_. append t ts
let pool = λt.
  share [] ► λts.
  ↻ (⊞ h ⧓ kill ts ⧓ init t ts)
```

This is effectively the way that iTasks implements dynamic task lists [Plasmeijer et al. 2011]. However, there are two reasons this is not a good fit for TopHat.

The first is that the forever combinator (↻), is not definable in TopHat. Our goal is to keep our host language total. The combinator could be added explicitly to the task layer, but this would impose severe restrictions on the symbolic execution engine, in the same way non-total functions would do.

The second is that it would require references and editors over the Task type constructor. Currently, both are restricted to basic types, because basic types are trivially displayable, parsable and equatable. This last property is used by [Klijnsma 2020] as a basis to reason about task equivalence.

This raises the question if tasks are more like basic types, or if they should behave more like functions. What would $\boxminus(\mathit{Update}\ 42)$ mean? How can end users dynamically enter a task that results in an integer? Could they create new tasks at will? Or should they pick from a list? Is this list predefined or is it populated dynamically during runtime? Although iTasks supports editors on tasks, above example cannot be rendered at runtime and the equality on two tasks in iTasks is always True.

Therefore, we think explicitly adding task pools to TopHat is a better fit for a language which main goal is formal reasoning about task-oriented programs.

ACKNOWLEDGMENTS

REFERENCES

Tosca Klijnsma. 2020. Semantic Equivalence of Task-Oriented Programs in TopHat. Supervised by Tim Steenvoorden and Herman Geuvers.

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, 47–68. https://doi.org/10.1007/978-3-030-57761-2_3

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, 1:1–1:11. https://doi.org/10.1145/3412932.3412933

Rinus Plasmeijer, Peter Achten, Pieter W. M. Koopman, Bas Lijnse, Thomas van Noort, and John H. G. van Groningen. 2011. iTasks for a change: type-safe run-time change in dynamically evolving workflows. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*. 151–160. https://doi.org/10.1145/1929501.1929528

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

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, 17:1–17:13. https://doi.org/10.1145/3354166.3354182