Creating Interactive Visualizations of TopHat Programs

Mark Gerarts
Open University
Heerlen, The Netherlands
mark.gerarts@gmail.com

Marc de Hoog
Open University
Heerlen, The Netherlands
mla.dehoog@studie.ou.nl

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

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

ABSTRACT

Many companies and institutions have automated their business process in workflow management software. The novel programming paradigm Task-Oriented Programming (TOP) provides an abstraction for such software. The largest framework based on TOP, iTasks, has been used to develop real-world software. Workflow software often includes critical systems, dealing with the safety of infrastructure and people. In such cases it is important to reason over the software to ascertain its correctness. To this end TopHat has been developed. TopHat is a TOP language with a formal semantics. However, there is no user interface available for TopHat, making it harder to use TopHat to develop actual systems. In this paper we develop a user interface for TopHat. By combining a server framework and a user interface framework, we have developed a fully functioning proof of concept implementation. We show that implementing a TOP language is possible. This is done by developing an implementation in Haskell, and running several example programs. The results of this paper show that you can have a system that has a formal semantics and a user interface. Having such a system brings TOP to the Haskell community and improves the quality and verifiability of TOP software in general.

CCS CONCEPTS

• **Information systems** → *Enterprise information systems*; *Web applications*; • **Applied computing** → *Enterprise information systems*; • **Software and its engineering** → **Domain specific languages**.

KEYWORDS

task oriented programming, user interface, functional programming

ACM Reference Format:
Mark Gerarts, Marc de Hoog, Nico Naus, and Tim Steenvoorden. 2021. Creating Interactive Visualizations of TopHat Programs. In *IFL '21: 33rd Symposium on Implementation and Application of Functional Languages, September 1–03, 2021, Online.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Workflow automation software is present in most businesses and institutions nowadays. From health care and first responders, to commerce and industrial processes. Businesses use workflow software to streamline their processes, increase efficiency and reduce costs. In these sectors, reliability of software is crucial.

Previous research into workflow automation software in the functional programming community aimed to improve reliability, while at the same time reducing the effort of development. This lead to the development of Task-Oriented Programming (TOP), a programming paradigm that aims to facilitate working with multiple people towards a shared goal over the internet. TOP separates the *what* from the *how*. This separation allows programmers to focus on the work that has to be done (*what*) instead of paying attention to design issues, implementation details, operating system limitations, and environment requirements (*how*) [1][23].

TOP is centred around the concept of *tasks*, which specify the work a user or system has to perform with a high level of abstraction. Tasks can be combined using combinators, allowing complex programs to be constructed from small building blocks [23].

Tasks provide a description of the work that has to be performed. It is left to the TOP framework to implement technical details such as event handling or creating a User Interface (UI). *iTasks* [1] is such a framework, implemented in the functional programming language Clean [8]. An example of a basic task in iTasks is presented in Listing 1. As a developer you only have to specify that you want the user to enter some information. Passing this task to iTasks generates an application that prompts the user for their name.

```
enterName :: Task String
enterName = Hint "What is your name?" @>>
    enterInformation []
```

Listing 1: A simple task prompting the user for their name (Clean)

iTasks has been used to create real-world applications, such as an incident coordination tool for the Dutch coast guard [16]. While this proves its practical usability, iTasks lacks in formalization. Formal program verification is necessary to ensure the correctness of mission-critical software, like the incident coordination tool. TopHat is a Domain-Specific Language (DSL) that paves the way to formally reason about task-oriented programs [29], by defining a formal semantics for tasks and combinators. It lacks an interactive web UI, and for this, TopHat relies on the iTasks framework.

Having an interactive UI for TopHat provides us with the best of both worlds, a TOP language that is both suitable for formal verification and usable in practice. In this paper we present a prototype framework written on top of TopHat's Haskell implementation that is able to create interactive visualizations of TopHat programs. We combine the best of both worlds: we generate practical task-oriented applications that can be formally verified. By using Haskell we hope to bring TOP within reach of researchers and programmers who are not familiar with Clean and iTasks.

iTasks is written in Clean, while the TopHat framework uses Haskell as its host language.

Clean was first introduced in 1987 [6], and has since been an inspiration for Haskell and vice versa. While both languages are largely similar, there are a few key differences that can be problematic. The most notable difference is the handling of I/O: Haskell uses the IO monad, while Clean uses uniqueness typing [4]. Another difference is Clean's generic programming system, which is heavily used by the iTasks framework. iTasks relies on many of these Clean specific language features. This begs the question if these constructs are strictly necessary for the development of a TOP framework, or if such a system can also be developed in a different language, in our case Haskell.

Work presented in this paper is based on the thesis project published under the same title [14].

We first provide some background about TOP in Section 2. We then demonstrate how our framework renders some example applications and take a deeper look at the architecture in Section 3. We highlight related work in Section 4 and provide a conclusion in Section 5.

2 TASK ORIENTED PROGRAMMING

This paper builds upon previous research into Task-Oriented Programming (TOP). In this section we describe the basic idea of TOP, together with two TOP frameworks; iTasks and TopHat. We compare the two frameworks and then conclude why there is a need for a graphical user interface for Tophat.

2.1 Task-Oriented Programming

The TOP paradigm provides an abstraction over workflow software. Instead of having to write a server, database, user interfaces, etc, programmers just define what needs to be done. The complete application is then derived from this specification. TOP is usually embedded in pure functional programming. TOP is made up of four core concepts [23]:

Tasks that describe the work that has to be performed, providing an abstraction that separates the *what* from the *how* [1].

Shared data sources that allow the sharing of data between tasks.

Generics to generate user interfaces based on data types.

Composition of tasks through combinators, allowing the creation of arbitrary large tasks. Composition can be both sequential and parallel.

TOP aims to facilitate collaborating with multiple people towards a shared goal, over the internet. Tasks lie at the heart of TOP. A task models the work that has to be done by the system or a user. Tasks can be combined using combinators: they can be executed sequentially, in parallel, or conditionally. These combinators closely resemble how collaboration happens in real life.

Combining small tasks allows creating large and complex application using simple building blocks. Creating complex applications is further facilitated because tasks are first-class citizens: they can be used as input of functions, they can be returned from them, and tasks can contain other tasks as value.

Tasks are interactive and input-driven. When a task receives input it is reevaluated and results in a new task. A task's value can be observed at all times, and tasks can share information with each other, either directly through shared data stores, or by passing task values to continuations.

2.2 iTasks

TOP focuses on the domain logic, with tasks providing merely a description of the work that has to be performed. It is left up to a TOP framework to do the heavy lifting, such as generating the user interface, storing and handling data, setting up a web server, and authenticating users. iTasks [22] is a TOP framework that uses Clean [6] as its host language. It supplements Clean with a set of combinators, model types, and algorithms that allow the construction of task-oriented programs.

An example of a basic task is given in Listing 1. iTasks will automatically generate an entire application for this task. It uses generics to deduce that a task of type `String` requires a text input field.

Tasks can be combined through combinators, allowing to construct complex applications from small building blocks. In Listing 2 we combine the previous task with a view task using a sequential step combinator. A user has to enter their name and is greeted by the program after stepping to the next task. Figure 1 shows how these steps would look in iTasks.

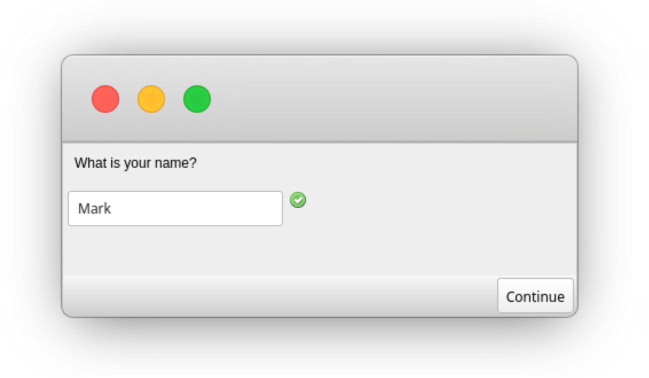
```
greet :: Task String
greet = enterName >>!
          \result -> viewInformation [] ("Hello " +++
     result)
```

Listing 2: Combining two tasks with a step combinator (Clean)

iTasks is a work in progress, receiving constant updates and improvements. For example, a recent addition is the usage of a distributed, dynamic infrastructure [21]. iTasks has formed the basis of further research as well. Tonic [31] facilitates the subject for non-technical people by providing graphical blueprints of iTasks specifications. It also provides a way to monitor the process while end users are interacting with the application [30]. iTasks acted as the starting point for research into declarative user interfaces, first for SVG images [2] and later as a generalized solution [3].

2.3 TopHat

When software is used in critical applications, it is important that its behaviour can be verified and formally reasoned about. iTasks is primarily focused on practical applicability, and therefore lacks this


What is your name?

Mark

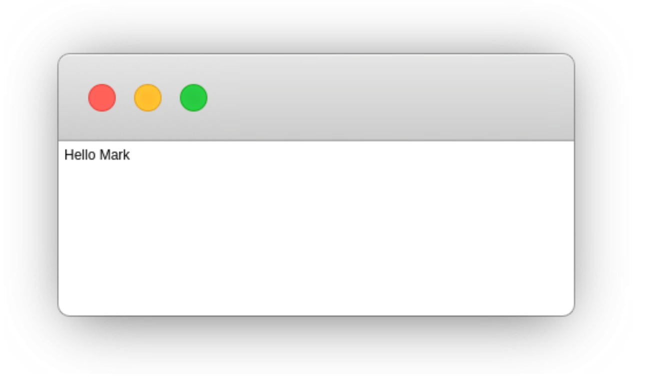

Hello Mark

Figure 1: Entering your name (left) and the result after pressing continue (right)

formalisation. Testing an iTasks application is time consuming and often incomplete because of the many different execution paths.

TopHat [29] distills TOP's core features into a DSL that aims to provide a way to reason about task-oriented programs. By employing symbolic execution it is possible to formally verify TopHat programs [19]. Symbolic execution has also been used to provide end-users of tasks with additional feedback [20].

Our work is based on TopHat's Haskell implementation, but an implementation exists for Idris as well. Idris is a programming language that features dependent types and a totality checker.

An example task written in TopHat's Haskell implementation is given in Listing 3. Similar to the iTasks example, this task uses a step combinator to ask a user their name and subsequently greet them.

```haskell
greet :: Task h String
greet = enter >>? \result -> view ("Hello " ++ result)
```

Listing 3: A TopHat task that greets the user (Haskell)

TopHat contains the following set of tasks and combinators:

- **Editors** editors model user interaction. They are typed containers that are either empty or hold a value.
  - **Update editor** an editor with a predefined value.
  - **View editor** an editor with a view-only value.
  - **Enter editor** an editor that is initially empty. Filling it transforms it into an Update editor.
  - **Select editor** an editor with a predefined list of options.
  - **Watch editor** an editor that displays the value of a shared data store.
  - **Change editor** an editor that allows to change the value of a shared data store.
- **Done and Fail** success and failure end states.
- **Pair** a combination of two tasks (parallel task composition).
- **Step** sequentially moves from one task to another.
- **Choose** internal choice between two tasks.
- **Assign** used in the context of shared data stores to assign a value to a reference.

2.4 Research question

Above we introduced TopHat as a tool to formally reason about task-oriented programs. There is a need for a TOP framework with TopHat as its foundation, because such framework can render a web UI directly, without a translation to iTasks, while formal reasoning about TOP programs is still possible. This paper answers the question: *To what extent is it possible to add an interactive web UI to a TopHat program without a translation to iTasks, and without the use of Clean-specific constructs?*

3 TOPHAT USER INTERFACE

In this section we describe our prototype TOP framework, which is a proof-of-concept and not a full-fledged TOP framework. Our application supports the TopHat's tasks that are mentioned in Section 2.3, except for the choice combinator. The choice combinator will be added in the post-conference version of this paper. We limit ourselves to a select number of datatypes: only integers, booleans, and strings are supported. Advanced framework features such as multi-user support are out of scope as well.

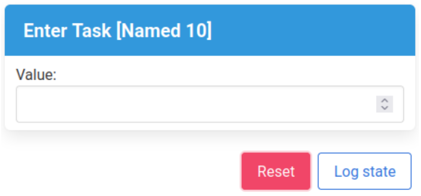
We start by presenting some example TopHat programs and how our application renders them in Section 3.1. In Section 3.2 we describe the architecture of our prototype. The framework is published on GitHub [1], along with the examples described below, and documentation on how to install and run them.

3.1 Application

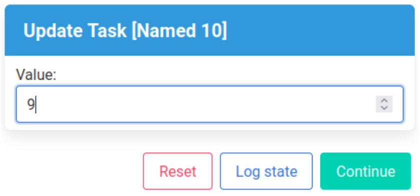
We present a few examples to demonstrate how our framework handles TopHat programs. We use a simple multiplication-by-seven machine to demonstrate the Step task and the Edit task (with View, Enter, and Update editors). The candy vending machine combines the Select and View editor, the Step Task, and the Pair Task to construct a candy machine. The calorie calculator demonstrates a real-world application of our framework. Finally, the chat sessions demonstrates the use of shared datastores.

3.1.1 Multiplication-by-seven machine. A multiplication-by-seven machine is a simple machine that multiplies a user's input by seven and displays the result. We use different tasks and editors, which we mention explicitly in the next steps. The implementation of the initial task is given in Listing 4.

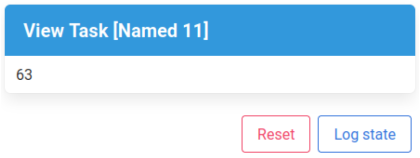
[1]https://github.com/mark-gerarts/ou-afstuderen-artefact



(a) Enter a value.



(b) Editor changes from Enter to Update.



(c) A view editor displays the answer.

Figure 2: Different stages of the multiplication-by-seven machine

```
multBySevenMachine :: Task h Int
multBySevenMachine = enter >>? multBySeven

multBySeven :: Int -> Task h Int
multBySeven = multiplication 7

multiplication :: Int -> Int -> Task h Int
multiplication x y = view (x * y)
```

Listing 4: Initial task of the multiplication-by-seven machine (Haskell)

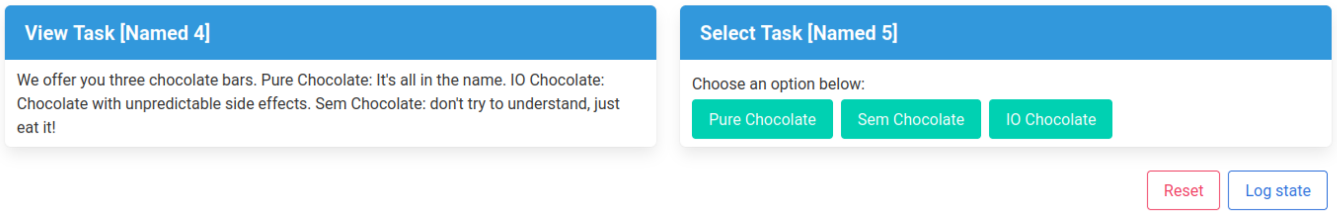
(1) After the application is started, the application renders an Edit task with an Enter editor (see Figure 2a).
(2) Enter a value.
The application sends the input to the backend and a new task is returned. This task is still an Edit task. However, the editor is changed from Enter to Update (see Figure 2b).
(3) Press the continue button or press <Enter>.
The continue button is part of the Step Task.
(4) The application shows a View task, which contains the value multiplied by seven (see Figure 2c).

3.1.2 Candy vending machine. The candy machine allows a user to choose a chocolate bar and, after the bill is paid, the candy machine returns the bar. The candy machine combines the Edit, Pair and Step task. We have defined different Edit tasks with View and Select editors. The implementation of the initial task is given in Listing 5. The Pair combinator is denoted with the operator ><.

(1) After the candy machine is started, the machine displays some introductory text and a selection of chocolate bars (See Figure 3a). This is done using a Pair Task that consists of two Edit tasks: an Edit task with a View editor and an Edit Task with a Select editor.
(2) Select a chocolate bar. After choosing a bar, the candy machine displays the price of the bar (see Figure 3b). This is done using another Pair Task that consists of an Edit task with a View editor (*"you need to pay:"*) and a Step Task. The Step task consists of two tasks: first a view editor is shown (with the price) and after the step, a select editor is rendered (see Figure 3c).
(3) Press the continue button.
(4) Insert coins until you have paid the bill (see Figure 3c). The application alternates a view and a select editor.
(5) The application shows a view editor to indicate to the user that the bill is paid (see Figure 3d).

```
data CandyMachineMood = Fair | Evil

startCandyMachine :: (Task h (Text, (Text, Text)))
startCandyMachine = view "We offer you three chocolate
    bars. Pure Chocolate: It's all in the name. IO
    Chocolate: Chocolate with unpredictable side effects.
    Sem Chocolate: don't try to understand, just eat
    it!" >< select candyOptions

candyOptions :: HashMap Label (Task h (Text, Text))
candyOptions =
  [ entry "Pure Chocolate" 8,
    entry "IO Chocolate" 7,
    entry "Sem Chocolate" 9
  ]
  where
    entry :: Text -> Int -> (Label, Task h (Text, Text))
    entry name price =
      (name, view "You need to pay:" >< (view price >>?
     payCandy))

payCandy :: Int -> Task h Text
payCandy bill =
    select (payCoin bill) >>? \billLeft ->
        case compare billLeft 0 of
        EQ -> dispenseCandy Fair
        LT -> dispenseCandy Evil
        GT -> payCandy billLeft

payCoin :: Int -> HashMap Label (Task h Int)
payCoin bill =
  [ coinSize 5,
    coinSize 2,
    coinSize 1
  ]
  where
    coinSize :: Int -> (Label, Task h Int)
    coinSize size = (display size, view (bill - size))

dispenseCandy :: CandyMachineMood -> Task h Text
dispenseCandy Fair =
    view "You have paid. Here is your candy. Enjoy it!"
dispenseCandy Evil =
    view "You have paid too much! You don't get change,
     but here is your candy."
```

Listing 5: Initial Task of the candy vending machine (Haskell)

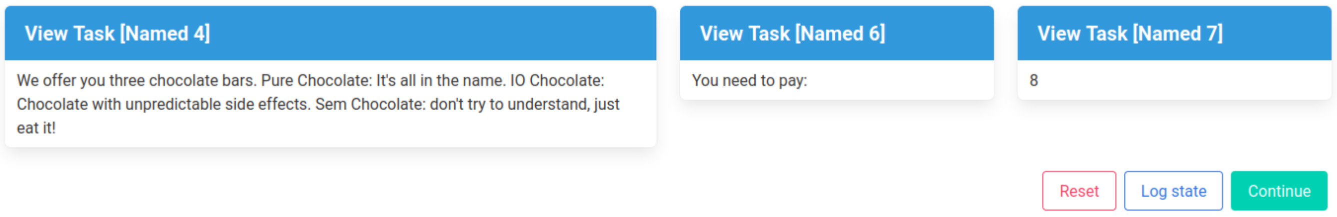

View Task [Named 4]

We offer you three chocolate bars. Pure Chocolate: It's all in the name. IO Chocolate: Chocolate with unpredictable side effects. Sem Chocolate: don't try to understand, just eat it!

Select Task [Named 5]

Choose an option below:

(a) Step 1: Select a chocolate bar


View Task [Named 4]

We offer you three chocolate bars. Pure Chocolate: It's all in the name. IO Chocolate: Chocolate with unpredictable side effects. Sem Chocolate: don't try to understand, just eat it!

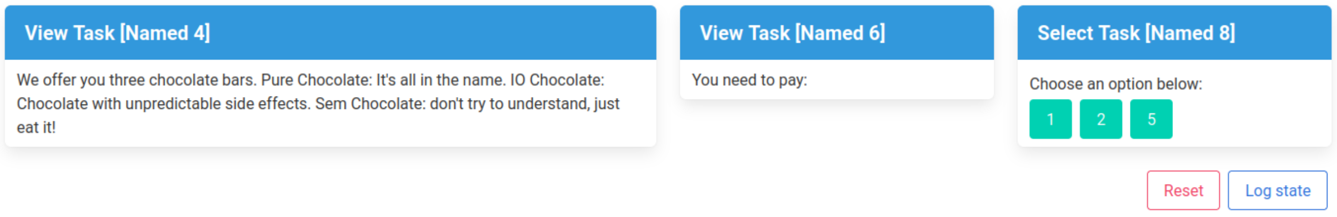
View Task [Named 6]

You need to pay:

View Task [Named 7]

8

(b) Step 2: Select a chocolate bar


View Task [Named 4]

We offer you three chocolate bars. Pure Chocolate: It's all in the name. IO Chocolate: Chocolate with unpredictable side effects. Sem Chocolate: don't try to understand, just eat it!

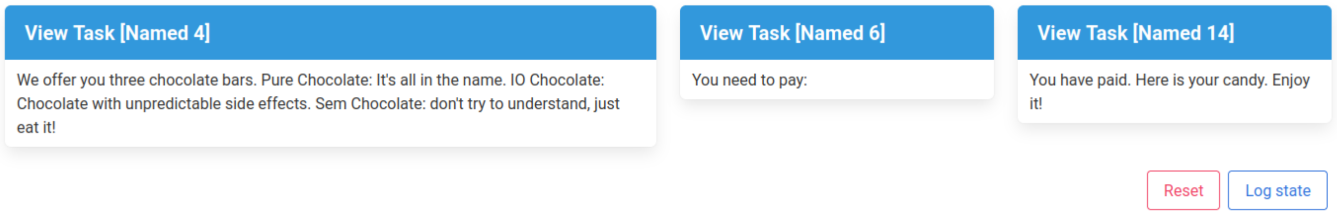
View Task [Named 6]

You need to pay:

Select Task [Named 8]

Choose an option below:

(c) Step 3: Insert a coin


View Task [Named 4]

We offer you three chocolate bars. Pure Chocolate: It's all in the name. IO Chocolate: Chocolate with unpredictable side effects. Sem Chocolate: don't try to understand, just eat it!

View Task [Named 6]

You need to pay:

View Task [Named 14]

You have paid. Here is your candy. Enjoy it!

(d) Step 4: You have paid the bill

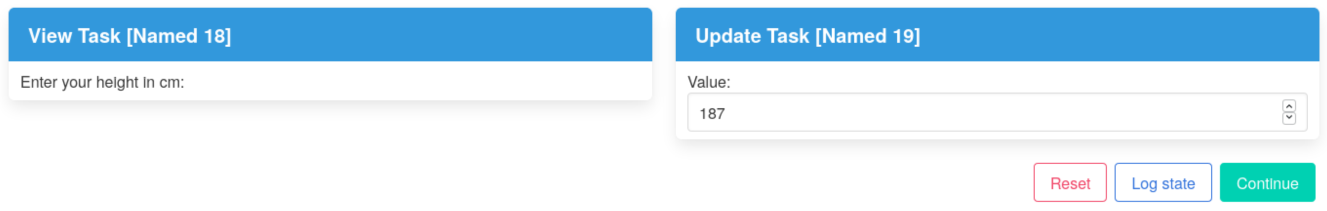
Figure 3: Different stages of the candy vending machine

3.1.3 *Calorie calculator.*

As a final example we created a calorie calculator to demonstrate a more real-world application that incorporates most task types. This application calculates how many calories a person should eat per day in order to maintain their weight. The calculation depends on several factors, such as age, weight, and activity level. The application can be broken down in several steps to prompt the user for input, and finally calculating the result. The implementation of the initial task is given in Listing 6.

(1) When started, the application presents the user with some information about the calculation using a View editor.
(2) After pressing continue, the user is prompted to enter the required data in different steps: height, weight, and age using Enter editors, and gender and activity level using Select editors. Each prompt is wrapped in a Pair task with a View editor on the left side to act as the label. Such a prompt is shown in Figure 4.
(3) In the last step the result is displayed using a View editor.

```
data Gender = Male | Female

data ActivityLevel
    = Sedentary
    | Low
    | Active
    | VeryActive

type Height = Int

type Weight = Int

```



View Task [Named 18]

Enter your height in cm:

Update Task [Named 19]

Value:

187

Figure 4: Prompting the user to enter his/her height

```haskell
type Age = Int

calculateCaloriesTask :: Task h Text
calculateCaloriesTask =
    introduction >>? \_ -> do
        (_, height) <- promptHeight
        (_, weight) <- promptWeight
        (_, age) <- promptAge
        (_, gender) <- promptGender
        (_, activityLevel) <- promptActivityLevel
        let calories = calculateCalories gender
     activityLevel height weight age
        view
        ( "Your resting metabolic rate is: "
            <> display calories
            <> " calories per day."
        )

introduction :: Task h Text
introduction = view <| unlines
    [ "This tool estimates your resting metabolic rate,",
      "i.e. the number of  calories you have to consume",
      "per day to maintain your weight.",
      "Press \"Continue\" to start"
    ]

promptGender :: Task h (Text, Gender)
promptGender =
    view "Select your gender:"
        >< select
            [ "Male" ~> Done Male,
              "Female" ~> Done Female
            ]

promptHeight :: Task h (Text, Height)
promptHeight = view "Enter your height in cm:" >< enter

promptWeight :: Task h (Text, Weight)
promptWeight = view "Enter your weight in kg:" >< enter

promptAge :: Task h (Text, Age)
promptAge = view "Enter your age:" >< enter

promptActivityLevel :: Task h (Text, ActivityLevel)
promptActivityLevel =
    view "What is your activity level?"
        >< select
            [ "Sedentary" ~> Done Sedentary,
              "Low active" ~> Done Low,
              "Active" ~> Done Active,
              "Very Active" ~> Done VeryActive
            ]

-- We omit the actual calculation here since it is a bit
     lengthy.
calculateCalories :: Gender -> ActivityLevel -> Height ->
      Weight -> Age -> Int
calculateCalories gender al h w age = ...
```

Listing 6: Initial task of the calorie calculator (Haskell)

3.1.4 Chat session. We use shared data stores to model a chat session between two users, as displayed in Figure 5. Each user can write messages to the chat history on the left hand side using their respective inputs on the right hand side.

The implementation for this example is given in Listing 7. The function `share` creates a data store that can be accessed by multiple tasks, in this case the two `chat` tasks. The `<<=` operator is used to transform the contents of the shared data store.

```haskell
chatSession :: Reflect h => Task h (Text, ((), ()))
chatSession = do
    history <- share ""
    watch history ><
        (chat "Tim" history >< chat "Nico" history)
    where
    chat :: Text -> Store h Text -> Task h ()
    chat name history = repeat <|
        enter >>* ["Send" ~> append history name]

    append :: Store h Text -> Text -> Text -> Task h ()
    append history name msg = do
        history <<= \h ->
        (if h == "" then h else h ++ "\n")
            ++ name ++ ": '"
            ++ msg ++ "'"
```

Listing 7: A chat Session using shared data stores (Haskell)

3.2 Architecture

Figure 6 shows the architecture of the artefact. The artefact is architecturally separated in two parts: the backend and the frontend. The figure shows the main modules of each part. At the backend we initialize tasks and handle communication with TopHat. At the frontend we render the UI. After a comparative study of existing web server and UI frameworks, we have selected Servant [24] as our webserver and Halogen[7] for the UI. Other options are discussed in the Section 4. We use JSON to establish the communication between frontend and backend. In section 3.2.1 we illustrate the communication between frontend and backend. In section 3.2.2 we explain the working of the backend in detail. The frontend is discussed in section 3.2.3.

3.2.1 Communication between backend and frontend. Figure 7 shows the communication between frontend and backend. The frontend first requests the initial task, which the backend returns using a JSON representation of this task. A user can now interact with the system. In this example, the user updates a value. The frontend sends the input as JSON to the backend, and the backend responds with the updated task. This step can be repeated as necessary. In this

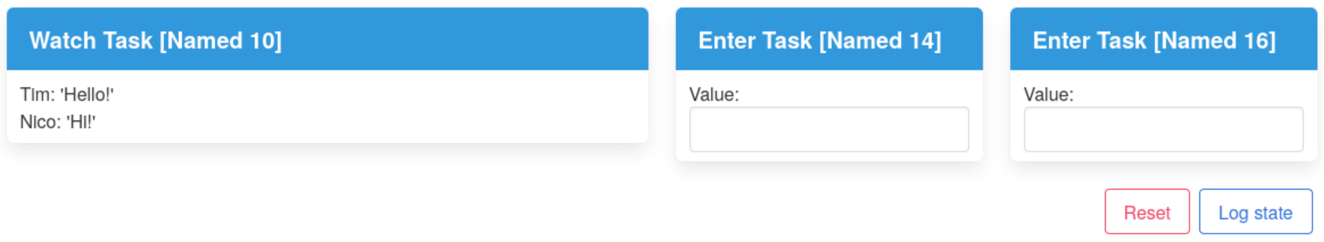


Figure 5: A chat session using shared data stores.

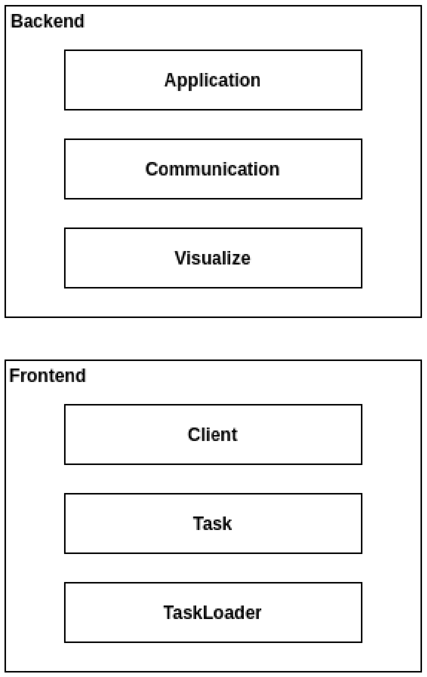


Figure 6: Architecture. Each box represents a main module.

case, the user resets the application, which results in the backend resetting back to the initial task.

The frontend is written in PureScript and the backend in Haskell. We choose JSON as data interchange format, because JSON allows custom data structures, it is easy to use, and both backend and frontend support JSON out-of-the-box.

3.2.2 Backend. The backend is written in Haskell, using Servant [24] as the web server. Servant provides combinators to implement our features, which makes coding less error prone and time-consuming. Servant is up-to-date, well-maintained, well documented and it is easy to get a working prototype.

Servant was developed for practical reasons at Zalora in 2014. Servant is a Domain Specific Language for Haskell that uses APIs to describe the request types, response types and the constraints that are imposed. Web APIs are Haskell types and they are first-class. Developers can check different implementations against these

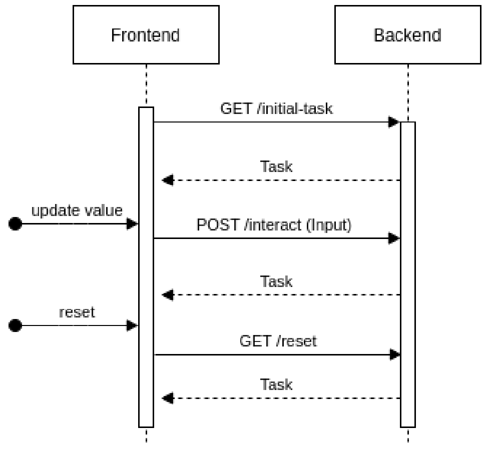


Figure 7: Communication between frontend and backend. Sequence diagram that displays requests (solid arrows) and responses (dashed arrows). `update value` **and** `reset` **are user actions. Task and Input are JSON objects.**

APIs and they use APIs to perform compatibility checks [18]. Servant is developed to create type-safe web applications, clients and documentation. Servant consists of four modules: serving an API, querying an API, JavaScript for querying and documenting an API [24].

The backend consists of three modules:

(1) Application: this module loads the application, sets up the server and defines the handlers (initial tasks, interact, reset and static files).
(2) Communication: this module handles JSON conversion.
(3) Visualize: module to run a web server with a given task. It is possible to load either a development or production environment.

Application module. We create an abstract web Application (WAI-application) in the Application module (see the `application` function in Listing 8). We define the endpoints, the request and the response formats. For example, see the `TaskAPI` in Listing 8. The `server` function provides handlers to serve the initial task, to handle interaction with the frontend and to perform a reset. We bridge the gap to TopHat in `initialiseIO`, `inputsIO` and `interactIO`. We have only added key signatures to Listing 8.

```haskell
module Application (application, State (..)) where

data State h t = State
    { currentTask :: TVar (Task RealWorld t),
      initialised :: Bool,
      originalTask :: Task RealWorld t
    }

type TaskAPI =
    "initial-task"
        :> Get '[JSON] JsonTask
    :<|> "interact"
        :> ReqBody '[JSON] JsonInput :> Post '[JSON]
     JsonTask
    :<|> "reset"
        :> Get '[JSON] JsonTask

type StaticAPI = Get '[HTML] RawHtml :<|> Raw
type API = TaskAPI :<|> StaticAPI

interactIO :: Input Concrete -> Task RealWorld a -> IO (
     Task RealWorld a)
initialiseIO :: Task RealWorld a -> IO (Task RealWorld a)
inputsIO :: Task RealWorld a -> IO (List (Input Dummy))

server :: ToJSON t => State h t -> ServerT API (AppM h t)

application :: ToJSON t => State h t -> Application
```

Listing 8: Application module (Haskell)

Communication module. In Listing 9 we show the core of the communication module. We introduce a new datatype, `JsonTask`, that wraps a TopHat Task to prevent orphaned instances. User input, which is sent back and forth from the client to the server, is defined in `JsonInput`. Both datatypes are provided with instances to convert them from and to JSON.

```haskell
module Communication (JsonTask (..), JsonInput (..))
     where

data JsonTask where
    JsonTask :: ToJSON t
        => Task h t
        -> List (Input Dummy)
        -> JsonTask

instance ToJSON JsonTask

taskToJSON :: Task h t -> Value
nameToJSON :: Name -> Value
editorToJSON :: Editor h t -> Value
inputToJSON :: Input Dummy -> Value

data JsonInput where
    JsonInput :: Input Concrete -> JsonInput

instance FromJSON JsonInput

parseConcrete :: Value -> Parser Concrete
parseName :: Value -> Parser Name
```

Listing 9: Communication module (Haskell)

Visualize module. In Listing 10 we show the signatures of the visualize module. We use this module to run the web server in production (`visualizeTask`) or development (`visualizeTaskDevel`) mode. We differentiate between these modes because we implemented live code reloading for development, which requires a bit of additional setup. Both `visualizeTask` and `visualizeTaskDevel` use the `initApp` function. `InitApp` on its turn invokes the application-function of the Application Module.

```haskell
module Visualize (visualizeTask, visualizeTaskDevel)
     where

initApp :: ToJSON t => Task RealWorld t -> IO Application

visualizeTaskDevel :: ToJSON t => Task RealWorld t -> IO
     ()

visualizeTask :: ToJSON t => Task RealWorld t -> IO ()
```

Listing 10: Visualize module (Haskell)

3.2.3 *Frontend.*

The frontend renders the UI. The code is written in PureScript using the Halogen framework. We have selected PureScript to render the UI of our framework, because PureScript is a featureful language that fits our problem domain. Halogen provides an excellent developer experience, has a component based architecture and PureScript's power and expressiveness.

PureScript [13] is a functional programming language that compiles to JavaScript and shares a lot of similarities with Haskell. Key differences include strict evaluation and extensible records, which ease the translation to JavaScript. Halogen [7] is a declarative UI library for PureScript. It is based on components; a single component is similar to an Elm application, but components can be composed to create complex user interfaces.

The frontend consists of three main modules and some auxiliary modules. In this section we explain the main modules.

(1) Client: module to send requests (initial tasks, interact, reset) to the backend and handle responses.
(2) Task: module to handle JSON encoding and decoding of our domain's datatypes (Task, Editor, Value, Input, and InputDescription).
(3) TaskLoader: module to render the UI.

Client module. The client module is responsible for the communication between frontend and backend. The backend sends a response in JSON that consists of two parts: a Task and a description of possible inputs. We decode this JSON object into a `TaskResponse`. See Listing 11.

```purescript
module App.Client (ApiError, TaskResponse(..),
     getInitialTask, interact, reset) where

data TaskResponse
    = TaskResponse Task (Array InputDescription)

instance decodeJsonTaskResponse :: DecodeJson
     TaskResponse

getInitialTask :: Aff (Either ApiError TaskResponse)

interact :: Input -> Aff (Either ApiError TaskResponse)

reset :: Aff (Either ApiError TaskResponse)
```

Listing 11: Client module (PureScript)

Task module. In the Client module we defined a `TaskResponse`. This `TaskResponse` consists of two parts: a `Task` and an array of `InputDescription`. In the Task module we define the decoding process of `Task` and `InputDescription`. See Listing 12.

```
module App.Task where

data Task
    = Edit Name Editor
    | Pair Task Task
    | Step Task
    | Done
    | Fail

instance showTask :: Show Task

instance decodeJsonTask :: DecodeJson Task

data Input
    = Insert Int Value
    | Option Name String

instance showInput :: Show Input

instance encodeInput :: EncodeJson Input

data InputDescription
    = InsertDescription Int String
    | OptionDescription Name String

instance showInputDescription :: Show InputDescription

instance decodeJsonInputDescription :: DecodeJson
     InputDescription

```

Listing 12: Task module (PureScript)

TaskLoader module. The TaskLoader module renders the user interface (the `render` function in Listing 13). The module also contains logic to handle events (`handleAction`), for example when a user modifies a value. Finally, the `taskLoader` function (see Listing 13) initialises the component.

```
module Component.TaskLoader (taskLoader) where

taskLoader :: forall query input output m. MonadAff m =>
     H.Component query input output m

handleAction :: forall output m. MonadAff m => Action ->
     H.HalogenM State Action Slots output m Unit

render :: forall m. MonadAff m => State -> HH.
     ComponentHTML Action Slots m
```

Listing 13: TaskLoader module (PureScript)

With the above, we have shown how an interactive web UI can be added to TopHat, using existing frameworks. The implementation has been validated by running several example applications.

4 RELATED WORK

Section 2 already discussed related work on TOP, therefore we focus on non-TOP related work in the discussion below.

4.1 Functional Reactive Programming

Functional Reactive programming (FRP) is another approach to UI development using functional programming. FRP is a programming paradigm centered around interactive event-based applications. It has implementations in multiple programming languages, such as Haskell and JavaScript [5]. FRP consists of two main concepts: *behaviors* and *events*. A behavior consists of a value and can be mapped to output, for example a label. Behaviors can depend on other behaviors, so a change in a behavior can propagate through a network of dependent behaviors. An event only occurs at a certain point in time and contains a value. Input is mapped to events, for example the pressing of a key or the position of the mouse cursor. Events can trigger changes in behaviors.

4.2 User interface frameworks

Besides Halogen, many other UI frameworks exist.

Elm [10] refers to both Elm, a functional programming language that compiles to JavasScript [9], and TEA [12], a programming pattern that emerged from it. Elm's ecosystem consists of a large number of available libraries that help in creating web applications.

Miso [15] is a Haskell front-end framework inspired by Elm and Redux. It relies on GHCJS [17], a Haskell-to-JavaScript compiler based on GHC. The main drawbacks of GHCJS are that it is notoriously difficult to install and that it has a limited ecosystem with regards to development tools and IDE integration. The installation process can be largely facilitated by using the Nix package manager [11]. Nix comes with its own sets of problems though, such as a small community and limited documentation

Reflex [32] is an FRP framework written in Haskell with support for a variety of platforms, including the web, desktop, and mobile. Reflex applications are modular, which makes growing and refactoring an application efficient and swift.

4.3 Web servers

We have opted for Servant as our webserver, but other alternatives are available.

In 2010, Michael Snoyman developed the Yesod Web Framework [26]. "Yesod is a Haskell web framework for productive development of type-safe, RESTful, high performance web applications" [27]. The Yesod Web Framework adds the strengths of Haskell (like type safety) to the web. Especially on the boundaries of Yesod and the world, for example a user enters input or persistent data is loaded, Yesod adds mechanisms to define the expected types [25].

In 2011, Michael Snoyman developed SimpleServer to test code of the Yesod Web Framework. Matt Brown added some improvements. The result was a small and fast web server, which was called Warp. Nowadays, Warp is the default web server of the Yesod Web Framework [28]. Warp implements the Web Application Interface (WAI).

5 CONCLUSION

We conclude that it is possible to create an interactive web UI for TopHat programs without resorting to Clean or iTasks. Even though our implementation does not have the full capabilities of the iTasks framework. we show that all the basic requirements for a TOP framework can be implemented. We have support for tasks, shared data stores, combinators and generics. Having said that, we do not yet support the complete feature-set a TOP framework should have.

5.1 Future work

Our prototype framework can be extended in several ways, to cover the complete feature-set of iTasks. We consider these extensions to be interesting pieces of future work.

Datatypes. Currently, only integers, booleans and strings are supported by the prototype. In practice, many other types of data occur, and indeed iTasks supports many more. It would be interesting to support more sophisticated data types, such as lists, coordinates, times and dates.

Multi-user. A workflow management system, and therefore also a TOP system, is not complete without support for multiple users. Working together lies at the heart of the programming paradigm. It therefore speaks for itself that multi-user support would be an interesting and important next step.

REFERENCES

[1] Peter Achten, Pieter Koopman, and Rinus Plasmeijer. 2015. *An Introduction to Task Oriented Programming*. Springer International Publishing, Cham, 187–245. https://www.researchgate.net/profile/Peter_Achten/publication/295505446_An_Introduction_to_Task_Oriented_Programming/links/56e1350d08ae9b93f79c46e5/An-Introduction-to-Task-Oriented-Programming.pdf

[2] Peter Achten, Jurriën Stutterheim, László Domoszlai, and Rinus Plasmeijer. 2014. Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics. ACM, 1–13.

[3] Peter Achten, Jurriën Stutterheim, Bas Lijnse, and Rinus Plasmeijer. 2016. Towards the Layout of Things. ACM, 1–13. https://dl-acm-org.ezproxy.elib11.ub.unimaas.nl/doi/pdf/10.1145/3064899.3064905

[4] Peter Achten, John Van Groningen, and Rinus Plasmeijer. 1993. High Level Apecification of I/O in Functional Languages. In *Functional Programming, Glasgow 1992*. Springer, 1–17. https://repository.ubn.ru.nl/bitstream/handle/2066/111106/111106.pdf?sequence=1

[5] Engineer Bainomugisha, Andoni Carreton, Tom Cutsem, Stijn Mostinckx, and Wolfgang Meuter. 2013. A survey on Reactive Programming. *ACM computing surveys* 45, 4 (2013), 1–34.

[6] TH Brus, Marko CJD van Eekelen, MO Van Leer, and Marinus J Plasmeijer. 1987. Clean—A Language for Functional Graph Rewriting. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 364–384. https://link-springer-com.ezproxy.elib10.ub.unimaas.nl/content/pdf/10.1007%2F3-540-18317-5_20.pdf

[7] Burgess and Honeyman et al. 2021. Halogen. https://github.com/purescript-halogen/purescript-halogen. Version 6.1.2.

[8] Clean 2021. Clean. https://clean.cs.ru.nl. Version 3.0.

[9] Czaplicki. 2012. Elm. https://elm-lang.org. Version 0.19.1.

[10] Czaplicki. 2012. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University* 30 (2012).

[11] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. 2004. Nix: A Safe and Policy-Free System for Software Deployment.. In *LISA*, Vol. 4. 79–92.

[12] Elm Guide 2021. The Elm Architecture. https://guide.elm-lang.org/architecture. Accessed at 2021-07-01.

[13] Freeman and Burgess et al. 2021. PureScript. https://www.purescript.org/. Version 0.13.8.

[14] Mark Gerarts and Marc de Hoog. 2021. Creating Interactive Visualizations of TopHat Programs.

[15] Johnson. 2020. Miso. https://haskell-miso.org. Version 1.7.1.

[16] Bas Lijnse, Jan Martin Jansen, Rinus Plasmeijer, et al. 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. https://www.academia.edu/download/44093349/Incidone_A_Task-Oriented_Incident_Coordi20160325-5075-7woz7i.pdf

[17] Mackenzie et al. 2021. GHCJS. https://github.com/ghcjs/ghcjs. Version 8.6.

[18] Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. 2015. Type-level Web APIs with Servant: an Exercise in Domain-Specific Generic Programming. ACM, 1–12.

[19] Nico Naus, TJ Steenvoorden, et al. 2019. A symbolic execution semantics for TopHat. (2019). https://repository.ubn.ru.nl/bitstream/handle/2066/214654/214654.pdf?sequence=1

[20] N. Naus, T. J. Steenvoorden, A. Byrski, and J. Hughes. 2020. Generating Next Step Hints for Task Oriented Programs Using Symbolic Execution. *Lecture notes in computer science* (2020), 47–68. https://link-springer-com.ezproxy.elib10.ub.unimaas.nl/content/pdf/10.1007%2F978-3-030-57761-2_3.pdf

[21] Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. ACM, 1–12. https://dl-acm-org.ezproxy.elib11.ub.unimaas.nl/doi/pdf/10.1145/3205368.3205375

[22] 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. https://www.academia.edu/download/39518337/ITasks_Executable_specifications_of_inte20151029-18594-mn0lg0.pdf

[23] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. ACM, 195–206. https://repository.ubn.ru.nl/bitstream/handle/2066/103802/103802.pdf

[24] Servant Contributors. 2021. Servant – A Type-Level Web DSL. https://docs.servant.dev/en/stable/index.html. Version 0.18.3.

[25] Snoyman. 2012. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc.

[26] Snoyman. 2020. The Abominable Snoyman). https://www.snoyman.com/.

[27] Snoyman. 2020. Yesod Web Framework. https://www.yesodweb.com/.

[28] M. Snoyman. 2011. Warp: A Haskell Web Server. *IEEE internet computing* 15, 3 (2011), 81–85.

[29] Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A formal foundation for task-oriented programming. ACM, 1–13. https://dl-acm-org.ezproxy.elib11.ub.unimaas.nl/doi/10.1145/3354166.3354182

[30] J. Stutterheim, P. Achten, R. Plasmeijer, V. Zsók, Z. Porkoláb, and Z. Horváth. 2019. Static and Dynamic Visualisations of Monadic Programs. *Lecture notes in computer science* (2019), 341–379. https://link-springer-com.ezproxy.elib11.ub.unimaas.nl/content/pdf/10.1007%2F978-3-030-28346-9_9.pdf

[31] 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, 122–141. https://link.springer.com/chapter/10.1007/978-3-319-14675-1_8

[32] Trinkle et al. 2020. Reflex. https://reflex-frp.org/. Version 0.8.0.0.