

BACHELOR'S THESIS COMPUTING SCIENCE

# Visual Programming with the TopHat Builder

DION BREMER  
s1020299

August 22, 2023

*First supervisor/assessor:*  
Dr. Peter Achten

*Second supervisor:*  
Dr. Tim Steenvoorden

*Second assessor:*  
Dr. Pieter Koopman

Radboud University



## **Abstract**

The TopHat Builder is a program developed by Steenvoorden (2022), which is used to build visual taskflows. These taskflows can be used to facilitate communication between domain experts and software developers. However, the TopHat Builder is still a prototype. This thesis proposes a design of an expansion to the TopHat Builder, together with a partial implementation.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Research questions . . . . .	4
1.2	Structure . . . . .	4
<b>2</b>	<b>Task Oriented Programming</b>	<b>5</b>
2.1	Visualisation . . . . .	6
<b>3</b>	<b>TopHat</b>	<b>9</b>
3.1	Editors . . . . .	9
3.2	Tasks . . . . .	9
<b>4</b>	<b>TopHat Builder</b>	<b>11</b>
4.1	Visual TopHat . . . . .	11
4.1.1	Restricted tasks . . . . .	11
4.1.2	Transformations . . . . .	13
4.2	PureScript with Concur . . . . .	15
4.3	Technical overview . . . . .	16
4.4	Overview of current features . . . . .	17
<b>5</b>	<b>New Features</b>	<b>20</b>
5.1	Overview of new features . . . . .	20
5.2	Design principles . . . . .	21
5.3	Side-menu . . . . .	22
5.4	Hover menu . . . . .	23
5.5	Storyboards . . . . .	23
5.5.1	Switching taskflows . . . . .	24
5.5.2	Creating new taskflows . . . . .	26
5.5.3	Making a selection . . . . .	28
5.5.4	Removing a task . . . . .	30
5.5.5	Swapping parallel branches . . . . .	32
5.5.6	Abstracting tasks . . . . .	34
5.6	Exporting tasks . . . . .	37
5.7	Importing tasks . . . . .	38

<b>6</b>	<b>Implementation</b>	<b>40</b>
6.1	Task implementation . . . . .	40
6.2	Allowed selections . . . . .	41
6.3	Abstractor implementation . . . . .	43
<b>7</b>	<b>Related Work</b>	<b>45</b>
7.1	Visualisation of functional languages . . . . .	45
7.2	Programming with holes . . . . .	45
<b>8</b>	<b>Conclusions</b>	<b>47</b>
8.1	Future work . . . . .	47

# Chapter 1

## Introduction

Software has become a building block of modern society. Everyone uses software in one way or another, but the creation of new software may lead to problems: domain experts do not know how to program, and software engineers know very little about the domain.

In agile software development, the domain expert often discusses their requirements for the software, and the developer shows the progress made on the software. Here is where miscommunications may occur: how can the developer and domain expert understand each other, while they know very little about each others area of expertise?

The workflow community attempts to solve this issue by formally defining graphical workflows, which are more intuitive to the layman. This experience with graphical workflows can now be applied to the *Task Oriented Programming* (TOP) paradigm. TOP models workflows directly, by combining *tasks* in different ways to create *taskflows*. What sets TOP apart from other workflow languages is that its model and implementations are embedded in functional languages. This combination has been visualised before (Henrix et al., 2012; Stutterheim, 2017; Mol, 2020), and more recently by Steenvoorden (2022).

The latter of these visualisation tools is the *TopHat Builder* (Steenvoorden, 2022), which is an interactive visual tool which can be used to build taskflows. The tool allows a user to build taskflows in the *TopHat* formalism, which is a model for TOP. The TopHat Builder is very structured, and has a clean interface. However, at it stands, it is more of a proof-of-concept than a fully functional tool. How can the tool be of more use to an end user?

## 1.1 Research questions

This leads us to the main research question of this thesis:

How can the TopHat Builder be expanded, in order to offer more functionality to the user?

The result of the research is a design and partial implementation of several new features. The research question can be split up into the following sub-questions:

- Which design principles should be used in the expansion?
- How should the user access the new features?

The rest of this thesis will aim to answer these questions.

## 1.2 Structure

Firstly, the concepts of TOP (chapter 2) and TopHat (chapter 3) are explained more thoroughly. After this, the TopHat Builder is introduced in chapter 4. The design for the new features is elaborated upon in chapter 5, followed by the implementation of one of these features in chapter 6. After this, related work is provided (chapter 7), followed by conclusion and future work (chapter 8).

## Chapter 2

# Task Oriented Programming

Task Oriented Programming (TOP for short) is a programming paradigm that focuses on *tasks* as its main unit of work. This means that tasks in task-oriented programming are analogous to objects in object-oriented programming, or functions in functional programming. TOP implementations are embedded in functional languages.

Tasks model workflows that one would encounter in daily life, or in business. These workflows are then called *taskflows*. Consider the following scenario:

I would like to go grocery shopping, and then make dinner. First I go to the supermarket and buy steak and pudding. Then I go home to prepare the steak. After finishing the steak, I can either have pudding or have coffee instead.

This scenario illustrates the so-called *combinators* of TOP: *sequential* execution, *parallel-and* execution and *parallel-or*. In the example, the sentence “first I go to the supermarket and then I prepare the steak” would be modeled using a sequential combinator. The sentence “I buy steak and pudding” is an example of a parallel-and combinator, and “I can have pudding or coffee” would be a parallel-or. Note that there is a slight misuse of the word parallel: parallel only signifies that the tasks can be executed in any order, not necessarily in parallel. The difference between parallel-and and parallel-or is intuitive: tasks in a parallel-and must both finish before moving on, while only one of the tasks in a parallel-or has to finish.

An important feature of tasks is that tasks can be interactive: a (human or machine) user can influence the value of a task. This value can be observed, and can be changed in the future. Editing these values can influence all future tasks that need to be performed.

## 2.1 Visualisation

An important aspect of taskflows is that they can be visualised. Since images are generally easier to interpret than code, these visualisations can be used for communication between software engineers and stakeholders. The first attempt at visualising taskflows was the *GiN* system (Henrix et al., 2012). *GiN* was built-in to the Clean compiler, in the preprocessing stage, and it visualised the iTask implementation of TOP. Using *GiN* resulted in a *GiN* diagram, as can be seen in figure 2.1. The *GiN* system also contained a tool to visually edit workflows. This tool, as shown in figure 2.2, was most useful to edit existing diagrams on the spot. It required an existing taskflow as input, which could then be opened in the editor.

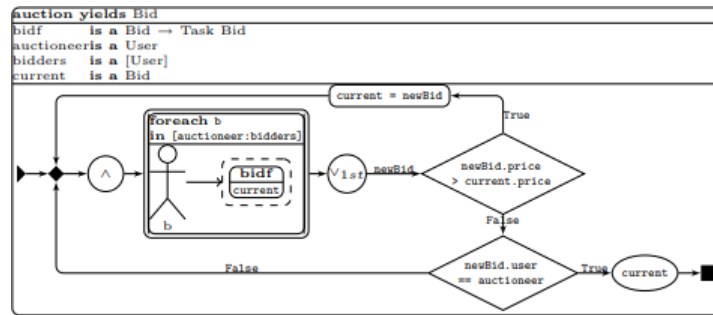


Figure 2.1: An example of a *GiN* diagram. (Henrix et al., 2012)

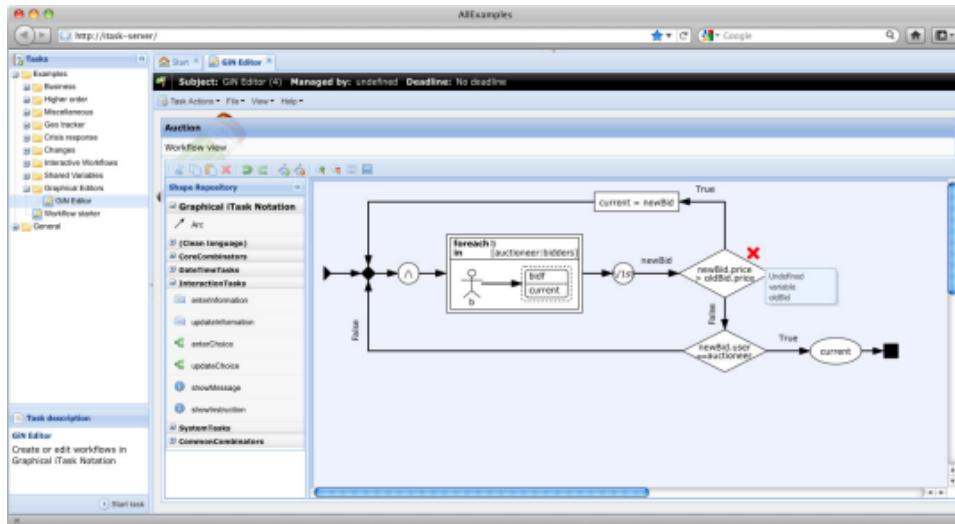


Figure 2.2: The visual *GiN* tool. (Henrix et al., 2012)

Another attempt was the *Tonic* system (Stutterheim, 2017). *Tonic* was used to generate *blueprints* from an iTask program. Like *GiN*, it was also



built into the Clean compiler, as both the compiler and Tonic partly used the same functionality. The Tonic stage was optional, and when activated, generated a blueprint as in figure 2.3. Interestingly, the blueprints can be set to be visible during execution of the taskflow, highlighting which stage the execution is at.

The issue with these blueprints is that they cannot be edited once generated. A more general issue with both GiN and Tonic is that these systems are built into the Clean compiler. This is a problem since the compiler keeps getting updated, while little work is done to keep the visualisations up-to-date. Because of this, a stand-alone program may be more useful in making visualisations.

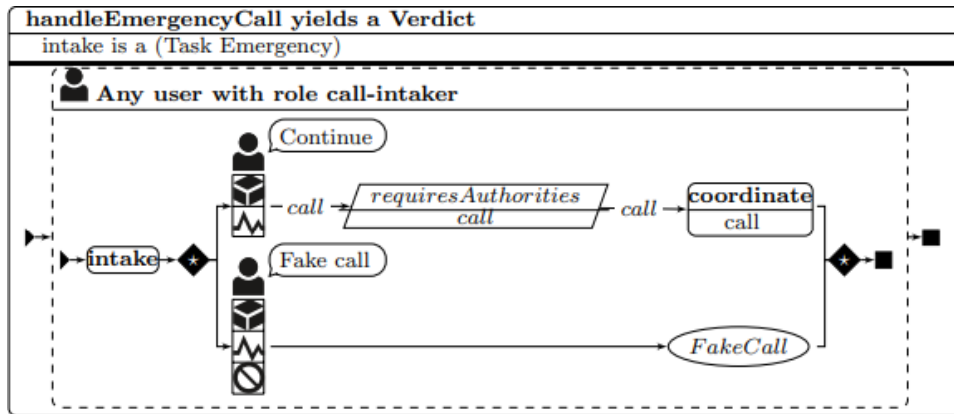


Figure 2.3: A blueprint generated by Tonic. (Stutterheim, 2017)

The *Mojito* system (Mol, 2020) is a stand-alone visualisation tool for iTask. Mojito’s strength lies in the fact that it allows a user to build taskflows from scratch, as opposed to the GiN and Tonic systems. The program consists of a grid where the user can place tasks. These tasks can then be connected by arrows, which represent steps. When the user is done with the current taskflow, they can abstract the current taskflow in order to use it as a task in a future taskflow. The program even has a type editor, where the user can create new types.

In my opinion, the main issue with Mojito is the freedom that it offers. It allows the user to place tasks randomly on the grid and connect them all together, creating a cluttered taskflow. Since visualisations of taskflows should be intuitively clear to anyone, it should not be possible for the user to clutter the taskflow.

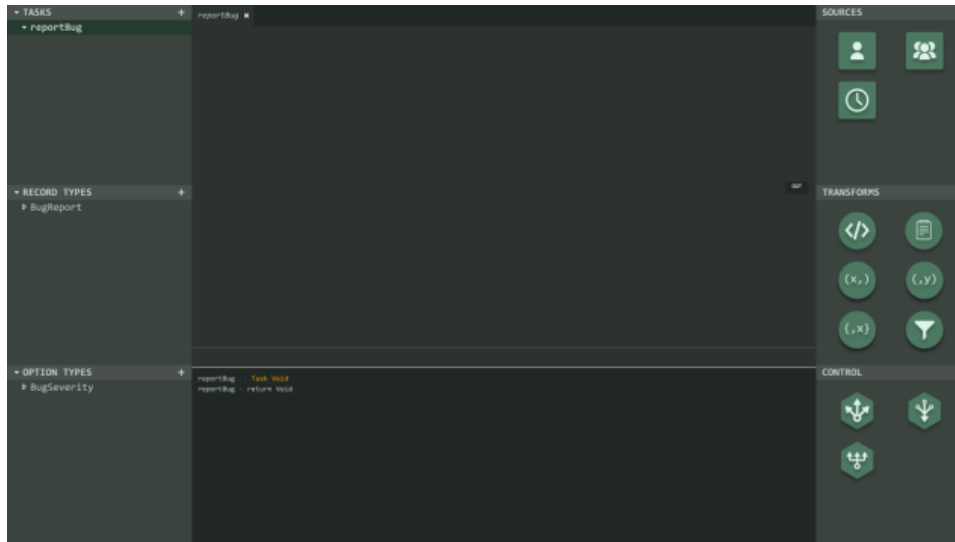


Figure 2.4: The interface of the Mojito system. On the left there are the taskflows and the types, on the right there are the available editors and combinators. (Mol, 2020)

Ideally, one would like to have a program that can build taskflows, but in a structured manner. To this end, Steenvoorden (2022) developed the *TopHat Builder*, which is discussed in chapter 4.

## Chapter 3

# TopHat

As mentioned before, Steenvoorden (2022) developed a *TopHat Builder* in order to visualise taskflows. But what is TopHat? In short, TopHat is a formalism to describe TOP formally. It formalises notions such as tasks, combinators and values of a task. Note that TopHat also relies on a functional host language.

### 3.1 Editors

As mentioned before, tasks can be interactive. In TopHat, this interaction is modeled by so-called *editors*. TopHat defines 5 types of editors (Steenvoorden, 2022): *unvalued* (notation:  $\square$ ), *valued* ( $\boxminus$ ), *shared* ( $\boxplus$ ), *valued read-only* ( $\boxtimes$ ) and *shared read-only* ( $\boxdot$ ). The editors have an intuitive meaning: in an unvalued editor, a user can enter a new value. In a valued editor, the user can update an existing value, or transition to an unvalued editor. A shared editor is similar to the valued editor, but the value may be shared by several of these editors and other task functions. Lastly, there are read-only variants of the valued and shared editors.

### 3.2 Tasks

These editors are only one type of task. TopHat also defines the tasks *transform* (notation:  $\bullet$ ), *pair* ( $\blacktriangleright\blacktriangleleft$ ), *lift* ( $\blacksquare$ ), *choice* ( $\blacklozenge$ ), *fail* ( $\blacklozenge$ ), *step* ( $\blacktriangleright$ ), *share* (**share**) and *assign* ( $:=$ ). Table 3.1 presents the meanings of these tasks. These tasks, together with the editors, are all the possible tasks in TopHat.

Using the symbols from table 3.1, we can re-write the dinner example in a more formal way:

Task	Notation	Meaning
Transform	$f \bullet t$	apply function $f$ to the value of task $t$
Pair	$t_1 \blacktriangleright t_2$	execute task $t_1$ and task $t_2$ in parallel
Lift	$\blacksquare e$	lift expression $e$ into the task type
Choice	$t_1 \blacklozenge t_2$	execute $t_1$ or $t_2$
Fail	$\zeta$	$\zeta$ is a task that never receives input and never holds a value
Step	$t \blacktriangleright e$	execute task $t$ and return its value to expression $e$ , which computes the next task to perform
Share	<b>share</b> $b$	create a new shared reference to value $b$
Assign	$h \coloneqq e$	assign expression $e$ to heap location $h$

Table 3.1: The meanings of the different tasks. Note the implementation of the aforementioned sequential, parallel-and and parallel-or combinators as step, pair and choice. The functions and expressions used in the table come from a host language.

**let** *haveDinner* = *goToSupermarket*  $\blacktriangleright$   $\lambda\_.$  (*buySteak*  $\blacktriangleright$  *buyPudding*)  $\blacktriangleright$   
 $\lambda$  (*steak*, *pudding*). *prepareSteak*  $\blacktriangleright$   $\lambda$  *preparedSteak*. *eat preparedSteak*  
 $\blacktriangleright$   $\lambda\_.$  (*eat pudding*  $\blacklozenge$  *drinkCoffee*) (3.1)

As a final remark, note that the tasks in table 3.1 rely a lot on the host language. This gives the language a lot of expressive power. For example, a task such as

$\square \text{Int} \blacktriangleright \lambda n. \text{if } n \text{ 'mod' } 2 == 0 \text{ then } \square \text{“input is even”} \text{ else } \square \text{“input is odd”}$   
(3.2)

is entirely valid. This task asks the user to input an integer, and shows whether it is even or odd. It relies on the host language to evaluate the lambda expression on the right-hand side, as well as handling the Int and String types used.

Like iTask, taskflows in TopHat can be visualised. The tool to visualise these taskflows is discussed in the next chapter.

## Chapter 4

# TopHat Builder

### 4.1 Visual TopHat

Where the GiN, Tonic and Mojito systems were used to visualise (and build, in the case of GiN and Mojito) iTask taskflows, the TopHat Builder<sup>1</sup> is used to build TopHat taskflows. Figure 4.1 shows the interface of the Builder. Contrary to Mojito, the Builder always places parallel tasks horizontally, and sequential tasks vertically. Where GiN and Tonic were built-in to the Clean compiler, TopHat is a standalone web program. Another advantage compared to GiN and Tonic is that the TopHat Builder can be used to build taskflows from the ground up, instead of visualising or editing existing taskflows.

#### 4.1.1 Restricted tasks

Since it is notoriously difficult to visualise functional languages, the TopHat Builder uses a special set of tasks called *restricted tasks* (Steenvoorden, 2022). These restricted tasks are a subset of the previously defined tasks, and have some of the functionality of the host language built-in. This removes the need to visualise the entire host language. The restricted tasks consist of the editors, pair, lift, choice, share and assign tasks as described previously, as well as the tasks in table 4.1. It is noteworthy that there are four kinds of steps in the restricted tasks: guarded/unguarded steps, and guarded/unguarded selects. These steps do not rely on evaluating expressions on the right-hand side of the step, so that there is no more need for a host language to evaluate these.

Note that there are several concepts that are usually performed by the host language: pattern matching, lambda expressions and records.

The restricted tasks also make use of *records* in its definition. These

---

<sup>1</sup>For a demo of the TopHat Builder by Steenvoorden (2022), see <https://timjs.github.io/tophat/builder.html>

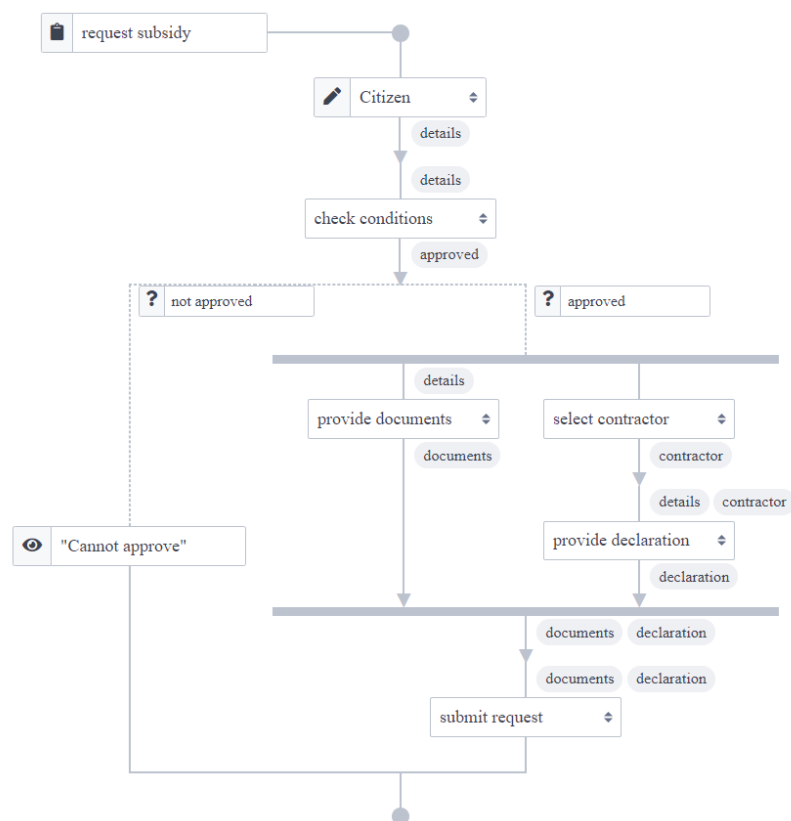


Figure 4.1: The interface of the TopHat Builder, which shows the taskflow that is being worked on.

records are similar to those found in functional programming languages: they are sets of labeled expressions. In the case of restricted tasks, records are used as arguments for an *execute* task. These execute tasks can be seen as predefined tasks: they can take in arguments, and produce a result.

Lastly, the restricted tasks contain a “special” execute task: a *hole*. Holes are tasks that do not yet have a meaning. They are used in the Builder in places where a task is not yet known, for example when a new task is placed in the taskflow. Holes always give the user an error. Holes can be turned into specific execute tasks, see section 4.1.2.

Task	Notation	Meaning
Unguarded step	$r_1 \blacktriangleright \lambda m. r_2$	Execute $r_1$ and then execute $r_2$
Guarded step	$r_1 \blacktriangleright \lambda m. < \overline{g_i \mapsto r_i}; >$	Execute $r_1$ , and then execute $r_i$ only if guard $g_i$ is true
Unguarded select	$r_1 \triangleright \lambda m. r_2$	Execute $r_1$ , and after user input, execute $r_2$
Guarded select	$r_1 \triangleright \lambda m. < \overline{l \mid g_l \mapsto r_l}; >$	Execute $r_1$ , and after user input, execute $r_l$ if guard $g_l$ is true
Execute	$\chi \{ \overline{l = e_l} \}$	Execute task $\chi$ with arguments $\{ \overline{l = e_l} \}$

Table 4.1: An overview of the additional restricted tasks. The symbol  $m$  refers to a match,  $g$  to a guard,  $l$  to a label,  $e$  to an expression and  $r$  to a restricted task.

#### 4.1.2 Transformations

To build a taskflow from scratch, the user should start with a minimal taskflow and apply certain *transformations* to this taskflow. These transformations are (small) actions that alter the taskflow, so that the user can build a taskflow step-by-step. Steenvoorden (2022) describes several transformations that should be possible in order to build a complete taskflow. Figure 4.2 lists these transformations. Note that the tasks in these transformations are restricted tasks (with symbol  $r$ ), as described in section 4.1.1. Some of these transformations have already been implemented: selecting/unselecting steps, inserting holes, filling holes/clearing executes, choosing/combining parallel branches, and forking parallel branches (but only whenever a parallel branch already exists).

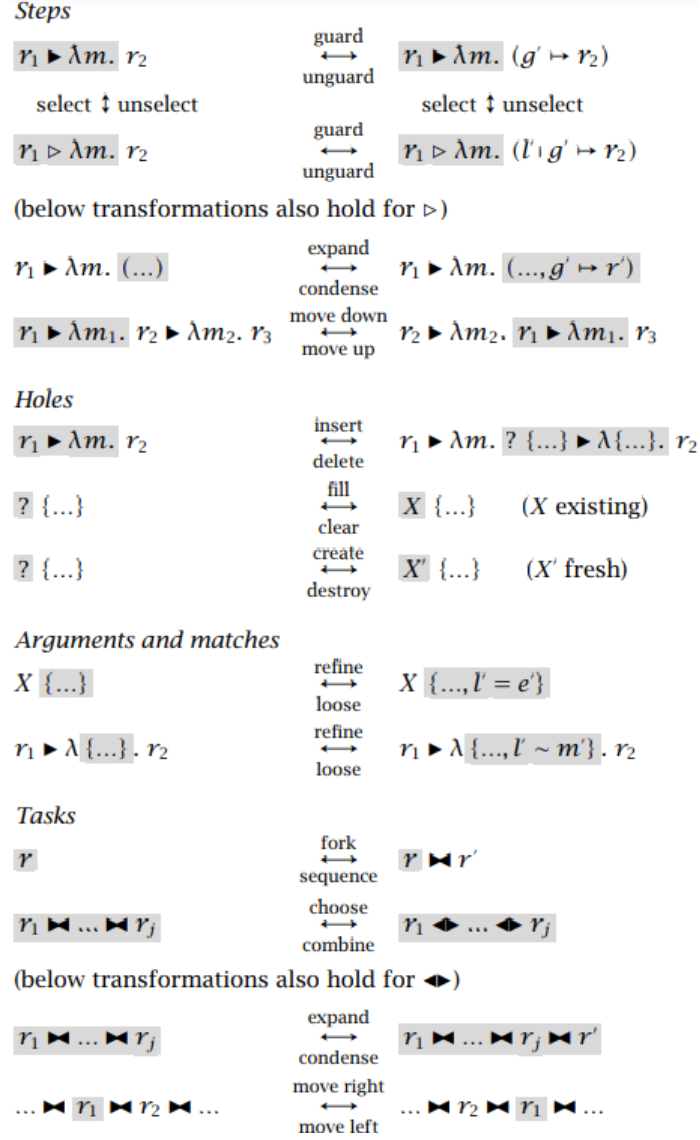


Figure 4.2: The transformations that the user should be able to perform in the Builder. Figure from (Steenvoorden, 2022).



## 4.2 PureScript with Concur

The TopHat Builder is implemented in the PureScript language<sup>2</sup>, using the Concur<sup>3</sup> library. Purescript is a strongly typed functional programming language that compiles to JavaScript. It can be used to build web applications, such as the TopHat Builder. In syntax, it is very similar to the Haskell language, but there are a few notable differences<sup>4</sup>. Under the hood, PureScript uses eager evaluation whereas Haskell (famously) uses lazy evaluation. A PureScript program consists of *modules*, that each contain different functionality.

In order to build a user interface, the Builder relies on the Concur library. Concur introduces *Widgets* to the Purescript environment. Widgets are elements that are rendered to the screen, such as pieces of text, images, icons, buttons, and everything else that is necessary to build graphical user interfaces.

These widgets are implemented as monads. The idea behind monads is that they can be composed, but what does the composition of widgets mean? In Concur, widgets are composed *in time*. Take for example widget A, a button, and widget B, a piece of text. Composing these widgets will result in a new widget AB. At first glance, widget AB looks exactly like widget A, but when clicked, transforms into widget B. Handling events in Concur is therefore easily implemented by composing several widgets together, as demonstrated below. As a side note, the implementation<sup>5</sup> uses a special set of imports and definitions, dubbed the **Preload**. This **Preload** can also be found in the implementation, together with several useful predefined widgets.

---

<sup>2</sup><https://www.purescript.org/>

<sup>3</sup><https://github.com/purescript-concur/purescript-concur-core>

<sup>4</sup><https://github.com/purescript/documentation/blob/master/language/Differences-from-Haskell.md>

<sup>5</sup>The implementation can be found on <https://gitlab.science.ru.nl/dbremer/tophat-visual>.

```

import Preload

import Concur.Dom (Widget)
import Concur.Dom.Input as Input
import Concur.Dom.Style as Style
import Concur.Dom.Text as Text

helloText :: forall a. Widget a
helloText = Text.text "Hello_World"

helloButton :: Widget Unit
helloButton = Input.button Style.Default Style.Secondary
               Style.Large "Press_me"

helloWorld :: forall a. Widget a
helloWorld = do
    helloButton
    helloText

```



(a) Before pressing the button, the button says “Press me”.

Hello World

(b) After pressing the button, the button disappears and turns into the text “Hello World”.

Figure 4.3: The result of composing widgets in Concur.

## 4.3 Technical overview

The TopHat Builder consists of several modules, which I shall briefly discuss. It follows the model-view paradigm, where the logic of the program is separate from how the program is rendered to the user.

Firstly, there is the *Syntax* module, which contains a model for TopHat: this includes tasks, matches, expressions and all that is necessary to build TopHat programs. Similarly, there is a *Types* module which contains all of the necessary types. All of the tasks and types that the program uses are stored in a *World* record. The World consists of three hashmaps: a *Typetext* which maps type names to basic types, a *Context* which maps term names to (not so basic) types and a *Tasktext* which maps names to tasks, with associated parameters.

Secondly, there is a type checker. Every task has a type: they take in some (or no) arguments, and may produce a value. The type checker turns *unchecked* tasks into *checked* tasks. An unchecked task is the most basic task(flow) in the program: it has no additional information, besides the

task information. In contrast, checked tasks are annotated with a status: a checked task can have status *Success*, *Failure* and *Unknown*. These statuses indicate whether a task has the correct type or not (or in the worst case, it indicates that it is unknown whether the type is correct). If a task receives the **Success** status, it also receives the **Context** of the program, to let it know which other tasks are available. In case of a **Failure** status, the task receives an error message that it can display to the user. In case of **Unknown**, the task receives no further information.

Lastly, the *Renderer* module takes care of the main program loop, and makes sure that checked tasks are rendered to the screen. The loop begins with a given name and world. It looks up the name in the world, which returns a checked task to render. Since the user may have edited this checked task, it runs the task through the checker again. Now that the renderer has an up-to-date checked task, it can render the task to the screen. It does so in two ways: the actual builder, and a pretty print of the task. The former is interactive, and the user can use this to update the current task. The renderer responds by updating the world so that the next iteration shows the newest taskflow. This automatically saves the task, so that the task is always up-to-date. Note that the renderer could render unchecked tasks as well, but this would not be very helpful to the user: unchecked tasks do not contain the annotation, and so the user would not be able to fill holes.

## 4.4 Overview of current features

The TopHat Builder is used to build taskflows visually. The interface of the builder contains an interactive representation of a taskflow, then a pretty-print of the taskflow, followed by several notes on how to use the program.

In contrast to Mojito, the TopHat Builder does not allow the user to place tasks anywhere on a grid. Instead, new tasks are added by double-clicking the triangles in-between existing tasks. These triangles are the visualisation of steps. Double-clicking adds a new task under this triangle, so that steps are always rendered top-to-bottom. In contrast, parallel branches are rendered left-to-right, so that it is immediately clear whether tasks are in series or in parallel. Removing a task from the taskflow is not yet supported.

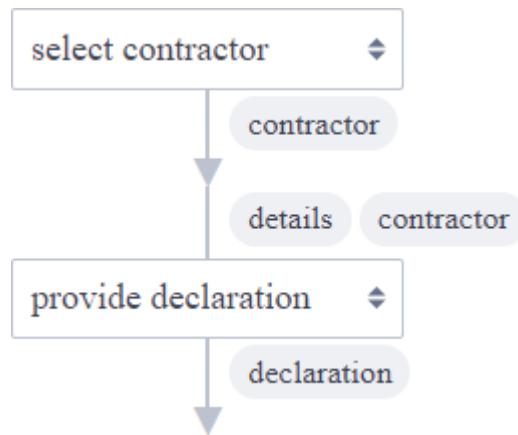


Figure 4.4: The visualisation of a step in the TopHat Builder. This step steps from “select contractor” to “provide declaration”.

So far, parallel branches support two operations: single-clicking changes the branch from pair to choice (see figure 4.5) and vice-versa, and double-clicking adds another parallel branch. These parallel branches can then again be extended by double-clicking the triangles that are created when adding another branch.

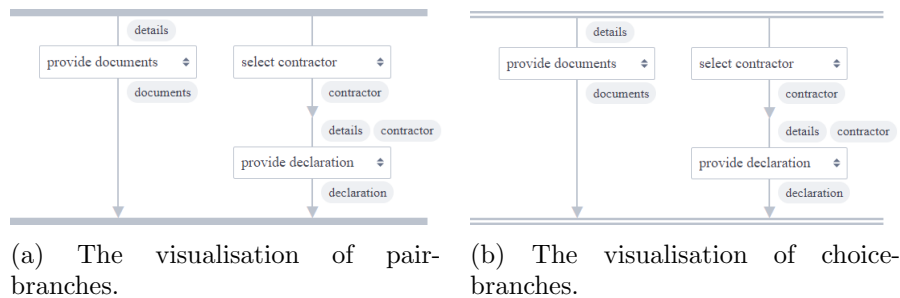


Figure 4.5: The difference in visualisation of pair and choice.

When adding another task to the taskflow, the task appears as a *hole*: it simply says “??” instead of the name of a task. In order to change the task to its proper name, the user can click the task, and a list of all possible tasks in the program appears. The user can select the task that it needs from here. There is no way to add a completely new task to the program as of yet.



Figure 4.6: The visualisation of a hole. The orange line around the hole indicates that the typing is wrong.

In the builder, each task has a type. These types are checked by a type checker. If the user adds a task whose type does not fit in the location that it is in, the outline of the task turns orange (see figure 4.6). Hovering over this wrongly placed task provides an error message. The user can determine which type a task has by looking at the rounded boxes. The boxes under a task show the result of a task, whereas boxes above a task show the arguments of that task.

All throughout the program, the Builder lets the user know when something can be interacted with by highlighting certain parts or showing text when hovering over something. This is an important design principle that should be kept in mind when extending the program.

## Chapter 5

# New Features

### 5.1 Overview of new features

The TopHat Builder already has useful features to work on an existing taskflow. The next step to turn the Builder into a proper tool is to add several features that one would expect from such a tool. The selection of features that I will design are the following:

- A tool for extracting parts of a task into a new task
- A button to export the project
- A button that imports the project
- A way to swap parallel branches

Here, the term *project* means the entire `World` object as discussed in section 4.3.

These desired new features can be divided into several sub-features. Especially the extracting tool (which I shall call the abstractor from now on) relies on many sub-features. For starters, it needs to be possible to create new taskflows, so that the abstractor has a place to put the abstracted taskflow. The user must be able to create these taskflows themselves, and must be able to switch between taskflows. Next, to abstract part of a taskflow, the user must be able to select several tasks in a taskflow. The user needs some way of telling which tasks have been selected, and which have not. Finally, it must be possible to remove tasks from a taskflow. When tasks can be removed, the abstractor can insert a new task in place of the selection.

The other new features are a lot smaller in scope. These new features and their dependencies are shown below.

1. A tool for extracting parts of a task into a new task
  - 1.1 A way to create new tasks
    - 1.1.1 A button that creates a new task
    - 1.1.2 A way to switch between tasks
      - 1.1.2.1 A menu to select a task to work on
  - 1.2 A tool to select several tasks in the taskflow
    - 1.2.1 A visual way of showing selected items
  - 1.3 A way to remove several sub-tasks from a larger task
    - 1.3.1 A way to remove a single sub-task from a larger task
      - 1.3.1.1 A delete button for each item that can be deleted
2. A button that exports project to a file
  - 2.1 A way to store a project in a file
    - 2.1.1 A way to serialize a taskflow
3. A button that loads a project from a file
  - 3.1 A way to load a project in a file
    - 3.1.1 A way to deserialize a taskflow
4. A way to swap parallel branches

Before designing these features, it is wise to consider some design principles to adhere to, as discussed in the next section.

## 5.2 Design principles

The design for these new features is guarded by a few general design principles. Since the visual taskflow should be easily understandable by both programmers and domain experts, the interface of the Builder must remain clean: the taskflow should be clearly visible, without any buttons, options, or other non-relevant items obstructing its view. Any buttons that are added should be placed to the side of the taskflow, where they do not clutter the interface.

Another important factor to keep the interface clean are the colours used. These colours should be easy on the eyes, and have a specific purpose: a blue element indicates that an action can be performed, orange indicates an error, and grey is the default colour. It is important that grey is the default, as it contrasts nicely with the blue and orange colours. This way, it is immediately clear to the user when something requires their attention.

The final key principle used in the design is *hovering*. Steenvoorden (2022) already used this in the original implementation: hovering over a

task turns it blue, hovering over a pair or choice turns its bars blue, hovering over an orange (erroneous) task shows the error message and hovering over a variable allows the user to change it. This is in line with the first principle of keeping the interface clean: options or operations only appear when the user hovers over them, so that they do not clutter the interface.

### 5.3 Side-menu

The menu needs to consist of two parts: one part where the (newly) created taskflows are presented, and another part that contains four buttons. These buttons are the “New”-button, the “Import”-button, the “Export”-button and the “Abstract”-button. Three of these buttons have in common that they create a new taskflow. It is therefore best to place these three buttons together. The “Export” button does not create a new taskflow, but is inseparable from the “Import” button, hence it also belongs in this list.

In my design, I decided to place both parts of the menu on the left-hand side. Placing the menu to the right of the Builder felt unnatural to me. A design where the buttons of the menu were above the current taskflow and the stored taskflows were on the side, would have also been possible. I have chosen, however, to place the two parts on the same side. I figured that it would be intuitive to have the user select taskflows in the same place as where the user creates taskflows. Therefore, the buttons are located on the left.

One may argue that a menu above the taskflow would also be possible: the Builder is laid out from top to bottom, after all. I decided against this, as the list of stored taskflows expands whenever the user creates a new taskflow. If this list is stored above the taskflow, it would have to expand left-to-right, in order to not obstruct the interface. This expansion, however, feels strange. This may be due to the fact that many other programs have a fixed menu on the top, not an expanding one. An expanding menu on the side does not feel alien, as other programs tend to do the same.

A final question must be answered about the menu: what happens when the user creates many taskflows, and the menu overflows? If this were the case, the user would not be able to use the menu effectively, since there may be buttons that are off-screen. To prevent this, the menu should stop expanding after five or more taskflows. When a project contains more than five taskflows, the menu no longer expands, but a scroll bar is added to the side of the taskflows. The user can now scroll through their taskflows in order to find the one that they would like to work on. Another advantage of this approach is that the “create” buttons are always in the same place, so that they can be easily accessed.



## 5.4 Hover menu

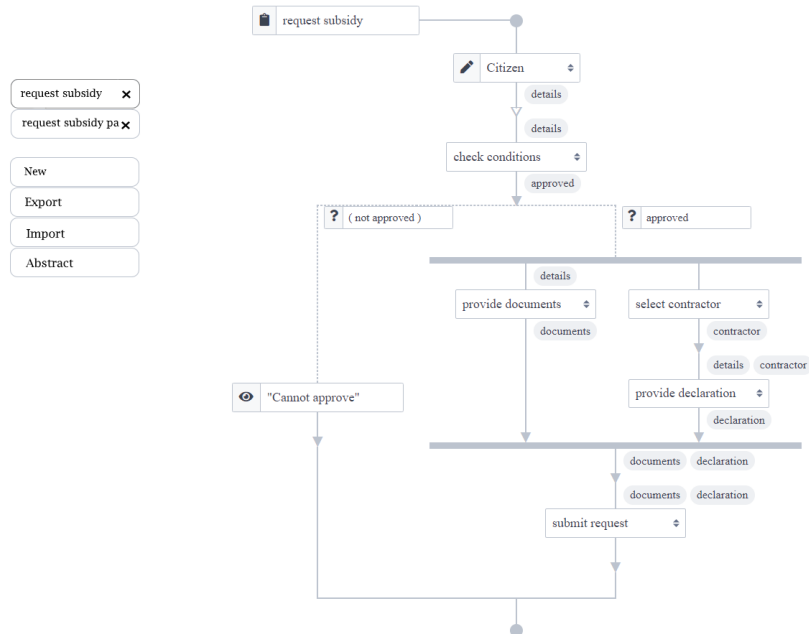
Since the side menu only supports “large” operations, another menu is needed for operations on individual tasks. Keeping clarity in mind, I designed a the *hover menu*. This menu pops up whenever the user hovers over a task, and reveals several operations that the user can apply to that specific task. For now, this menu supports deleting a task and selecting a task. The menu is designed in a way that is flexible, so that in the future other operations may be added to the list. It can also behave differently for different tasks. For example, hovering over a pair task will reveal arrows that the user can use to swap parallel branches.

The hover menu is an important addition, as it is the most logical place to implement the transformations described in section 4.1.2. These transformations apply to only one task, so it is only logical that they are somehow tied to individual tasks.

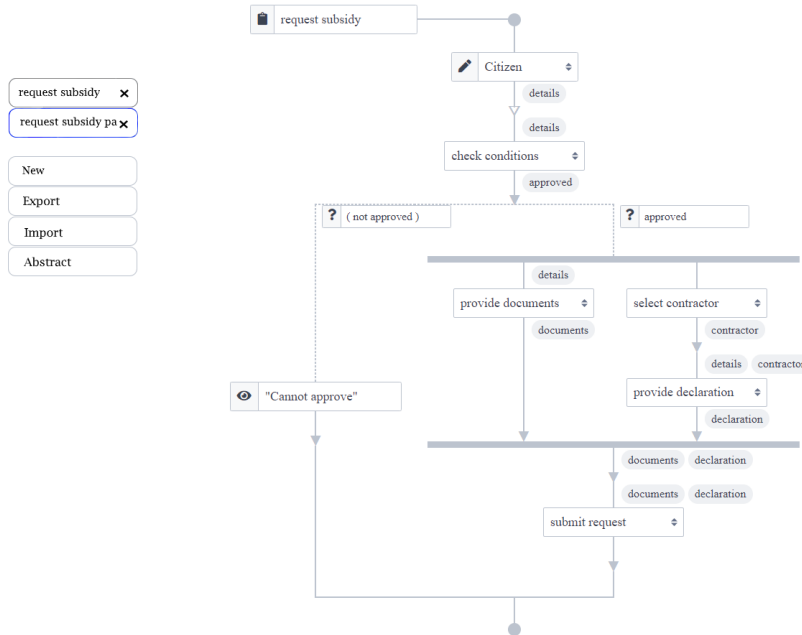
## 5.5 Storyboards

The design as discussed in the previous sections is visualised in so-called *storyboards*. These storyboards present several user stories, which show how a user might use the new features of the Builder.

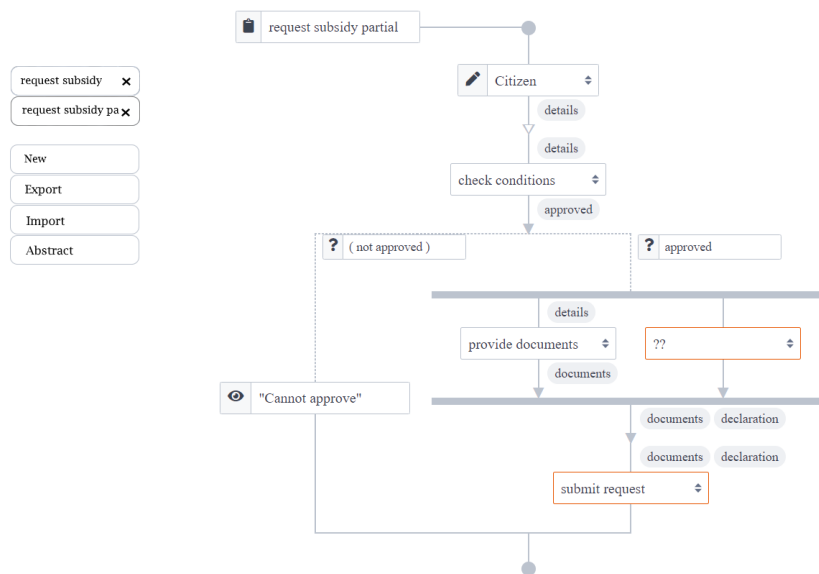
### 5.5.1 Switching taskflows



(a) The user is working on task “request subsidy”, but would like to work on task “request subsidy partial” instead.

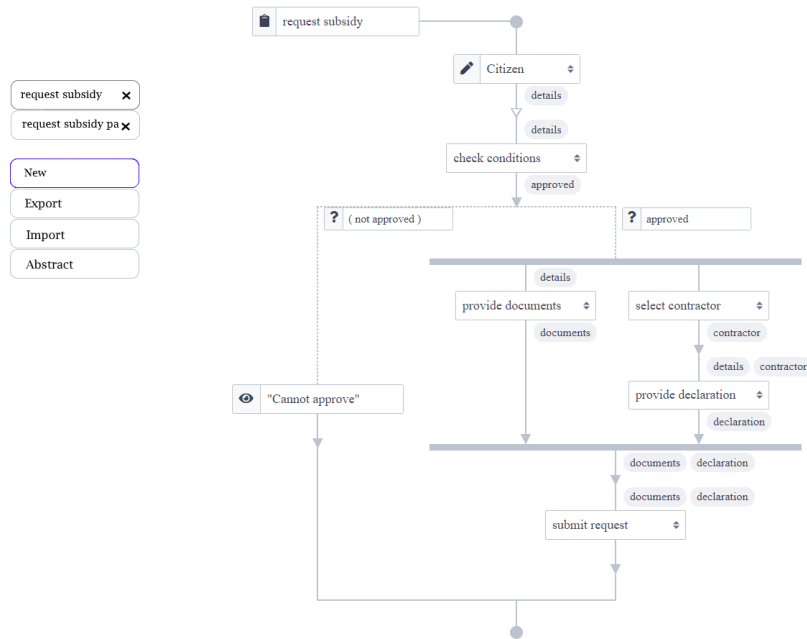


(b) The user locates the task in the menu on the side, and clicks this. Any changes that the user made to “request subsidy” are automatically saved.

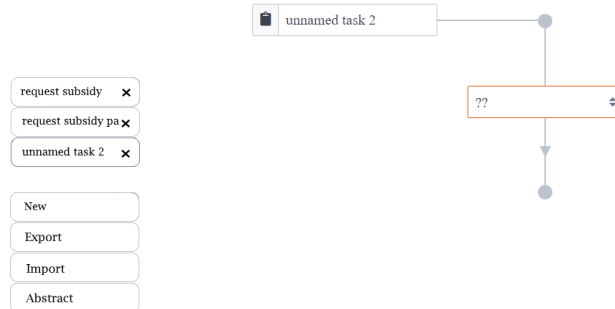


(c) After clicking the task in the menu, the user is presented the “request subsidy partial” taskflow, which they can now continue working on.

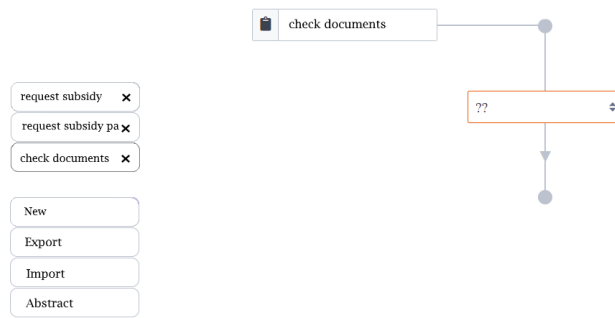
## 5.5.2 Creating new taskflows



(a) The user is working on task “request subsidy”, and would like to introduce a new taskflow. The user clicks on the “New” button on the left-hand side. Any changes made to “request subsidy” are automatically saved.

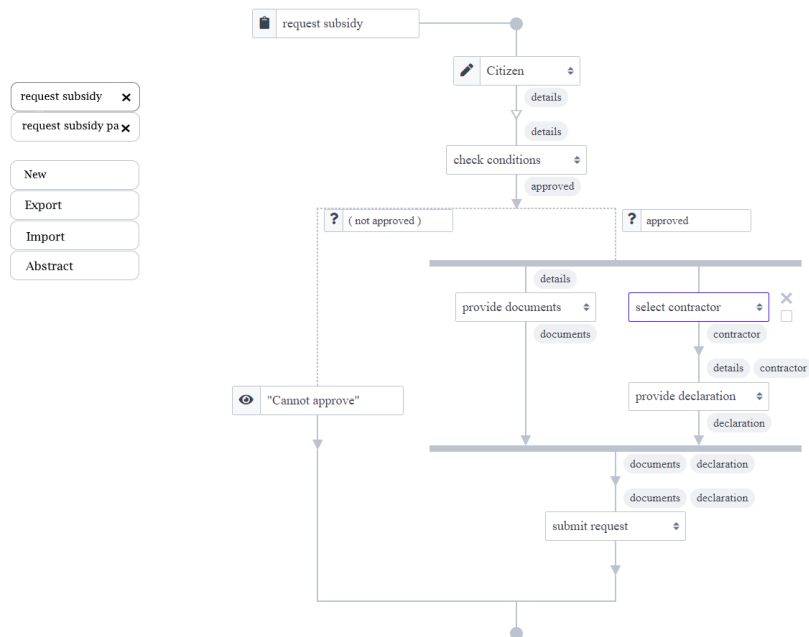


(b) After clicking the “New” button, the user is provided a new task flow, “unnamed task 2”, which consists of a hole and an empty lift. This new task is added to the menu on the side. The user edits its name by clicking the name above the taskflow.

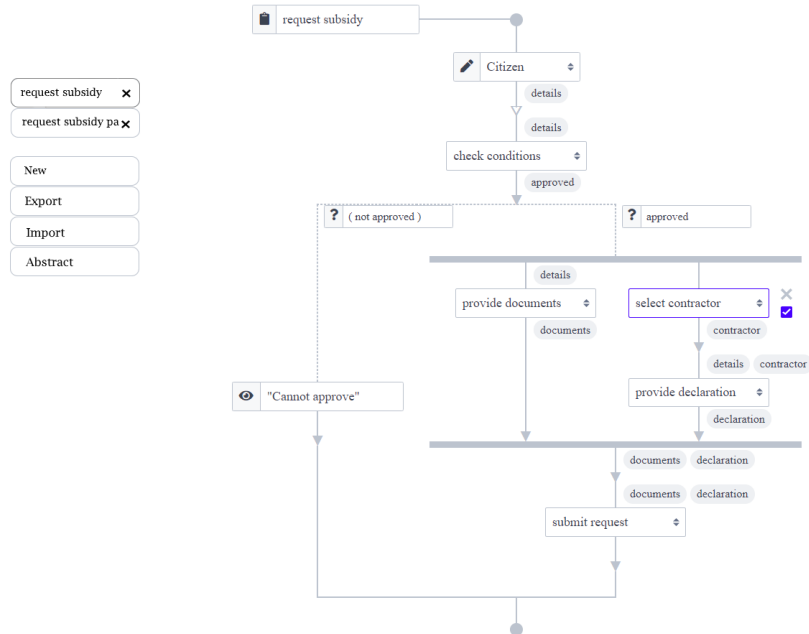


(c) The user changes the name to “check documents”. This also updates the name in the menu. The user can now begin working on this taskflow. If the user happens to enter a name that already exists in the program, an error message pops up.

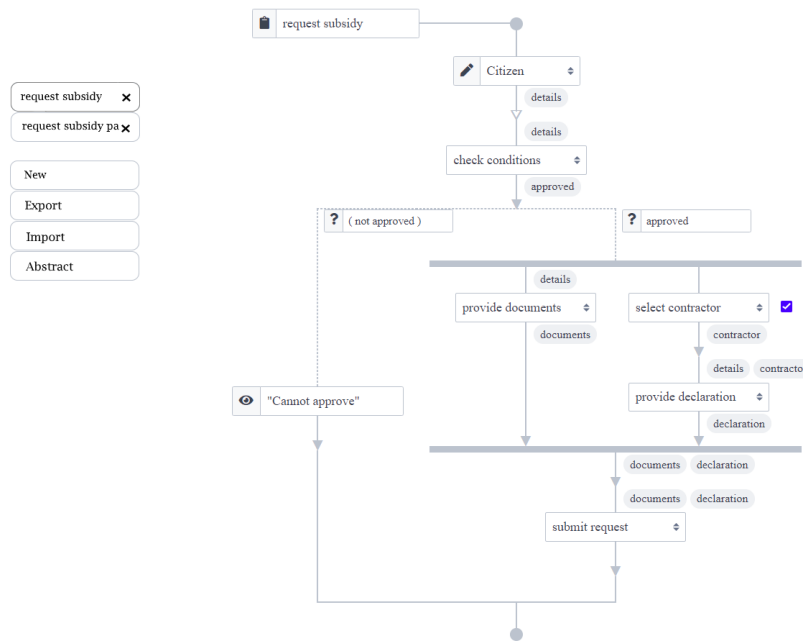
### 5.5.3 Making a selection



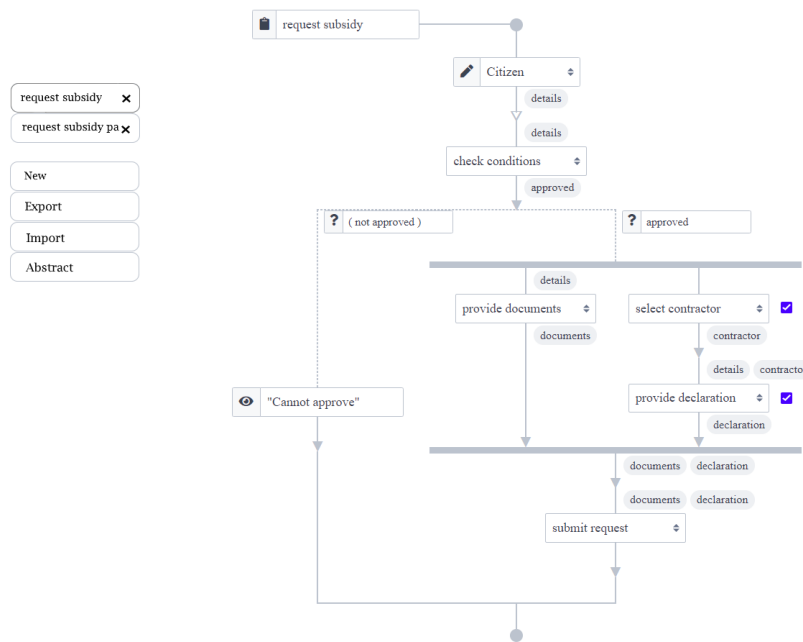
(a) The user would like to select multiple tasks in the flow, in order to perform an operation that involves multiple tasks. The user hovers over the “select contractor” task, which they would like to select, and is presented with several options on its right-hand side.



(b) The user clicks the checkbox next to the task, which is now checked.

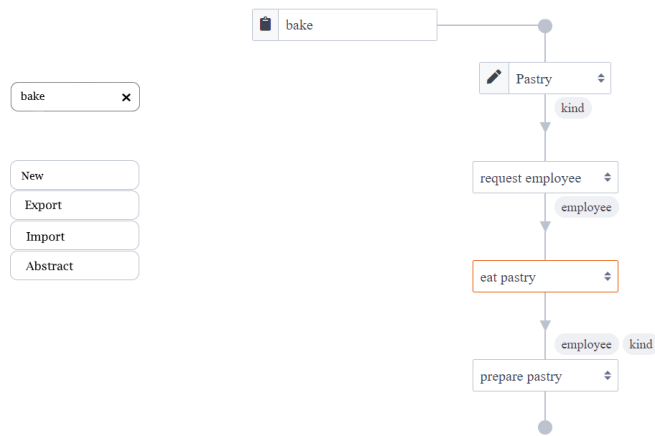


(c) When the user moves their mouse away from the task, the check mark remains visible.

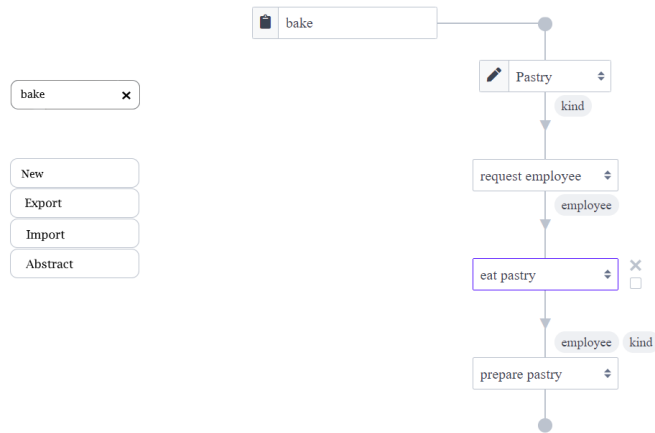


(d) In the same fashion, the user also selects the “provide declaration” task, in order to perform an operation on both tasks at once.

### 5.5.4 Removing a task

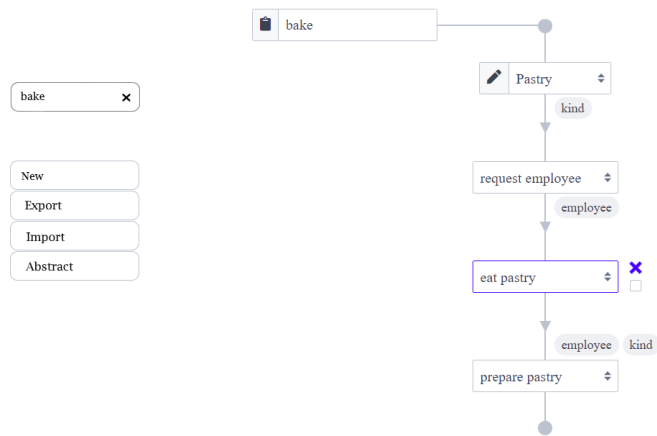


(a) The user notices a mistake in their taskflow, as indicated by the type checker. The user would like to remove the task “eat pastry” from the flow.

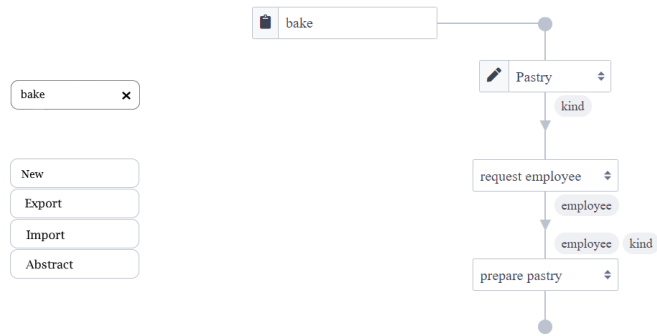


(b) The user hovers over the task, revealing the available options.



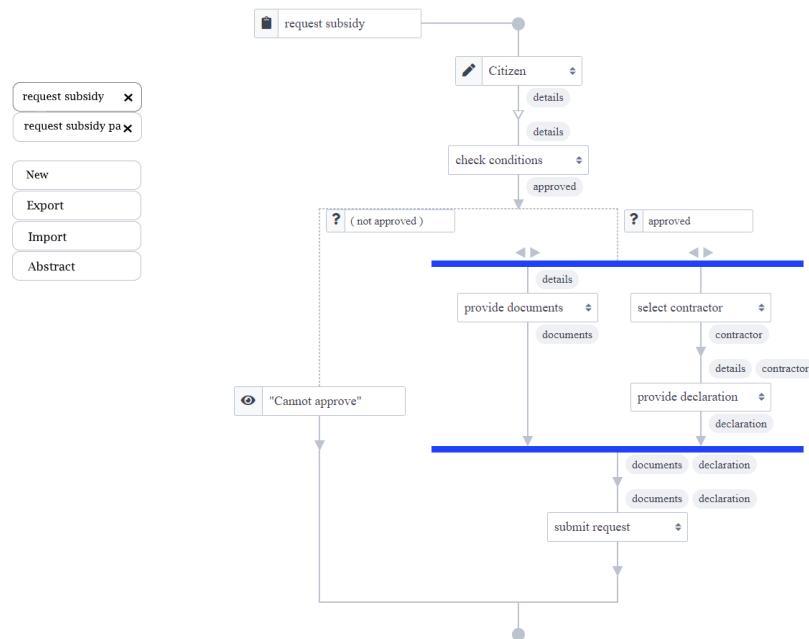


(c) The user clicks the cross icon.

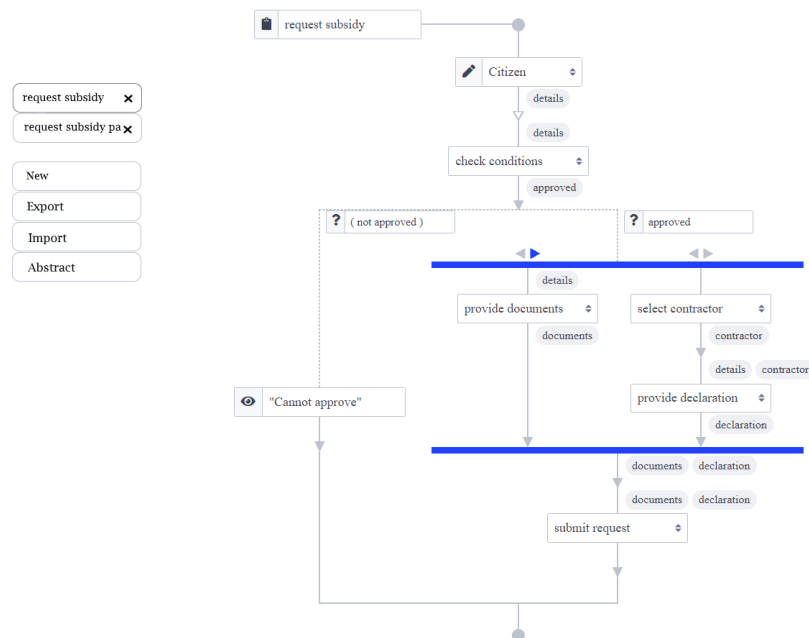


(d) Upon clicking the cross, the task “eat pastry” is removed from the taskflow.

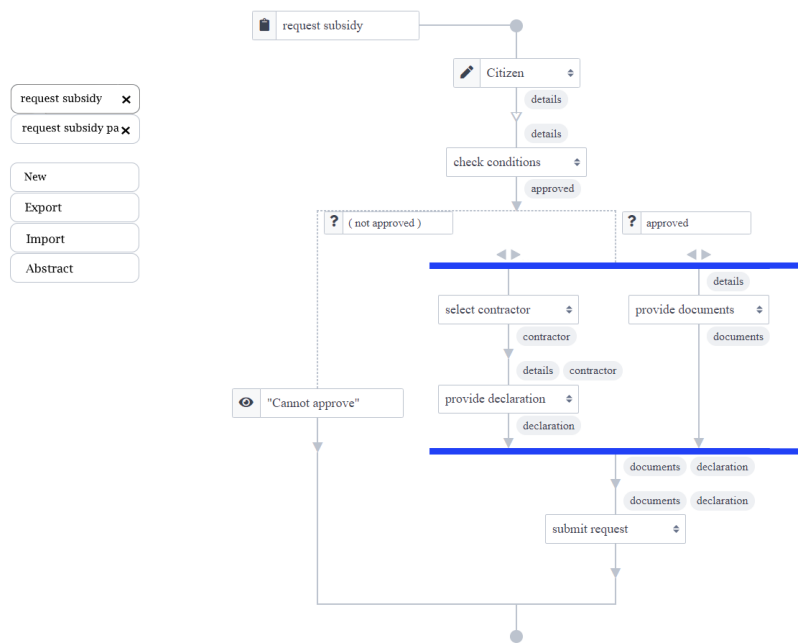
### 5.5.5 Swapping parallel branches



(a) The user is working on a taskflow, and would like to swap the two parallel branches. Upon hovering over the parallel section, the horizontal bars light up, indication that an action is possible. Two arrows appear over each branch.

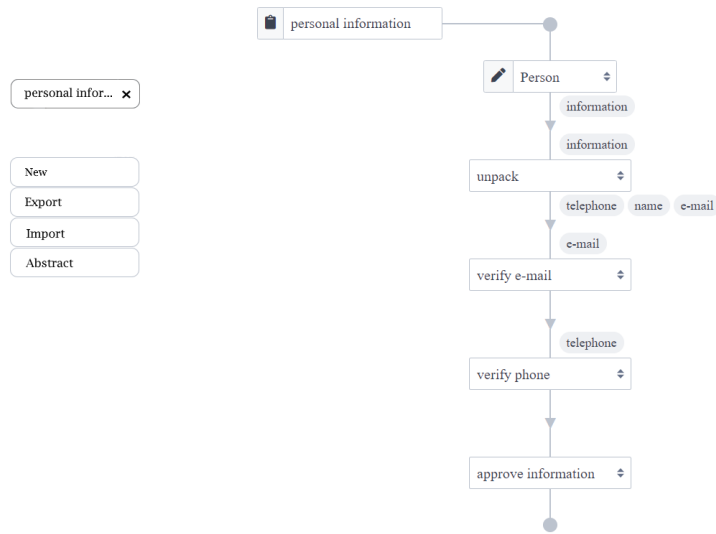


(b) The user hovers over an arrow, which lights up. The user then clicks on it.

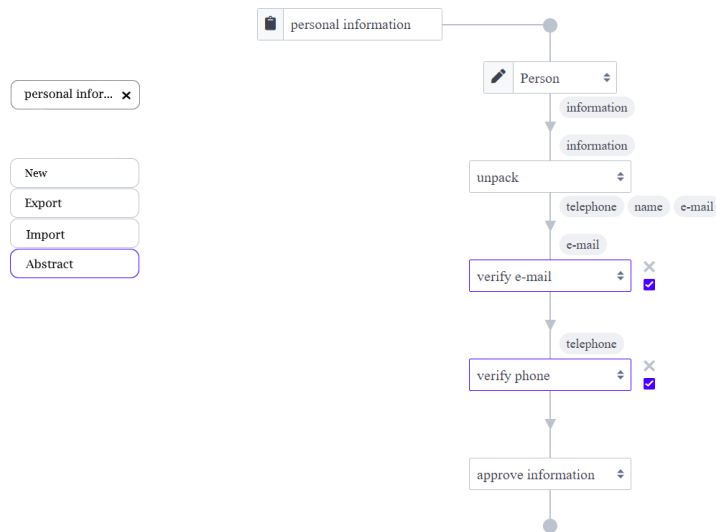


(c) After clicking the arrow, the branch moves in the direction of the arrow, swapping with the branch in that direction.

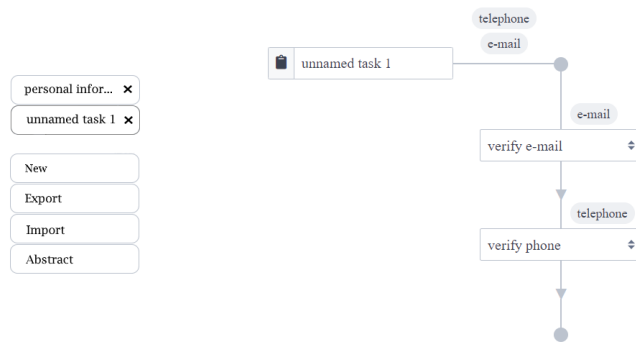
### 5.5.6 Abstracting tasks



(a) The user is working on a taskflow, and would like to abstract the verification tasks into a new taskflow.



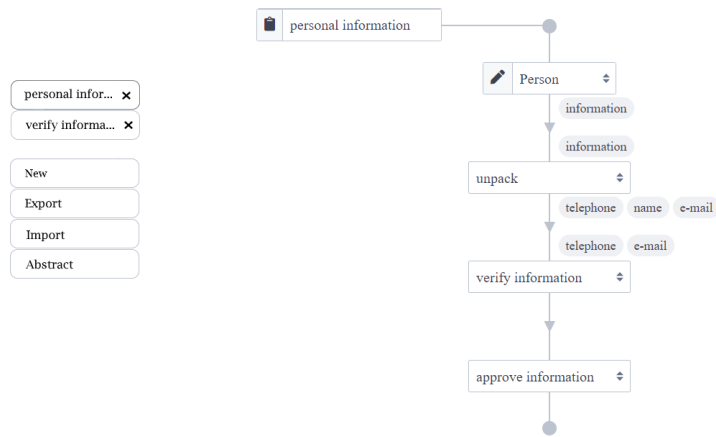
(b) To this end, the user selects both the “verify e-mail” as well as “verify phone” tasks. The “Abstract” button on the left lights up, to indicate that the selection can be abstracted.



(c) After clicking the button, the user is shown a new taskflow, which contains the selection. The required variables “telephone” and “e-mail” have also been added to this taskflow.

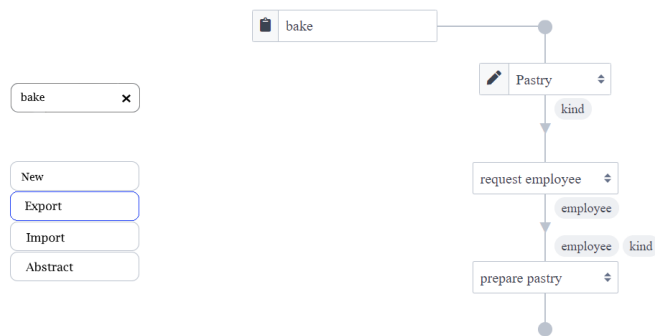


(d) The user renames the taskflow to a more descriptive name.

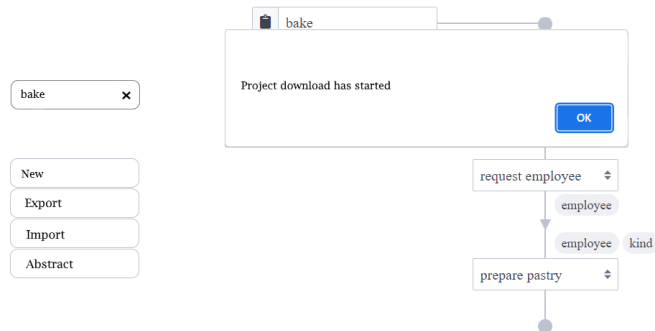


(e) Upon returning to the “personal information” taskflow, the user sees that the previously-made selection has been replaced by the new “verify information” task, and it receives the arguments “telephone” and “e-mail”.

## 5.6 Exporting tasks

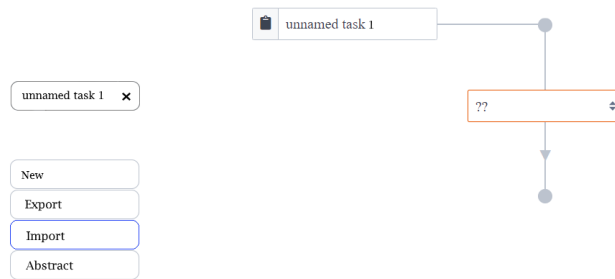


(a) The user has been working on a project, but would like to continue working some other time. They would like to export the project, and store it on their machine. To this end, they click the “Export” button.

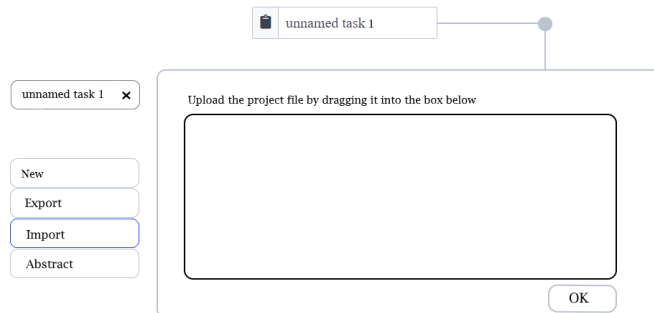


(b) A pop-up appears that notifies the user that the project is being downloaded. The result of the download is a text file containing all of the project data.

## 5.7 Importing tasks

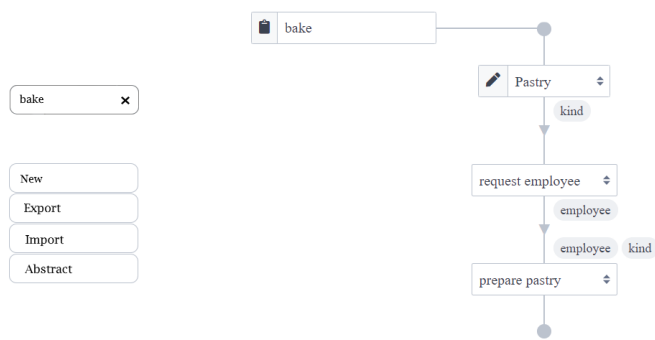


(a) The user would like to continue working on a previously made project. The project has been stored locally in a text file. The user now clicks the “Import” button.



(b) A layover screen with a box appears. The user drags the project file into the box. The project is now opened by the program.





(c) The layover screen disappears, and the user is presented with the project that they stored previously. The user can now continue to work on this project.

## Chapter 6

# Implementation

Some of the features of chapter 5 have been (partially) implemented<sup>1</sup>. As an example, I would like to discuss the implementation of the abstraction mechanism (storyboard 5.5.6), which I call the abstractor for brevity. This abstractor is the most advanced out of all the new features, as it depends on a lot of smaller features (see chapter 5).

### 6.1 Task implementation

Before discussing the implementation of the abstraction mechanism, a brief overview of the existing implementation is necessary. The main data type that the Builder revolves around is the **Task** datatype, implemented by Steenvoorden (2022).

```
data Task t
= Step Match t t
  | Branch (Branches t)
  | Select (LabeledBranches t)
  | Enter Name — Unvalued
  | Update Expression — Valued
  | Change Expression — Shared
  | View Expression — Valued read-only
  | Watch Expression — Shared read-only
  | Lift Expression
  | Pair (Array t)
  | Choose (Array t)
  | Execute Name Arguments
  | Hole Arguments
  | Share Expression
  | Assign Expression Expression
```

In this definition, **Branches t** is an **Array (Tuple Expression t)**, and **LabeledBranches t** is a labeled variant on this: **Array (Tuple Label (Tuple**

---

<sup>1</sup>Only a small part of the code is discussed here. For the full code, see <https://gitlab.science.ru.nl/dbremer/tophat-visual>.

`Expression t`). All but two of these tasks can be directly linked to the restricted tasks discussed in section 4.1.1. The `Branch` and `Select` data types are used to distinguish between *implicit* and *explicit* steps: `Branch` is used for implicit steps, and `Select` for explicit steps. These implicit and explicit steps correspond to the “step” and “select” tasks in table 4.1. Here, “step” is implicit and “select” is explicit. Since there is only one `Step` constructor in the definition of `Task`, the Builder must distinguish between implicit and explicit steps another way. To make this distinction, an invariant is introduced: a `Step` must always step to a `Branch` or a `Select` (it also holds that `Branch` and `Select` can never be outside of a `Step`). This means that the renderer only accepts steps of the form `Step m t1 (Branch [...])` or `Step m t1 (Select [...])`. If the stepped-to task only consists of a single branch, the forms `Step m t1 (Branch [Constant (B true) t2])` or `Step m t1 (Select ["Continue" Constant (B true) t2])` must be used.

The definition of `Task` incorporates a type `t`. This `t` is used to *Annotate* tasks with additional information. The `Annotation` type, also implemented by Steenvoorden (2022), is defined as follows:

```
data Annotated a f =
    Annotated a (f (Annotated a f))

type Checked = Annotated Status —Note the partial application
    of Annotated

data Status
    = Success Context FullType
    | Failure Context Error
    | Unknown
```

Now, a `Checked Task` is a taskflow where each task in the tree holds additional information: `Success` (including `Context` and a `FullType`), `Failure` (including `Context` and `Error`) or `Unknown`. The type checker is the unit that annotates the tasks in this way.

The selection mechanism uses one additional piece of information: is the current task selected or not? To implement this, I added another, very similar, definition:

```
type CheckedSelected = Annotated (Bool*Status)
```

In this definition, `*` is defined as an infix for `Tuple`. The `Bool` is `true` when the task is selected, and `false` if not. In the rest of the program, I work with the `CheckedSelected Task` data type.

## 6.2 Allowed selections

Before considering how to implement the abstractor, it is necessary to determine which selections can be abstracted. The expectation of the user should

be immediately clear to the abstractor. For this reason, the abstractor only accepts sequential selections (see figures 6.1 and 6.2). Abstracting these selection is a well-defined procedure, as the location of the resulting **Execute** is certain. Also, the abstraction is a well-defined taskflow without gaps.

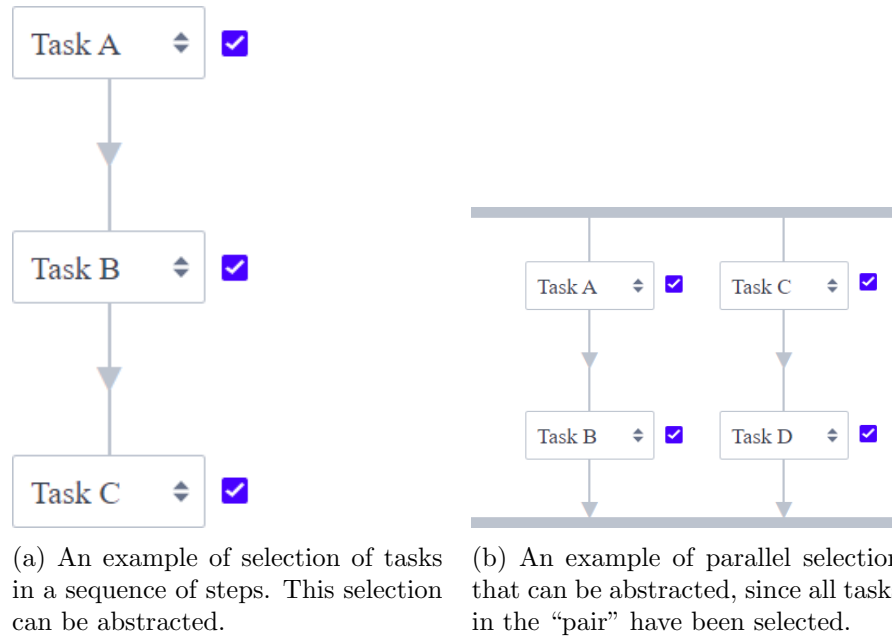


Figure 6.1: Several selections that can be abstracted.

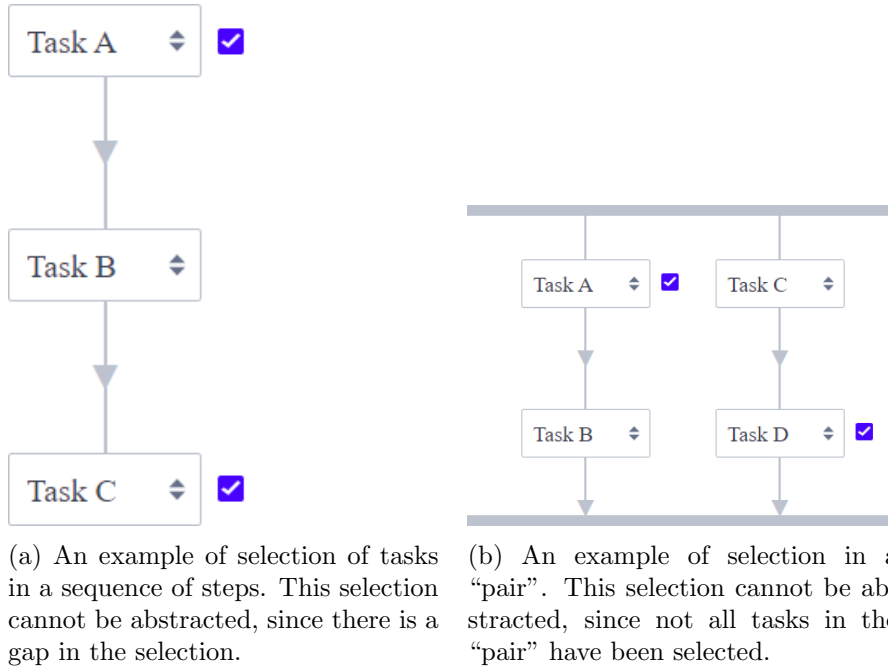


Figure 6.2: Several selections that cannot be abstracted.

In order to test whether a selection is viable, the program counts the *entry points* of the selection. Here, an entry point is defined as a **Step** from a task that has not been selected to a task that has been selected.

Importantly, this test requires a subroutine to be performed before execution. This subroutine, which I called **fixSelection**, makes sure that **Step m t1 t2** tasks are marked as selected if and only if **t1** is selected, and tasks of the form **Pair branches** and **Choice branches** are selected if and only if every task in the array **branches** is selected (however, this feature has not been entirely implemented yet).

With these selections in mind, a selection can only be abstracted if it has exactly one entry point: zero entry points would mean that the user selected no tasks, and more than one would mean that the selection is not sequential. See the **Selection** module for the code on finding entry points.

### 6.3 Abstractor implementation

The abstractor essentially does two things: isolate the selection from the taskflow, and replace the selection with an **Execute** task. The former relies on the entry points that were discussed earlier. The abstractor first looks through the taskflow to determine the entry point, and returns this entry point, including the “tail” of steps that lie past it. It then “cuts off” this tail, by finding an *exit point*: a **Step m t1 t2** where **t1** is selected, but **t2**

is not. The selection rules of **Pair** and **Choice** ensure that there can only ever be one exit point. When it finds this exit point, it returns **Step m t1 (Lift Wildcard)** instead of **t2**. This way, the selection is isolated from the taskflow. The code for this can be found in the **Selection** module.

The previously described procedure makes a copy of the selected part of the taskflow. This means that the selection is still part of the taskflow, which the user would like to have replaced. The **replaceSelectionWith** function does exactly this: it replaces the selected part of a taskflow with another task. In the case of the abstractor, this other task is always an **Execute** task. **replaceSelectionWith** first finds the entry point of the selection, and then replaces this with the replacement. Entry points are of the form **Step m t1 t2**, and turn into **Step m replacement t2** by applying **replaceSelectionWith**. Now, **t2** may still be selected. It should therefore be replaced by the aforementioned “tail” of the selection. The result of the entire operation is **Step m replacement tail**. The code for this operation can be found in the **Abstractor** module.

Now that the abstractor has performed the abstraction and replaced the selection, it should update the **World**, so that the changes that have been made are stored properly. Therefore, it adds the abstraction to the **Tasktext** under the name “unnamed task”, followed by a number, and replaces the current taskflow by the taskflow where the selection has been replaced. The program now continues execution with this new **World**.

## Chapter 7

# Related Work

Related to this thesis are the topics of visualising functional languages and programming with holes. Both of these areas have been studied before. The GiN (Henrix et al., 2012), Tonic (Stutterheim, 2017) and Mojito (Mol, 2020) systems are closely related to the TopHat Builder, and have already been discussed. For an overview of these systems, see section 2.1.

### 7.1 Visualisation of functional languages

The TopHat Builder is used to build taskflows visually. Since TopHat is embedded in a functional language, the visualisation of functional languages is closely related to this topic. Hanna (2006) uses a document-centered environment for the functional language Haskell, which allows a user to visually edit Haskell programs. It allows the user to define instances of the `Display` class, which contains functions for displaying data visually, as well as editing and adjusting that data.

Clerici et al. (2011) proposes another visualisation tool for functional languages: the NiMo system. The NiMo language is equivalent to Haskell, but uses visual graphs to represent programs. NiMo performs type inference on these programs, and shows the user visually where type conflicts occur. The user can then interact with the visual representation to solve these conflicts, and debug the program in this way.

### 7.2 Programming with holes

The TopHat Builder uses the concept of holes to represent unfinished tasks. The user can then change these holes to the correct task. The *Hazel* programming language (Omar et al., 2017) is a functional programming language that also uses holes. A hole in Hazel is a part of the program that is missing, or a part of the program that contains errors. By formally defining

these holes, Hazel is able to execute incomplete programs, and deliver partial results.

To a lesser extent, the Haskell GHC compiler<sup>1</sup> supports the use of holes in programs. These holes are again blank parts of the program, indicated by the wildcard symbol `_`. It is not possible to execute programs with holes, but the compiler will provide suggestions of bindings that fit the holes typing. This may help the user decide on how to fill the hole.

---

<sup>1</sup>[https://ghc.gitlab.haskell.org/ghc/doc/users\\_guide/extends/typed\\_holes.html](https://ghc.gitlab.haskell.org/ghc/doc/users_guide/extends/typed_holes.html)



## Chapter 8

# Conclusions

In this thesis, a design for an extension to the TopHat Builder has been drawn up, as well as partially implemented. Notable new features include the creation of new taskflows, deleting tasks from a taskflow, making a selection in a taskflow, abstracting a selection into a new taskflow, and importing and exporting projects. These features have been designed with cleanliness of the interface and logical placement of elements in mind.

### 8.1 Future work

There are a lot of possible extensions to the TopHat Builder. In fact, it could be described as “a project that is never truly done”, as there are always new additions to be made. The list of extensions that I show here is far from exhaustive.

Firstly, completing the list of transformations from section 4.1.2 would be a large step forward. This would allow a user to build a single, complete taskflow.

Secondly, the Builder lacks any form of type inference. There needs to be a type inference algorithm to calculate the types of taskflows, so that they can be used as **Execute** tasks in other taskflows. As it stands, the Builder assigns every task the same, trivial type.

Furthermore on the topic of types, the Builders needs a tool with which the user can create their own types. The Builder now only uses built-in types. It is essential that the user can create their own types, as every software project uses custom types. The Mojito system (Mol, 2020) does have such a system, which may fit the Builder as well.

Lastly, as an interesting extension to the abstractor, a “search and replace” functionality would be interesting. This would take two taskflows as input, one “base” and one “pattern”, and would replace every occurrence of the pattern in “base” with an **Execute** with the correct name. This way, the user could quickly replace multiple occurrences of an abstracted taskflow.

Besides extensions, usability studies should be conducted on the Builder. It is still unknown whether a user with no programming experience can find their way around the Builder. As a prerequisite to this study, the users' interpretation of these visualisations should be studied to test if user can intuitively understand these.

# Bibliography

- S. Clerici, C. Zoltan, and G. Prestigiacomo. Graphical and Incremental Type Inference: A Graph Transformation Approach. In R. Page, Z. Horváth, and V. Zsók, editors, *Trends in Functional Programming*, pages 66–83, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22941-1.
- K. Hanna. A Document-Centered Environment for Haskell. In A. Butterfield, C. Grelck, and F. Huch, editors, *Implementation and Application of Functional Languages*, pages 196–211, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-69175-4.
- J. Henrix, R. Plasmeijer, and P. Achten. GiN: A Graphical Language and Tool for Defining iTask Workflows. In R. Peña and R. Page, editors, *Trends in Functional Programming*, pages 163–178, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32037-8.
- J. Mol. iTask Mojito. Master’s thesis, Radboud University Nijmegen, 2020.
- C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, 2017.
- T. Steenvoorden. *TopHat*. PhD thesis, Radboud University Nijmegen, 2022.
- J. Stutterheim. *A Cocktail of Tools*. PhD thesis, Radboud University Nijmegen, 2017.