A Reflection on Task-Oriented Programming

Mart Lubbers[1][0000−0002−4015−4878] and Tim Steenvoorden[2][0002−8436−2054]

[1] Institute for Computing and Information Sciences,
Radboud University, Nijmegen, The Netherlands
mart@cs.ru.nl
[2] Computer Science
Open Universiteit, Heerlen, The Netherlands
tim.steenvoorden@ou.nl

Abstract. Task-Oriented Programming 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 two task 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, you have to create the behaviour by combining simple core combinators. The contribution is twofold, we perform a qualitative and quantitative analysis of task combinator usage in all published case study applications, some examples from the iTask distribution and a sizeable real-world industrial application. We reflect on combinator usage and show that over 95% of real-world task-oriented code is expressible using the core combinator approach once it is extended with another combinator: `reflect`.

Keywords: Task-Oriented Programming

1 Introduction

Task-Oriented Programming (TOP) is a relatively new programming paradigm [18]. It is a declarative programming paradigm where tasks are the basic building blocks. 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. Besides exposing the progress of a task via its task value, tasks can also share data using Shared Data Sources (SDS). 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 complex workflow systems.

There are several implementations of task-oriented programming, for example iTask, an industry-grade TOP system for distributed web applications; TOPPYT, a TOP implementation in Python; mTask, a TOP language for microprocessors that integrates with iTask; and TopHat, a fully formalised TOP language.

The iTask system is an TOP implementation that generates an interactive multi-user distributed web server that allows user to perform the work that was specified [16]. It has a long history and the set of task combinators changed continuously throughout the years [9]. Furthermore, the many documented case studies in literature and the usage in industry results in a relatively large codebase of real-world TOP applications. The philosophy behind iTask is that with two super combinators, all other combinators could be derived. So this means that there is only one sequential super combinator (`step`), and one parallel super combinator (`parallel`). As a consequence, deriving new combinators is relatively easy, but understanding or changing the exact semantics of the super combinators is very difficult. Attempts have been made but always only on a subset of iTask [8].

TopHat is a TOP implementation that is fully mathematically formalised [22]. The design of task combinators in TopHat is exactly opposite of iTask. Instead of deriving all combinators from two complex super combinators, there is a rich set of core combinators from which more complex combinators are derived. Over the years, the set of core combinators of TopHat has been extended to cover more and more of the real-world workflow patterns. For example, doing things in parallel, and allowing the user to dynamically spawn more tasks [21].

1.1 Research contribution

In this paper we analyse twelve published case studies, two of iTask's internal workflow applications and a real-world industrial application. Furthermore we introduce `reflect`, a new core combinator that can expose a task's task value to siblings. With this new combinator, w capture over 95%[3] of the real-world TOP combinator use.

2 Examples

Reflect is mostly use together with a selection and a whileUnchanged.

For example in incidone (cite)

taxman (cite)

interactive test suite (from iTask, no citation)

workflow admin (from iTask, no citation)

store admin (from iTask, no citation)

codequalitymonitor (from iTask, no citation)

[3] draft preliminary

3 Semantics

We present our formal semantics as an extension of TopHat[23]. TopHatis a formal semantics of task-oriented programming, with a verified implementation in Idris and a practical one in Haskell. It specifies the semantics of basic task-oriented operations. The framework has been extended for symbolic execution of tasks [14], and next-step hint generation [13]. Also, it is the foundation of proving equivalence of task definitions [7].

3.1 Overview of TopHat

TopHat semantics is defined in three layers: the host layer, the internal layer, and the external layer. These are depicted in Fig. 1. At the bottom, there is the *host layer*, which evaluates pure lambda terms. On top of that, there are two *task layers*. 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. Besides semantic arrows, TopHat has the notion of *observations*. These are summarized at the right side of Fig. 1.

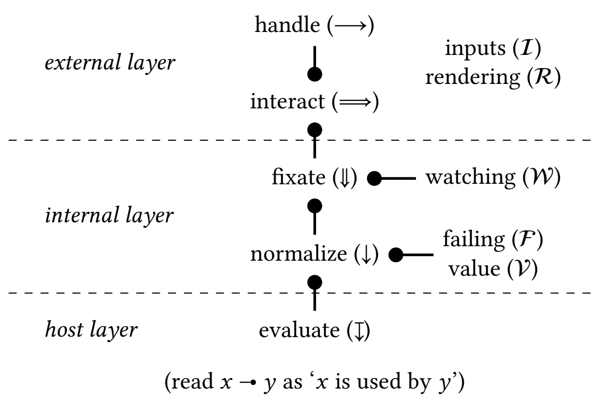

Fig. 1. Overview of semantic layers, relations and functions in TopHat.

Our reflection extension does not alter the host layer. In these layers, only the handle (⟶) and normal (↓) semantics need to be extended. Below, we first introduce the host and task languages of TopHat. Next, we introduce observations on tasks and the influence of reflection. After that, we show the additions to the normalisation and handling semantics.

3.2 Host and task languages

TopHat's host language is the simply typed λ-calculus with *basic types* such as Booleans, integers, and strings, extended with product and sum types. It also contains *heap locations*, 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 heap locations are restricted to only contain basic types, that is, no functions nor other heap locations. 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 Fig. 2 Terms e, v, and b are expressions, values, and values of basic types in the host language. Type β stands for basic types. Next, we'll discuss the operators in the language. For more details about types and expressions in the host layer, we refer to previous work [20].

Editors

$$
\begin{array}{rcll}
d & ::= & \square^{\nu}\beta \mid \boxminus^{\nu} b \mid \boxdot^{\nu} b & \text{– unvalued, valued, read-only} \\
& \mid & \boxplus^{\nu} h \mid \boxplus^{\nu} h & \text{– shared, read-only}
\end{array}
$$

Tasks

$$
\begin{array}{rcll}
t & ::= & d \mid \blacksquare v \mid \lightning & \text{– editor, done, fail} \\
& \mid & v_1 \bullet t_2 \mid t_1 \blacktriangleright v_2 & \text{– transform, step} \\
& \mid & t_1 \bowtie t_2 \mid t_1 \blacklozenge t_2 & \text{– pair, choose} \\
& \mid & \textbf{share}\ b \mid h_1 := b_2 & \text{– share, assign}
\end{array}
$$

Fig. 2. Grammar of TopHat's task language.

Editors Editors are the end-points of a task, used to interact with end users. They are an abstraction over input fields or widgets. Editors are typed, which means that, for example, in an Int editor, end users can really only fill in integers. Editors come in multiple flavours. *Unvalued editors* currently do not contain a value yet. They need to be filled with a value of the appropriate type. *Valued editors* do contain such a value, which can be modified by end users. *Read-only editors* also contain a value, but cannot be modified. We will discuss editors on shared date shortly hereafter.

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* $t_1 \blacktriangleright v_2$. Here, when task t_1 has an observable value, this value is passed on to function v_2 which calculates its continuation, also a task. When this calculated continuation happens to be *fail* ($\lightning$), the step is not made and we stay working on t_1.

The parallel combinators come in two forms: pair and choose. 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, the combination does not have a value. Choosing between two task $t_1 \blacklozenge t_2$ also means one can work on both tasks interleaved. However, the observed value is the value of t_1, if it is available, otherwise, we choose the value of t_2. If both are unavailable, the combination also does not have a value.

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 to 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 on a heap and returns a heap location h. Using this h, multiple tasks can watch the same data. For example, *shared editors* watch a heap location, show it to end users, and allow them to change it. Similarly, *read-only shared editors* watch a heap location, but end users cannot modify it. The application itself can set heap locations to any basic value using $h_1 := b_2$.

3.3 Observations

Tasks form syntax trees which can be *observed*. The most important observation on tasks is their current *value*. This is a partial function from task trees to values. For example, the unvalued editor □Bool of Booleans, does not have a value yet. Such a value can be entered into the editor by sending it the *input* True. This rewrites the task to the valued editor ⊟42, which currently has value 42. Value observations are defined recursively on task trees. Notably steps never have a value, as we cannot tell what continuation it will evaluate to.

Invariant:

$$\mathcal{V}(\circledcirc_h n, \sigma) = \mathcal{V}(n, \sigma) \neq \sigma(h)$$

But this does not hold!

$$\textbf{share}\ \text{Nothing} \blacktriangleright \lambda h.\ \circledcirc_h \blacksquare 38 \bowtie h := 42$$

Normalises to something which sets h to 38, so the value of the left task is not reflected in the heap location. Can be solved by using read-only memory locations for the programmer.

$$\mathcal{F}(\circledcirc_h t) = \mathcal{F}(t)$$

$$\mathcal{W}(\circledcirc_h n) = \mathcal{W}(n) \cup \{h\}$$

$$\mathcal{R}(\circledcirc_h n, \sigma) = \mathcal{R}(n, \sigma)$$

$$\mathcal{I}(\circledcirc_h n) = \mathcal{I}(n)$$

3.4 Normalisation and handling

$$\textsc{N-Reflect}\quad \frac{t, \sigma \downarrow n', \sigma', \delta'}{\circledcirc_h t, \sigma \downarrow \circledcirc_h n', [h \mapsto v]\sigma', \delta' \cup \{h\}}\ \mathcal{V}(n', \sigma') = v$$

$$\textsc{H-Reflect}\quad \frac{n, \sigma \xrightarrow{\iota} t', \sigma', \delta'}{\circledcirc_h n, \sigma \xrightarrow{\iota} \circledcirc_h t', \sigma', \delta'}$$

3.5 Sugar

$$\begin{aligned}
\lfloor t \rfloor &:= t \blacktriangleright \lambda_.\ \lightning \\
\lceil t \rceil &:= t \blacktriangleright \lambda x.\ \blacksquare x \\
t \circledgt e &:= \textbf{share}\ \mathsf{Nothing} \blacktriangleright \lambda h.\ \lfloor \circledcirc_h t \rfloor \blacklozenge e\ h \\
t \circledless e &:= \textbf{share}\ \mathsf{Nothing} \blacktriangleright \lambda h.\ \circledcirc_h\ t \blacklozenge \lfloor e\ h \rfloor \\
e_1 \circledwedge e_2 &:= (\textbf{share}\ \mathsf{Nothing} \bowtie \textbf{share}\ \mathsf{Nothing}) \blacktriangleright \lambda(h_1, h_2).\ \circledcirc_{h_1}\ (e_1\ h_2) \bowtie \circledcirc_{h_2}(e_2\ h_1)
\end{aligned}$$

4 Applications

There are many test programs, case studies and entire applications that have been published in literature. We analysed these applications with a tool that uses the compiler to gather some statistics about the usage of task combinators.

- conf2009, a conference management system [17].
- itasks22009, a set of example programs for iTask 2 [11].
- trax2013, a single-player puzzle game [1].
- gin2012, the frontend for GiN, an graphical interactive task creater [6].
- incidone2012, an incident report application [10].
- tonic2014, the fronted for Tonic, a visualisation tool of iTask tasks [25].
- ligretto2014, a multi-user card game game [2].
- tasklets2015, bigger examples for executing small tasks in the browser using TaskLets and EditLets [4, 3, 5].
- shipadventure2017, an interactive fire-extinguishing game situated on a naval ship [24].
- serviceengineer2017, a distributed multi-user application to manage and perform job allocation for service engineers [15].
- taxman2018, workflow system for entering solar panel reimbursements [24].
- cws2023, smart campus monitoring system prototype [12].
- admin2024, several administrative task workflows for the iTask system to administrate the server itself [18].
- basicapiexamples2024, a set of example programs [18].
- VIIA2024, Vessel Information Integrating Application, a commercial application to monitor coasts [19].

Goed opletten, lange tijd gebruikte iTask de »= als bind, dus we moeten onderscheid maken tussen iTask modules en monadische modules en hopen dat ze niet door elkaar gebruikt worden.

niet door elkaar gebruikt worden. Table here with statistics

T

Table 1. Some statistics…

Sequence		50%
Parallel	and	50%
	or	50%
	all	50%

5 Related work

6 Conclusions

6.1 Discussion

…if needed…

6.2 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. It would be interesting to see if and what core combinator we would need in order to express this behaviour as well.

Furthermore, the super combinator `parallel` allows tasks to add, remove or even replace sibling tasks automatically. Figuring out which core combinators can provide this behaviour is ongoing research.

References

1. Achten, P.: Why Functional Programming Matters to Me. In: Achten, P., Koopman, P. (eds.) The Beauty of Functional Code, vol. 8106, pp. 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40355-2_7
2. Achten, P., Stutterheim, J., Domoszlai, L., Plasmeijer, R.: 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. p. 7. ACM (2014)
3. Domoszlai, L., Kozsik, T.: Clean Up the Web! In: Achten, P., Koopman, P. (eds.) The Beauty of Functional Code: Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday, pp. 133–150. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40355-2_10
4. Domoszlai, L., Lijnse, B., Plasmeijer, R.: 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 (2014). https://doi.org/10.1145/2746325.2746331, event-place: Boston, MA, USA
5. Domoszlai, L., Plasmeijer, R.: Tasklets: Client-Side Evaluation for iTask3. In: Zsók, V., Horváth, Z., Csató, L. (eds.) Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers, pp. 428–445. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-15940-9_11

6. Henrix, J., Plasmeijer, R., Achten, P.: GiN: A Graphical Language and Tool for Defining iTask Workflows. In: Peña, R., Page, R. (eds.) Trends in Functional Programming. pp. 163–178. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
7. Klijnsma, T., Steenvoorden, T.: Semantic equivalence of task-oriented programs in tophat. In: Swierstra, W., Wu, N. (eds.) 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, pp. 100–125. Springer (2022). https://doi.org/10.1007/978-3-031-21314-4_6, https://doi.org/10.1007/978-3-031-21314-4_6
8. Koopman, P., Plasmeijer, R., Achten, P.: An Executable and Testable Semantics for iTasks. In: Scholz, S.B., Chitil, O. (eds.) Implementation and Application of Functional Languages. pp. 212–232. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
9. Lijnse, B.: Evolution of a Parallel Task Combinator. In: Achten, P., Koopman, P. (eds.) The Beauty of Functional Code: Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday, pp. 193–210. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40355-2_14
10. Lijnse, B., Jansen, J.M., Plasmeijer, R., others: 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 (2012)
11. Lijnse, B., Plasmeijer, R.: iTasks 2: iTasks for End-users. In: International Symposium on Implementation and Application of Functional Languages. pp. 36–54. Springer (2009)
12. Lubbers, M., Koopman, P., Ramsingh, A., Singer, J., Trinder, P.: Could Tierless Languages Reduce IoT Development Grief? ACM Trans. Internet Things **4**(1) (Feb 2023). https://doi.org/10.1145/3572901, https://doi.org/10.1145/3572901, place: New York, NY, USA Publisher: ACM
13. Naus, N., Steenvoorden, T.: Generating next step hints for task oriented programs using symbolic execution. In: Byrski, A., Hughes, J. (eds.) 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, pp. 47–68. Springer (2020). https://doi.org/10.1007/978-3-030-57761-2_3, https://doi.org/10.1007/978-3-030-57761-2_3
14. Naus, N., Steenvoorden, T., Klinik, M.: A symbolic execution semantics for tophat. In: Stutterheim, J., Chin, W. (eds.) IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019. pp. 1:1–1:11. ACM (2019). https://doi.org/10.1145/3412932.3412933, https://doi.org/10.1145/3412932.3412933
15. Oortgiese, A., van Groningen, J., Achten, P., Plasmeijer, R.: A Distributed Dynamic Architecture for Task Oriented Programming. In: Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages. p. 7. ACM, New York, NY, USA (2017), event-place: Bristol, UK
16. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. ACM SIGPLAN Notices **42**(9), 141–152 (2007)
17. Plasmeijer, R., Achten, P., Koopman, P., Lijnse, B., van Noort, T.: An iTask Case Study: A Conference Management System. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures, pp. 306–329. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04652-0_7
18. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-Oriented Programming in a Pure Functional Language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. pp. 195–206. PPDP '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2370776.2370801, event-place: Leuven, Belgium
19. Software, T.: VIIA (Vessel Information Integrating Application) (2023), https://www.top-software.nl/VIIA.html

20. Steenvoorden, T.: Visual task-oriented programming using taskflows. In: Zsók, V., Horváth, Z., Grelck, C. (eds.) Central European Functional Programming School - 8th Summer School, CEFP 2019, Budapest, Hungary, June 17–21, 2019, Revised Selected Papers. Lecture Notes in Computer Science, Springer (2019), accepted for publication
21. Steenvoorden, T., Naus, N.: 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 (2024). https://doi.org/10.1145/3652561.3652574, event-place: Braga, Portugal
22. Steenvoorden, T., Naus, N., Klinik, M.: 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 (2019). https://doi.org/10.1145/3354166.3354182, https://doi.org/10.1145/3354166.3354182, event-place: Porto, Portugal
23. Steenvoorden, T., Naus, N., Klinik, M.: Tophat: A formal foundation for task-oriented programming. In: Komendantskaya, E. (ed.) Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019. pp. 17:1–17:13. ACM (2019). https://doi.org/10.1145/3354166.3354182, https://doi.org/10.1145/3354166.3354182
24. Stutterheim, J., Achten, P., Plasmeijer, R.: Maintaining Separation of Concerns Through Task Oriented Software Development. In: Wang, M., Owens, S. (eds.) Trends in Functional Programming, vol. 10788, pp. 19–38. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89719-6
25. Stutterheim, J., Plasmeijer, R., Achten, P.: Tonic: An infrastructure to graphically represent the definition and behaviour of tasks. In: International Symposium on Trends in Functional Programming. pp. 122–141. Springer (2014)