Bachelor thesis
Computing Science

Radboud University

Semantic Equivalence of Task-Oriented Programs in TopHat

Author:
Tosca Klijnsma
s4471393

First supervisor/assessor:
prof. dr. Herman Geuvers
herman@cs.ru.nl

Second supervisor/assessor:
Tim Steenvoorden MSc
tim@cs.ru.nl

January 27, 2020

Abstract

Task-oriented programming (TOP) is a new programming paradigm designed for describing multi-user workflows. It is implemented in the iTasks framework, in the functional programming language Clean. To reason formally about iTasks programs, a formal language called $\widehat{\text{TOP}}$ (TopHat) has been defined, together with its operational semantics. For proving properties about task-oriented programs, it is desirable to have a definition for the semantic equivalence of two $\widehat{\text{TOP}}$-programs. This thesis aims to answer this question. We show that a task can be in either one of five states after normalisation, and for every two tasks in the same state, we define what it means for them to be semantically equivalent. Using this definition, we define a number of properties we believe hold for $\widehat{\text{TOP}}$-programs. Amongst those, we show that the Task operation on types in $\widehat{\text{TOP}}$ cannot be a monad.

Contents

Chapter 1

Introduction

Task-oriented programming (TOP) is a new programming paradigm designed for developing distributed interactive multi-user systems. In this programming paradigm, the concept of a "task" plays a central role. A task is a unit of work assigned to some user, and consists of two parts: a description of the work that should be done, and a typed interface that defines the type of the task value that it returns. Tasks are described in an abstract, declarative manner, and from this abstract description, TOP automatically generates a GUI. It also takes care of the client-server communication that is needed for users to work together on tasks. TOP allows programmers to define workflows which describe what tasks should be executed by its users, without having to worry about how this is achieved.

User collaboration is a central concept in TOP. The different ways in which users can collaborate are captured by *task combinators*. By using these combinators, TOP-programmers can construct larger tasks from smaller ones in several ways. There is *sequential composition*, which allows tasks to be executed one after the other. And there is *parallel composition*, which allows tasks to be executed in parallel at the same time. For parallel composition, it is possible to either combine the results, or to conditionally continue with either one of two tasks.

In order to collaborate, users also need to be able to communicate with each other, and with the system. Using task combinators, it is already possible to pass along data from one task onto the next. For communication with the outside world, there are *editors*. Editors provide interaction with the environment via input events. They are typed containers which remember the last value that has been sent to them, and users can communicate with the system through these editors. Furthermore, they allow users to view and edit *shared data sources*, which are mutable references whose changes are immediately visible to all other tasks watching them.

Another important component of tasks is that they are typed. This type is important to determine the type of the task values that are communicated

to the environment. Not all tasks have a value, and a task does not produce just one value when it is complete. Instead, a task's value is continually updated while the work takes place, and can be observed at any point during execution. Moreover, it may be possible that a task never completes, and that its value never reaches a stable state. A task's value reflects a task's current progress. They can be inspected by other tasks to base decisions on, which in turn can impact the things users can see or do.

Finally, the TOP language is modular: tasks are composed of smaller tasks, and can be arguments or results of functions. This allows programmers to re-use tasks, and to model their own collaboration patterns [1, 6].

1.1 iTasks

TOP describes in an abstract way *what* work should be done by the system and its users. It does not describe *how* this should be done, this question should be answered by the TOP language implementation. The iTasks framework is an implementation of TOP, written in the pure and lazy functional programming language Clean. It is implemented as a shallowly embedded domain-specific language, which means that it inherits features from its host language Clean. Amongst these features is a strong typing system, and because Clean is a functional language, it allows task combinators to be expressed as functions. From the high level description of the tasks, iTasks generates a web application that is able to execute the described tasks. It takes care of generating a GUI, and of coordinating the tasks in a distributed manner by using a client-server architecture. The server side of an iTasks application runs a web service to which users on a wide range of different devices can connect, and the client side realises the front-end components. This way, programmers of iTasks-programs need not be concerned by lower-level implementation details. iTasks has shown itself effective in the past for the implementation of interactive, distributed, workflow applications [1].

1.2 Research question

Because iTasks has been designed for developing real-world applications, reasoning formally about iTasks programs is hard. The paper *TopHat: A formal foundation for task-oriented programming* introduces $\widehat{\text{TOP}}$ (TopHat), a formal language plus operational semantics for reasoning about task-oriented programs. A follow up paper, *TopHat Next: even more stylish task-oriented programming*, is currently being written [7]. This thesis uses these two papers as starting point. Our research question is the following:

When are two programs in $\widehat{\text{TOP}}$ *semantically equivalent?*

Furthermore, if we can define such a notion of semantic equivalence for task-oriented programs, what interesting properties can we prove (or disprove)? An example of an interesting property is whether the monad laws hold for the step combinator. Showing that certain equalities hold for task-oriented programs could prove useful for the iTasks system in the future. If we know that one task-oriented program is semantically equivalent to another, then we know that we can substitute one for the other, without changing the meaning of the program. Which in turn could be useful for doing compiler optimizations.

1.3 Structure of this thesis

Chapter 2 will explain the operational semantics of $\widehat{\text{TOP}}$. We will mostly follow the semantics presented in *TopHat Next* [7], but we omit some language constructs to ease our definition of semantic equivalence in the subsequent chapter. We do however keep all language features that capture the essence of TOP. In Chapter 3, we will give a definition for semantic equivalence in $\widehat{\text{TOP}}$, along with its motivation. We show that a task can be in either one of five states after normalisation, and for every two tasks in the same state, we define what it means for them to be semantically equivalent. Additionally, we present some properties of $\widehat{\text{TOP}}$-programs that we claim are true or false according to our definition. Finally, Chapter 4 will conclude this thesis.

Chapter 2

TopHat

$\widehat{\text{TOP}}$ (TopHat) is a formal language for reasoning about task-oriented programs. It is described by a layered operational semantics, consisting of multiple big-step semantic functions for reducing expressions, and two labelled transition systems for handling user inputs. Its main layers are evaluation, normalisation, and interaction. To make clear which features come from TOP and which features come from functional programming, $\widehat{\text{TOP}}$ is separated into a task language and an underlying host language [6, 7]. The host language will be described in Section 2.1. We will give its syntax, typing rules, and evaluation semantics. The task language, which is embedded into the host language, will be presented in Section 2.2. We will explain the task constructs, give their typing rules, and give the normalisation and interaction semantics. Finally, Section 2.3 will discuss some larger examples.

2.1 The host language

2.1.1 Syntax

The host language of $\widehat{\text{TOP}}$ is a simply typed λ-calculus, extended with some basic types. The grammar given in Figure 2.1 defines the syntax of $\widehat{\text{TOP}}$. Expressions e can be lambda expressions, variables, locations, branching, unit values, tuples, and constants for booleans, integers and strings. Booleans can be either True or False, integers use their decimal notation, and strings are enclosed by double quotation marks. There are a number of operations possible on expressions. There are equational operators ($<$, $\leq$, $\equiv$, $\not\equiv$, $\geq$, $>$), logical operators for boolean expressions ($\neg$, $\wedge$, $\vee$), and numerical operations for integers ($+$, $-$, $\times$, $/$). Locations l are used for references and shared editors, which will be defined later in Section 2.2.11. They are not meant to be used by the programmer directly. Additionally, expressions can be pretasks. Pretasks define the constructs of the task language, which we will explain in Section 2.2. For now however, we will only focus on the host

language, and postpone everything related to the task language until the next section.

Expressions	e	::=	$\lambda x : \tau.\ e \mid e_1\ e_2$	– abstraction, application
		|	$x \mid l \mid c$	– variable, location, constant
		|	$u\ e_1 \mid e_1\ o\ e_2$	– operations: unary, binary
		|	$\textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 \mid \langle\rangle$	– conditional, unit
		|	$\langle e_1, e_2\rangle \mid \textsf{fst}\ e \mid \textsf{snd}\ e$	– pair, projections
		|	p	– pretask
Constants	c	::=	$B \mid I \mid S$	– boolean, integer, string
Unary operations	u	::=	$\neg \mid -$	– not, negate
Binary operations	o	::=	$\wedge \mid \vee$	– logical
		|	$< \mid \leq \mid \equiv \mid \not\equiv \mid \geq \mid >$	– equational
		|	$+ \mid - \mid \times \mid /$	– numerical

Figure 2.1: Language grammar

Besides the expressions defined in the grammar, we will use the notation $e_1; e_2$ as an abbreviation for $(\lambda x : \textsc{Unit}.\ e_2)\ e_1$, where x is a fresh variable, and we will use the notation $\textsf{let}\ x : \tau = e_1\ \textsf{in}\ e_2$ as an abbreviation for $(\lambda x : \tau.\ e_2)\ e_1$. This is allowed because our evaluation semantics, which we will present in Section 2.1.3, is strict.

2.1.2 Typing

Besides function types, the simply typed λ-calculus of the host language is extended with pairs, unit types, references, task types, and primitive types for booleans, integers, and strings. Additionally, there are basic types β, which contain only a subset of all types τ. Figure 2.2 shows the type grammar of $\widehat{\text{TOP}}$. Reference types are used for locations. We say that a location l is of type $\textsc{Ref}\ \beta$ if it points to an expression of basic type β. To prevent recursive reference types, and to keep the language total, locations can only point to expressions of basic type. Task types $\textsc{Task}\ \tau$ are used for tasks. We will postpone the typing rules for tasks until Section 2.2.2.

Typing rules in $\widehat{\text{TOP}}$ are of the form $\Gamma, \Sigma \vdash e : \tau$, which should be read as "in environment Γ and store typing Σ, the expression e has type τ". The environment Γ is a mapping from variables to types, and is used in the rule T-Var to check the type of a variable, and updated in the rule T-Abs when using abstraction. The store typing Σ is a mapping from locations to types, and is used in the rule T-Loc to check the type of the expression that a

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

Figure 2.2: Type grammar

location refers to. Figure 2.3 shows the typing rules for expressions in the host language. To say that an expression e is of type τ, we will often omit the environment Γ and store typing Σ, and simply write $e : \tau$.

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

$$\text{T-Bool}\ \frac{c \in B}{\Gamma, \Sigma \vdash c : \textsc{Bool}} \qquad \text{T-Int}\ \frac{c \in I}{\Gamma, \Sigma \vdash c : \textsc{Int}} \qquad \text{T-String}\ \frac{c \in S}{\Gamma, \Sigma \vdash c : \textsc{String}} \qquad \text{T-Unit}\ \frac{}{\Gamma, \Sigma \vdash \langle\rangle : \textsc{Unit}}$$

$$\text{T-Var}\ \frac{x : \tau \in \Gamma}{\Gamma, \Sigma \vdash x : \tau} \qquad \text{T-Abs}\ \frac{\Gamma \cdot x : \tau_1, \Sigma \vdash e : \tau_2}{\Gamma, \Sigma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma, \Sigma \vdash e_2 : \tau_1}{\Gamma, \Sigma \vdash e_1\ e_2 : \tau_2}$$

$$\text{T-Loc}\ \frac{\Sigma(l) = \beta}{\Gamma, \Sigma \vdash l : \textsc{Ref}\ \beta} \qquad \text{T-If}\ \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Bool} \quad \Gamma, \Sigma \vdash e_2 : \tau \quad \Gamma, \Sigma \vdash e_3 : \tau}{\Gamma, \Sigma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}$$

$$\text{T-Tuple}\ \frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2}{\Gamma, \Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \text{T-First}\ \frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma, \Sigma \vdash \mathsf{fst}\ e : \tau_1} \qquad \text{T-Second}\ \frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma, \Sigma \vdash \mathsf{snd}\ e : \tau_2}$$

Figure 2.3: Typing rules for expressions in the host language

2.1.3 Semantics

Evaluating terms in the host language to values is handled by the evaluation semantics, where a value is an expression in the host language that cannot be reduced further. Values v can be lambda functions, pairs of values, unit, constants, locations, or tasks. Basic values b are a subset of values v that are of basic type β. The grammar for values is given in Figure 2.4.

The host language evaluates expressions using a big-step semantics. We denote the evaluation of expression e to the value v by $e \downarrow v$. Figure 2.5 gives the evaluation rules for expressions in the host language. The evaluation rules for the unary and binary operators are trivial, and so we omit them. We will postpone the evaluation of pretasks to tasks until Section 2.2.3.

Values	v	::=	$\lambda x : \tau.\ e \mid \langle v_1, v_2 \rangle \mid \langle\rangle$	– abstraction, pair, unit
		|	$c \mid l \mid t$	– constant, location, task
Basic values	b	::=	$\langle\rangle \mid \langle b_1, b_2 \rangle \mid c$	– unit, pair, constant

Figure 2.4: Value grammar for the host language

$$\boxed{e \downarrow v}$$

$$\text{E-Value}\ \frac{}{v \downarrow v} \qquad \text{E-App}\ \frac{e_1 \downarrow \lambda x : \tau.e_1' \quad e_2 \downarrow v_2 \quad [x \mapsto v_2]e_1' \downarrow v_1}{e_1 e_2 \downarrow v_1}$$

$$\text{E-IfTrue}\ \frac{e_1 \downarrow \mathsf{True} \quad e_2 \downarrow v_2}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \downarrow v_2} \qquad \text{E-IfFalse}\ \frac{e_1 \downarrow \mathsf{False} \quad e_3 \downarrow v_3}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \downarrow v_3}$$

$$\text{E-Tuple}\ \frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2}{\langle e_1, e_2 \rangle \downarrow \langle v_1, v_2 \rangle} \qquad \text{E-First}\ \frac{e \downarrow \langle v_1, v_2 \rangle}{\mathsf{fst}\ e \downarrow v_1} \qquad \text{E-Second}\ \frac{e \downarrow \langle v_1, v_2 \rangle}{\mathsf{snd}\ e \downarrow v_2}$$

Figure 2.5: Evaluation rules for expressions in the host language

2.2 The task language

2.2.1 Syntax

The task language of $\widehat{\text{TOP}}$ is embedded into the host language described in the previous section. As Figure 2.1 shows, expressions in $\widehat{\text{TOP}}$ can also be pretasks. A pretask p is a task that contains unevaluated sub-expressions. Pretasks can either be atomic tasks, or task combinators that compose larger pretasks from smaller ones. Figure 2.6 shows the grammar of pretasks.

Pretasks	p	::=	$vn.\ d \mid vn.\ e \mid d$	– label allocation, editors
		|	$e_1 \bowtie e_2 \mid e_1 \blacklozenge e_2$	– pairing, choosing
		|	$\lightning \mid \blacksquare\, e$	– fail, internal value
		|	$e_1 \blacktriangle e_2 \mid e_1 \blacktriangleright e_2$	– transform, step
		|	$\textbf{ref}\ e \mid e_1 := e_2$	– new shared value, assignment
Editors	d	::=	$\boxtimes^n \beta \mid \square^n e \mid \boxplus^n e$	– editors: unvalued, valued, shared

Figure 2.6: Pretasks grammar

Atomic tasks are atomic units of work that do not contain subtasks. These include internal values, failing, editors, reference creation and assign-

ment. Internal values ($\blacksquare\, e$) simply return the expression e as result. A failing task ($\lightning$) stands for an impossible task. We describe internal value and failing in Section 2.2.7. Editors ($\boxtimes^n \beta$, $\square^n e$, $\boxplus^n e$) provide communication with the environment. They are labeled by a label n, and require their input to be tagged by the same label. To allocate a label n within an expression e, we write $\nu n.e$. For label allocation, the programmer is only allowed to write $\nu n.d$. The pretasks $\nu n.e$ and d are not meant to be used by the programmer directly. We explain editors in Section 2.2.4, and label allocation in Section 2.2.5. Finally, reference creation ($\mathbf{ref}\, e$) and assignment ($e_1 := e_2$) enable the creation and modification of shared data. We describe references in Section 2.2.11.

Task combinators describe ways in which users can collaborate. They provide a means to combine smaller tasks into larger ones. There are several ways to do this. There is *sequential composition*, where the result of one task can be used in the next. This is captured by the step combinator ($\blacktriangleright$), which we will explain in Section 2.2.8. And there is *parallel composition*, which comes in two flavors. There is pairing ($\bowtie$), or *and*-parallel, which combines the result of two tasks into one task. And there is choosing ($\blacklozenge$), or *or*-parallel, which chooses the result of the leftmost task that has a value. The pair and choice combinators are described in Section 2.2.10. Finally, there is the transform combinator ($\blacktriangle$), which applies a function to the result of a task, resulting in a new task. We describe the transform combinator in Section 2.2.9.

2.2.2 Typing

The typing rules for pretasks are presented in Figure 2.7. A task t is of type $\textsc{Task}\ \tau$ if it should produce a result of type τ. Label allocation can be of any $\textsc{Task}$-type (T-$\textsc{Label}$). Editors are typed containers of type $\textsc{Task}\ \beta$ for some basic type β, and only accept inputs of the same type β (T-$\textsc{Enter}$, T-$\textsc{Update}$, T-$\textsc{Change}$). By only allowing editors to edit basic values, this prevents higher-order tasks like $\nu n.\ \boxtimes^n (\textsc{Int} \to \textsc{Int})$, $\nu n.\ \boxtimes^n (\nu n.\ \boxtimes^n \textsc{Int})$, or $\nu n.\, \square^n(\lambda x : \tau.x)$ from being defined, which have no meaning in $\widehat{\textsc{top}}$. Failing tasks can be of any $\textsc{Task}$-type (T-$\textsc{Fail}$). Pairing combines the result of the two tasks into a tuple (T-$\textsc{Pair}$), and choosing requires that both operands are of the same $\textsc{Task}$ -type (T-$\textsc{Choose}$). Step requires that the continuation e_2 on right-hand side is a function that takes the result from the task on the left-hand side and produces a new task (T-$\textsc{Step}$). Similarly, transform requires that the function e_1 on the left-hand side takes the result from the task on the right-hand side (T-$\textsc{Trans}$). Reference creation produces a task of type $\textsc{Task}\ (\textsc{Ref}\ \beta)$ that contains the newly created location l of type $\textsc{Ref}\ \beta$ (T-$\textsc{Share}$). Lastly, reference assignment returns a task containing the unit value as result, and thus is of type $\textsc{Task}\ \textsc{Unit}$ (T-$\textsc{Assign}$).

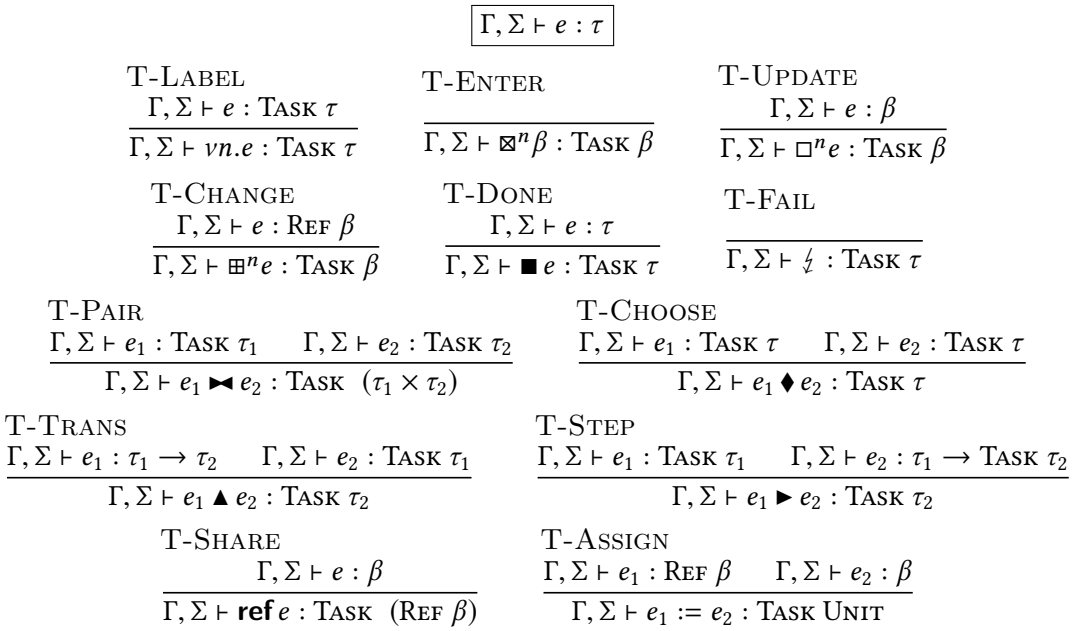

Figure 2.7: Typing rules for tasks

2.2.3 Semantics

$\widehat{\text{TOP}}$ uses a layered semantics to separate the semantics of the host language from the task language. So far, we have only seen the bottom layer of this semantics, namely evaluation ($\downarrow$). Evaluation is responsible for evaluating expressions in the host language to values. Before we explain the layers on top of evaluation, we still need to describe how to evaluate pretasks, which are also expressions in the host language. Pretasks evaluate to task values t. Figure 2.8 shows the task grammar. Whereas pretasks can have unevaluated sub-expressions, tasks can only contain subtasks. Exceptions to this are transform ($\blacktriangle$) and step ($\blacktriangleright$), where evaluation of the left-hand side respectively the right-hand side is delayed. Evaluation of pretasks to tasks is defined in Figure 2.9. Most task constructs simply evaluate their operands to values.

Figure 2.10 shows a graphical representation of the semantic layers and their relation. After evaluation is done, a task is ready to be normalised. Normalisation is a big-step semantics that is responsible for reducing tasks until they are ready to accept input. We write $t, \sigma \Downarrow t', \sigma', \delta'$ to denote the normalisation of task t in state σ to task t' in state σ'. The state σ is a mapping from locations to basic values. It keeps track of all references creates so far, and what value they currently hold. We will give the normalisation rules for each task construct in their respective subsections.

Tasks	t	::=	$\nu n.t \mid \boxtimes^n \beta \mid \square^n b \mid \boxplus^n l$	– label allocation, editors
		$\mid$	$t_1 \bowtie t_2 \mid t_1 \blacklozenge t_2$	– pairing, choosing
		$\mid$	$\lightning \mid \blacksquare v$	– fail, internal value
		$\mid$	$e_1 \blacktriangle t_2 \mid t_1 \blacktriangleright e_2$	– transform, step
		$\mid$	$\mathbf{ref}\ b \mid l := b$	– new shared value, assignment

Figure 2.8: Task grammar

$$\boxed{e \downarrow v}$$

$$\text{E-Label}\ \frac{e \downarrow t}{\nu n.e \downarrow \nu n.t} \qquad \text{E-Enter}\ \frac{}{\boxtimes^n \beta \downarrow \boxtimes^n \beta} \qquad \text{E-Update}\ \frac{e \downarrow b}{\square^n e \downarrow \square^n b} \qquad \text{E-Change}\ \frac{e \downarrow l}{\boxplus^n e \downarrow \boxplus^n l}$$

$$\text{E-Done}\ \frac{e \downarrow v}{\blacksquare e \downarrow \blacksquare v} \qquad \text{E-Pair}\ \frac{e_1 \downarrow t_1 \quad e_2 \downarrow t_2}{e_1 \bowtie e_2 \downarrow t_1 \bowtie t_2} \qquad \text{E-Choose}\ \frac{e_1 \downarrow t_1 \quad e_2 \downarrow t_2}{e_1 \blacklozenge e_2 \downarrow t_1 \blacklozenge t_2}$$

$$\text{E-Fail}\ \frac{}{\lightning \downarrow \lightning} \qquad \text{E-Trans}\ \frac{e_2 \downarrow t_2}{e_1 \blacktriangle e_2 \downarrow e_1 \blacktriangle t_2} \qquad \text{E-Step}\ \frac{e_1 \downarrow t_1}{e_1 \blacktriangleright e_2 \downarrow t_1 \blacktriangleright e_2}$$

$$\text{E-Share}\ \frac{e \downarrow b}{\mathbf{ref}\, e \downarrow \mathbf{ref}\, b} \qquad \text{E-Assign}\ \frac{e_1 \downarrow l \quad e_2 \downarrow b}{e_1 := e_2 \downarrow l := b}$$

Figure 2.9: Evaluation rules for pretasks

Normalisation also returns a set δ', which contains all locations whose value have been changed while normalisation took place. It will be used in the fixing semantics, which is defined on top of normalisation. Due to mutable references, the fixing semantics is required to make sure a task is fully normalised before any user interaction is allowed. We write $t, \sigma, \delta \Downarrow t', \sigma'$ to denote the fixing of task t in state σ given the set δ, resulting in task t' and state σ'. The fixing semantics will be explained in Section 2.2.11 on references.

For user interaction there are the handling and interaction semantics. Both semantics are small-step semantics that take an input event i for each step. For the handling semantics, we write $t, \sigma \xrightarrow{i} t', \sigma', \delta'$ to denote that handling the input i in task t and state σ results in the task t' and state σ'. Similar to the normalisation semantics, the handling semantics also returns a set δ', which contains all locations whose value have been changed while handling input. For the interaction semantics, we write $t, \sigma \overset{i}{\Rightarrow} t', \sigma'$ to denote that task t in state σ transitions to task t' in state σ' after the user

interaction i. The interaction semantics makes use of both the fixing and handling semantics to make sure that, after user interaction, a task is fully reduced and ready to accept the next input. The handling and interaction semantics will be discussed in Section 2.2.6 on input handling.

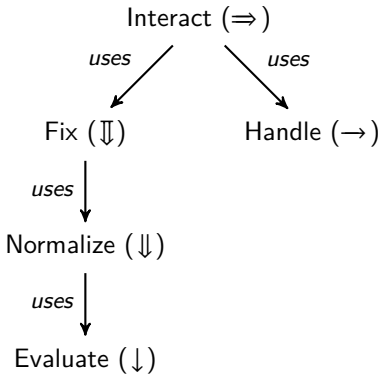

Figure 2.10: Semantic functions and their relation

2.2.4 Editors

Editors allow end users to interact with the system by entering and changing information. When a user sends an input event to an editor, the editor will update its current value to reflect the change. There are no output events. Instead, the current value of an editor can be observed and used in subsequent tasks. There are three types of editors in $\widehat{\text{TOP}}$:

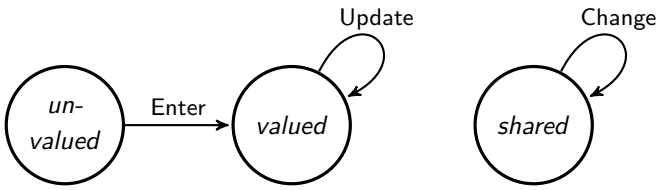

Figure 2.11: Possible editor states and their transitions.

- *Empty editors* ($\boxtimes^n \beta$) or *unvalued editors* are editors that currently hold no value. They can be seen as an input prompt to the user to enter data. Empty editors are annotated with a basic type β, which means that only basic values of type β are accepted by the editor. Once an empty editors receives a valid input event, it becomes a filled editor containing the new data.
- *Filled editors* ($\square^n b$) or *valued editors* are editors that currently hold the basic value b. Filled editors can be seen as either outputting a value, or as an input prompt that comes with a default value. They can never be cleared, only updated with new values of the same type.

- Shared editors ($\boxplus^n l$) watch references. They allow the user to view and change shared values. Whenever a shared editor is updated, all shared editors watching the same reference will be updated as well.

The relation between the different editors is illustrated in the state diagram in Figure 2.11. Since editors are already fully reduced, no normalisation needs to be done. Figure 2.12 gives the normalisation rules for editors.

$$\boxed{t, \sigma \Downarrow t', \sigma', \delta'}$$

$$\text{N-ENTER}\ \frac{}{\boxtimes^n \beta, \sigma \Downarrow \boxtimes^n \beta, \sigma, \varnothing} \qquad \text{N-UPDATE}\ \frac{}{\square^n b, \sigma \Downarrow \square^n b, \sigma, \varnothing} \qquad \text{N-CHANGE}\ \frac{}{\boxplus^n l, \sigma \Downarrow \boxplus^n l, \sigma, \varnothing}$$

Figure 2.12: Normalisation rules for editors

2.2.5 Label allocation

Editors are labeled, and require their input to be tagged by the same label. Label allocation ($\nu n.e$) ensures that the label n is bound within the expression e. Hence, it may be renamed within e, so long as there are no name conflicts. This is useful for introducing new labels within a task-program, without having to pre-assign all labels beforehand. This idea of label allocation closely follows the idea of channel allocation in π-calculus [2]. For normalisation, ν-expressions simply normalise their body. See the normalisation rule in Figure 2.13.

$$\boxed{t, \sigma \Downarrow t', \sigma', \delta'}$$

$$\text{N-LABEL}\ \frac{t, \sigma \Downarrow t', \sigma', \delta'}{\nu n.t, \sigma \Downarrow \nu n.t', \sigma', \delta'}$$

Figure 2.13: Normalisation rule for label allocation

Before sending input to a task-program, it should be in ν-standard form, which is defined as follows:

Definition 2.1 (ν-STANDARD FORM). A task t is in *ν-standard form* iff t is of the form $\nu n_1 ... \nu n_m . t'$, where t' does not contain any free ν-expressions. We will say a ν-expression is *free* iff it does not occur within a λ-expression. In this case, we call t' the *body* of t.

We will also define the task observation $\mathcal{N}$: Tasks $\rightarrow$ Booleans which, given a task t, returns true iff t is in ν-standard form. To bring a task-program in ν-standard form, we introduce ν-congruence. Two expressions e_1 and e_2 are said to be *ν-congruent* (denoted by $e_1 \equiv e_2$) if they are identical up to label names and the scope of ν-expressions. The rules by which to induce that two expressions are ν-congruent are given in Figure 2.14.

$$\boxed{e \equiv e'}$$

$$\text{C-Alpha}\ \frac{e =_\alpha e'}{e \equiv e'} \qquad \text{C-Reorder}\ \frac{}{\nu n.\nu m.e \equiv \nu m.\nu n.e} \qquad \text{C-Deallocate}\ \frac{}{\nu n.e \equiv e}\ n \notin e$$

$$\text{C-PairLeft}\ \frac{}{(\nu n.e_1) \bowtie e_2 \equiv \nu n.e_1 \bowtie e_2}\ n \notin e_2 \qquad \text{C-PairRight}\ \frac{}{e_1 \bowtie (\nu n.e_2) \equiv \nu n.e_1 \bowtie e_2}\ n \notin e_1$$

$$\text{C-ChooseLeft}\ \frac{}{(\nu n.e_1) \blacklozenge e_2 \equiv \nu n.e_1 \blacklozenge e_2}\ n \notin e_2 \qquad \text{C-ChooseRight}\ \frac{}{e_1 \blacklozenge (\nu n.e_2) \equiv \nu n.e_1 \blacklozenge e_2}\ n \notin e_1$$

$$\text{C-Step}\ \frac{}{(\nu n.e_1) \blacktriangleright e_2 \equiv \nu n.e_1 \blacktriangleright e_2}\ n \notin e_2 \qquad \text{C-Trans}\ \frac{}{e_1 \blacktriangle (\nu n.e_2) \equiv \nu n.e_1 \blacktriangle e_2}$$

Figure 2.14: Congruence rules

The rule C-Alpha introduces α-conversion for ν-expressions. Similar to α-conversion in λ-calculus, this means that bound labels within e may be renamed, so long as there are no name conflicts. The rule C-Reorder says that the order of ν-expressions does not matter, and the rule C-Deallocate says that labels may be de-allocated once they are no longer in use. All other rules are *scope extrusion* rules, which extend the scope of a label n. This is allowed so long as n does not occur in the rest of the expression that is now included within the scope. We write $n \notin e$ to say that the expression e does not contain the label n. Because $\nu n.e$ must be of type Task, we only apply scope extrusion to expressions whose sub-expressions can also be of type Task. This means that any ν's occurring within the right-hand side of step, or within a lambda function, are not extruded until after the step is taken, or after the function is evaluated. This is fine, because such editors are not yet reachable by the user, and cannot receive input yet.

In addition to the congruence rules defined in Figure 2.14, we also need a set of rules that say that congruence can be applied recursively to sub-expressions. For example, for label allocation we need a rule that says that if $e \equiv e'$, then $\nu n.e \equiv \nu n.e'$; and for pairing we need a rule that says that if $e_1 \equiv e_1'$, and $e_2 \equiv e_2'$, then $e_1 \bowtie e_2 \equiv e_1' \bowtie e_2'$. These rules can easily be derived

from the grammar, and so we will not give them here. We will simply say that, because $\equiv$ is a congruence relation, it can be applied structurally to sub-expressions.

2.2.6 Input handling

Once a program is in ν-standard form, it is possible to send input to it. The input event $E^n b$ indicates that the input b should be entered into the editor with label n. To do this, it is useful to know what inputs a given task accepts. The observation function $\mathcal{I}$ returns the set of input events that are currently possible for a given task. Its definition is given in Figure 2.15. For the three editors, $\mathcal{I}$ returns all input events $E^n b$ where b is of the correct type. For label allocation and the task combinators, $\mathcal{I}$ is defined recursively. For all other tasks, $\mathcal{I}$ returns the empty set. We consider an input event i a *valid* input event for the task t iff $i \in \mathcal{I}(t)$.

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\mathcal{I} : \text{Tasks} \to \mathbb{P}(\text{Inputs})} \\
\mathcal{I}(\nu n.t) & = & \mathcal{I}(t) & \\
\mathcal{I}(\boxtimes^n \beta) & = & \{E^n b' \mid b' : \beta\} & \\
\mathcal{I}(\square^n b) & = & \{E^n b' \mid b' : \beta\} & \textbf{where } \square^n b : \textsc{Task}\ \beta \\
\mathcal{I}(\boxplus^n l) & = & \{E^n b' \mid b' : \beta\} & \textbf{where } \boxplus^n l : \textsc{Task}\ \beta \\
\mathcal{I}(e_1 \blacktriangle t_2) & = & \mathcal{I}(t_2) & \\
\mathcal{I}(t_1 \blacktriangleright e_2) & = & \mathcal{I}(t_1) & \\
\mathcal{I}(t_1 \bowtie t_2) & = & \mathcal{I}(t_1) \cup \mathcal{I}(t_2) & \\
\mathcal{I}(t_1 \blacklozenge t_2) & = & \mathcal{I}(t_1) \cup \mathcal{I}(t_2) & \\
\mathcal{I}(\_) & = & \varnothing &
\end{array}
$$

Figure 2.15: Inputs observation on tasks

Finally, we are able to define how input events should be handled by tasks. This is done by the handling semantics ($\xrightarrow{i}$), whose rules are given in Figure 2.16. Most handling rules simply pass along the input events to their subtasks. The only interesting rules are the handling rules for editors. A valid input event to an empty editor results in a filled editor containing the new data (H-Enter), a valid input event to a filled editor updates its value (H-Update), and a valid input event to a shared editor updates the state σ such that the reference l now contains the new value (H-Change). It also returns $\delta = \{l\}$, which will later be used in the fixing semantics described in Section 2.2.11.

The interaction semantics uses the handling semantics to first handle the input i, after which it uses the fixing rules to make the task ready to accept the next input. The interaction semantics are given in Figure 2.17.

$$\boxed{t, \sigma \xrightarrow{i} t', \sigma', \delta'}$$

H-Label
$$\frac{t, \sigma \xrightarrow{i} t', \sigma', \delta'}{vn.t, \sigma \xrightarrow{i} vn.t', \sigma', \delta'}$$

H-Enter
$$\frac{}{\boxtimes^n \beta, \sigma \xrightarrow{E^n b'} \square^n b', \sigma, \varnothing}\ b' : \beta$$

H-Update
$$\frac{}{\square^n b, \sigma \xrightarrow{E^n b'} \square^n b', \sigma, \varnothing}\ b, b' : \beta$$

H-Change
$$\frac{}{\boxplus^n l, \sigma \xrightarrow{E^n b'} \boxplus^n l, \sigma[l \mapsto b'], \{l\}}\ \sigma(l), b' : \beta$$

H-Step
$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma', \delta'}{t_1 \blacktriangleright e_2, \sigma \xrightarrow{i} t_1' \blacktriangleright e_2, \sigma', \delta'}$$

H-Trans
$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma', \delta'}{e_1 \blacktriangle t_2, \sigma \xrightarrow{i} e_1 \blacktriangle t_2', \sigma', \delta'}$$

H-PairFirst
$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma', \delta'}{t_1 \bowtie t_2, \sigma \xrightarrow{i} t_1' \bowtie t_2, \sigma', \delta'}$$

H-PairSecond
$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma', \delta'}{t_1 \bowtie t_2, \sigma \xrightarrow{i} t_1 \bowtie t_2', \sigma', \delta'}$$

H-ChooseFirst
$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma', \delta'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{i} t_1' \blacklozenge t_2, \sigma', \delta'}$$

H-ChooseSecond
$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma', \delta'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{i} t_1 \blacklozenge t_2', \sigma', \delta'}$$

Figure 2.16: Handling rules

$$\boxed{t, \sigma \overset{i}{\Rightarrow} t', \sigma'}$$

I-Handle
$$\frac{t, \sigma \xrightarrow{i} t', \sigma', \delta' \qquad t', \sigma', \delta' \Downarrow t'', \sigma''}{t, \sigma \overset{i}{\Rightarrow} t'', \sigma''}$$

Figure 2.17: Interaction rules

2.2.7 Failing and internal value

A failing task ($\lightning$) stands for an impossible task. A task that is failing never has a value and never accepts input. Not just $\lightning$ can fail, tasks with failing subtasks can also fail. For example, the pairing ($\lightning \bowtie \lightning$) is also a failing task, and is equivalent to $\lightning$. To capture what tasks are failing we introduce the

failing observation $\mathcal{F}$, which is defined in Figure 2.18. Failing is especially useful when used in combination with the step combinator, which will be explained in the next section.

$$
\begin{array}{lcl}
\mathcal{F} : \text{Tasks} \rightarrow \text{Booleans} & & \\
\mathcal{F}(\nu n.t) & = & \mathcal{F}(t) \\
\mathcal{F}(t_1 \bowtie t_2) & = & \mathcal{F}(t_1) \wedge \mathcal{F}(t_2) \\
\mathcal{F}(t_1 \blacklozenge t_2) & = & \mathcal{F}(t_1) \wedge \mathcal{F}(t_2) \\
\mathcal{F}(\lightning) & = & \mathsf{True} \\
\mathcal{F}(e_1 \blacktriangle t_2) & = & \mathcal{F}(t_2) \\
\mathcal{F}(t_1 \blacktriangleright e_2) & = & \mathcal{F}(t_1) \\
\mathcal{F}(\_) & = & \mathsf{False}
\end{array}
$$

Figure 2.18: Failing observation on tasks

Internal value ($\blacksquare\, v$) can be used to output the value v as result. Unlike editors, it accepts no input, and thus the value of $\blacksquare\, v$ will always remain the same. Figure 2.19 gives the normalisation rules for fail and internal value.

$$\boxed{t, \sigma \Downarrow t', \sigma', \delta'}$$

$$\text{N-Fail}\ \frac{}{\lightning, \sigma \Downarrow \lightning, \sigma, \varnothing} \qquad \text{N-Done}\ \frac{}{\blacksquare\, v, \sigma \Downarrow \blacksquare\, v, \sigma, \varnothing}$$

Figure 2.19: Normalisation rules for fail and internal value

2.2.8 Step

The step combinator ($\blacktriangleright$) allows the result from one task to determine the next task. We call this *sequential composition*. The step combinator expects a task t of type $\textsc{Task}\ \tau_1$ on the left hand side, and a continuation e on the right hand side, which is a function from from τ_1 to a successor task of type $\textsc{Task}\ \tau_2$. Before we can define the normalisation rules of step, we need to have a way to determine the value of a task. For this, we introduce the task observation $\mathcal{V}$. Given a task $t : \textsc{Task}\ \tau$ and its current state σ, this function returns the task's value v of type τ. It is also possible that a task's value is undefined, in which case we write $\mathcal{V}(t, \sigma) = \bot$. The definition of $\mathcal{V}$ is given in Figure 2.20.

Steps are guarded. A step can only be taken if two conditions are met: (1) the task on the left-hand side has a value, and (2) the evaluation of the

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{V} : \text{Tasks} \times \text{States} \rightharpoonup \text{Values}} \\
\mathcal{V}(\nu n.t, \sigma) & = & \mathcal{V}(t, \sigma) \\
\mathcal{V}(\boxtimes^n \beta, \sigma) & = & \bot \\
\mathcal{V}(\square^n b, \sigma) & = & b \\
\mathcal{V}(\boxplus^n l, \sigma) & = & \sigma(l) \\
\mathcal{V}(\blacksquare\, v, \sigma) & = & v \\
\mathcal{V}(t_1 \bowtie t_2, \sigma) & = & \begin{cases} \langle v_1, v_2 \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
\mathcal{V}(t_1 \blacklozenge t_2, \sigma) & = & \begin{cases} v_1 & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \\ v_2 & \textbf{when } \mathcal{V}(t_1, \sigma) = \bot \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
\mathcal{V}(\lightning, \sigma) & = & \bot \\
\mathcal{V}(e_1 \blacktriangle t_2, \sigma) & = & v_2' \quad \textbf{when } \mathcal{V}(t_2, \sigma) = v_2 \wedge e_1\ v_2 \downarrow v_2' \\
\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) & = & \bot
\end{array}
$$

Figure 2.20: Value observation on tasks

continuation on the right-hand side with this value does not fail. The normalisation rules for step are given in Figure 2.21. There are three cases: either the first condition fails, and the step remains guarded (N-StepNone); or the first condition is met, but the second condition fails, and the step remains guarded (N-StepFail); or both conditions are met, and the step can proceed (N-StepCont). Note that the last two rules use evaluation in the host language to compute the successor task. The result of this evaluation is only used when the step can be taken successfully (N-StepCont), and discarded otherwise (N-StepFail).

Example 2.2 (Coffee machine). Consider the following task-program:

let coffeeMachine : Task Drink $= \nu n. \boxtimes^n \text{Int} \blacktriangleright \lambda x.$
  if $x \equiv 1$ **then** ■Coffee **else if** $x \equiv 2$ **then** ■Tea **else** $\lightning$

This program describes a coffee machine that can either serve coffee or tea. Coffee is served when one coin is inserted, and tea is served when two coins are inserted. For any other number of coins, the step remains guarded, which means that the coffee machine returns nothing and waits until a correct number of coins is inserted.

$$\boxed{t, \sigma \Downarrow t', \sigma', \delta'}$$

$$\text{N-StepNone}\ \frac{t_1, \sigma \Downarrow t_1', \sigma', \delta'}{t_1 \blacktriangleright e_2, \sigma \Downarrow t_1' \blacktriangleright e_2, \sigma', \delta'}\ \mathcal{V}(t_1', \sigma') = \bot$$

$$\text{N-StepFail}\ \frac{t_1, \sigma \Downarrow t_1', \sigma', \delta' \qquad e_2\ v_1 \downarrow t_2}{t_1 \blacktriangleright e_2, \sigma \Downarrow t_1' \blacktriangleright e_2, \sigma', \delta'}\ \mathcal{V}(t_1', \sigma') = v_1 \wedge \mathcal{F}(t_2)$$

$$\text{N-StepCont}\ \frac{t_1, \sigma \Downarrow t_1', \sigma', \delta' \qquad e_2\ v_1 \downarrow t_2 \qquad t_2, \sigma' \Downarrow t_2', \sigma'', \delta''}{t_1 \blacktriangleright e_2, \sigma \Downarrow t_2', \sigma'', \delta' \cup \delta''}\ \mathcal{V}(t_1', \sigma') = v_1 \wedge \neg\mathcal{F}(t_2)$$

Figure 2.21: Normalisation rules for step

2.2.9 Transform

The transform combinator (▲) maps a function over a task. It takes a function of type $\tau_1 \to \tau_2$ on the left-hand side, and a task of type Task τ_1 on the right-hand side, resulting in a task of type Task τ_2. The normalisation rule of the transform combinator, given in Figure 2.22, does not actually do anything. If we want to apply the function and use the result, we would need to use the transform combinator in combination with the step combinator. This would allow us to extract the value of the transform task by using the task observation $\mathcal{V}$, which in case of the transform combinator, is defined as $\mathcal{V}(e \blacktriangle t) = v'$ when $\mathcal{V}(t, \sigma) = v$ and $ev \downarrow v'$. So, if the task on the right-hand side has a value v, and applying the function e to v results in the value v', then the transform combinator has the value v'.

$$\boxed{t, \sigma \Downarrow t', \sigma', \delta'}$$

$$\text{N-Trans}\ \frac{t_2, \sigma \Downarrow t_2', \sigma', \delta'}{e_1 \blacktriangle t_2, \sigma \Downarrow e_1 \blacktriangle t_2', \sigma', \delta'}$$

Figure 2.22: Normalisation rule for transform

Example 2.3 (Traffic light). Let us consider a simple example:

let trafficLight : Task Light =
  ((λ*x*. **if** *x* **then** Green **else** Red) ▲ *vn*. ⊠^{n}Bool) ► λ*y*. ■*y*

This program describes a traffic light whose light is initially turned off, but given the right input, it can either become red or green. So long as no input

is given, the transform task on the left-hand side of the step combinator has no value, and the step remains guarded. Once an input is entered, transform returns the value Green if True was entered, and Red if False was entered, upon which the step proceeds and displays the result.

2.2.10 Parallel

Pairing (⋈) combines the result of two tasks, but only if both branches have a value. If the left task is of type τ_1, and the right task is of type τ_2, then their pairing is of type $\tau_1 \times \tau_2$. However, if one or both branches have no value, then the resulting task also has no value.

Choosing (♦) chooses one of two branches. This combinator is left-biased: it returns the leftmost task that has a value. If neither task has a value, then the resulting task also has no value. The normalisation rules for pairing and choice are given in Figure 2.23. See also Figure 2.20 for the definition of $\mathcal{V}$ for pairing and choice.

$$\boxed{t, \sigma \Downarrow t', \sigma', \delta'}$$

$$\text{N-ChooseLeft}\quad \frac{t_1, \sigma \Downarrow t_1', \sigma', \delta'}{t_1 \blacklozenge t_2, \sigma \Downarrow t_1', \sigma', \delta'}\ \mathcal{V}(t_1', \sigma') = v_1$$

$$\text{N-ChooseRight}\quad \frac{t_1, \sigma \Downarrow t_1', \sigma', \delta' \qquad t_2, \sigma' \Downarrow t_2', \sigma'', \delta''}{t_1 \blacklozenge t_2, \sigma \Downarrow t_2', \sigma'', \delta' \cup \delta''}\ \mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = v_2$$

$$\text{N-ChooseNone}\quad \frac{t_1, \sigma \Downarrow t_1', \sigma', \delta' \qquad t_2, \sigma' \Downarrow t_2', \sigma'', \delta''}{t_1 \blacklozenge t_2, \sigma \Downarrow t_1' \blacklozenge t_2', \sigma'', \delta' \cup \delta''}\ \mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = \bot$$

$$\text{N-Pair}\quad \frac{t_1, \sigma \Downarrow t_1', \sigma', \delta' \qquad t_2, \sigma' \Downarrow t_2', \sigma'', \delta''}{t_1 \bowtie t_2, \sigma \Downarrow t_1 \bowtie t_2', \sigma'', \delta' \cup \delta''}$$

Figure 2.23: Normalisation rules for pairing and choice

These combinators allow user to work on two tasks in parallel, but unlike the name suggests, parallel does not mean that there is non-determinism. The order of execution is determined by the order of user inputs send. Instead, parallel here means that the order in which we execute the tasks, and their subtasks, does not matter.

Example 2.4 (Breakfast). Let us consider the following task-program, which makes use of both parallel combinators:

```
let make : τ → Task τ = λx. νn. ⊠^n Unit ▶ λy. ■x in
let makeBreakfast : Task (Drink × Food) =
  ((make Tea ⧫ make Coffee) ⋈ make Egg) ▶ eatBreakfast
```

This program describes a simple workflow for making breakfast. Breakfast consists of something to drink (tea or coffee), and something to eat (eggs). The drink and the food are prepared in parallel (⋈), which means that the order in which they are made does not matter. For the drink, users have a choice (⧫) whether they want tea or coffee with their breakfast. For the food, users will always make an egg. We will use the function make to simulate that the user must first perform an action (i.e. send user input) before an item is prepared and the task has a value. Only when both the drink and food are ready, can the step be taken and can we enjoy our breakfast.

2.2.11 References

References model shared data sources in $\widehat{\text{TOP}}$. They provide a way for tasks to share information across control flow, and allow multiple users to simultaneously view or edit the same data. We have already seen shared editors, which allow users to modify a shared value, upon which the result immediately becomes visible to all other tasks interested in them. To create a reference to an expression e, we write **ref** e. Reference creation is a task, which upon normalisation, adds the reference to the state and results in a task whose value is the newly created location l pointing to the value b.

Reference assignment ($e_1 := e_2$) allows the system to assign a new value to a reference. It expects a location l of type Ref β on the left-hand side, and an expression of type β on the right-hand side. Upon normalisation, reference assignment saves the location's new value in the state and returns the unit value. Because the l's value has been changed, it also returns the set $\delta = \{l\}$. See Figure 2.24 for the normalisation rules of reference creation and assignment. This approach to references follows the one presented by Pierce [3], except that in our case, reference are lifted into the task domain.

$$\boxed{t, \sigma \;\Downarrow\; t', \sigma', \delta'}$$

$$\text{N-Share}\ \frac{l \notin \text{dom}(\sigma)}{\textbf{ref}\, b, \sigma \;\Downarrow\; \blacksquare l, [l \mapsto b]\sigma, \varnothing} \qquad \text{N-Assign}\ \frac{}{l := b, \sigma \;\Downarrow\; \blacksquare\langle\rangle, [l \mapsto b]\sigma, \{l\}}$$

Figure 2.24: Normalisation rules for references

There is still one problem that needs to be solved when using references. Consider the following example:

ref False ▶ λl : Ref Bool.
($\nu n.\boxplus^n\ l \blacktriangleright \lambda x$: Bool.**if** x **then** e **else** $\lightning$) ⋈ ($l :=$ True)

This program reduces to the following task after normalisation:

($\nu n.\boxplus^n\ l \blacktriangleright \lambda x$: Bool.**if** x **then** e **else** $\lightning$) ⋈ ■⟨⟩

where $\sigma = \{l \mapsto \mathsf{True}\}$. However, this task is not fully normalised. This happens because the normalisation rule N-Pair for pairing ($t_1 \bowtie t_2$) normalises its operands from left to right. Therefore, when the left task is normalised, the reference l is still set to False, and thus the step remains guarded. Only after normalisation of the left task is done, is the right task normalised, which sets the reference l to True. We would need to normalise the task-program a second time to fix this. To prevent this problem, we keep track of the set of references whose value has been changed while normalising or while handling input. This is the set δ that is returned after normalisation and input handling. There are two instances where this happens. Either the reference is updated by the system through reference assignment (N-Assign), or the reference is updated by the user through a shared editor (H-Change).

This motivates the fixing semantics presented in Figure 2.26. This semantics makes use of another task-observation function $\mathcal{W}$, which returns all references that are currently being watched by a shared editor inside a task. Its definition is given in Figure 2.25.

$$
\begin{array}{lcl}
\mathcal{W} : \text{Tasks} \to \mathbb{P}(\text{Locations}) & & \\
\mathcal{W}(\boxplus^n l) & = & \{l\} \\
\mathcal{W}(\nu n.t) & = & \mathcal{W}(t) \\
\mathcal{W}(t_1 \bowtie t_2) & = & \mathcal{W}(t_1) \cup \mathcal{W}(t_2) \\
\mathcal{W}(t_1 \blacklozenge t_2) & = & \mathcal{W}(t_1) \cup \mathcal{W}(t_2) \\
\mathcal{W}(e_1 \blacktriangle t_2) & = & \mathcal{W}(t_2) \\
\mathcal{W}(t_1 \blacktriangleright e_2) & = & \mathcal{W}(t_1) \\
\mathcal{W}(\_) & = & \varnothing
\end{array}
$$

Figure 2.25: Watching observation on tasks

Fixing rules are of the form $t, \sigma, \delta \Downarrow t', \sigma'$. They make sure that normalisation is applied until the task is truly reduced. When given a task, the current state, and the set δ, the fixing semantics will first apply normalisation on the given task and state. If it turns out that after normalisation, the resulting task watches a reference that has been changed meanwhile (either by the normalisation or by the initial set δ), then normalisation must be applied again. This is captured by the rule F-Loop. This process is repeated until the task is truly normalised, as determined by the rule F-Done. This rule also ensures that the resulting task t'' is in ν-standard form, which is

required before handling user input. Recall that the interaction semantics makes use of the fixing semantics. This way, the semantics of $\widehat{\text{TOP}}$ ensures that interaction with the environment can only take place after all updates to shared data sources are fully processed.

$$\boxed{t, \sigma, \delta \Downarrow t', \sigma'}$$

$$\text{F-Done}\quad \frac{t, \sigma \Downarrow t', \sigma', \delta' \qquad t' \equiv t''}{t, \sigma, \delta \Downarrow t'', \sigma'} \; (\delta \cup \delta') \cap \mathcal{W}(t') = \varnothing \wedge \mathcal{N}(t'')$$

$$\text{F-Loop}\quad \frac{t, \sigma \Downarrow t', \sigma', \delta' \qquad t', \sigma', \delta' \Downarrow t'', \sigma''}{t, \sigma, \delta \Downarrow t'', \sigma''} \; (\delta \cup \delta') \cap \mathcal{W}(t') \neq \varnothing$$

Figure 2.26: Fixing rules

2.3 Examples

This section will present two larger example-programs in $\widehat{\text{TOP}}$. Example 2.5 models a stopwatch, which shows how shared editors can be used to create timed tasks. It also shows an example of a higher-order task, and it demonstrates the use of the transform, step and choice combinators. Example 2.6 models a simple multi-user flight booking system, which shows how multiple users can work in parallel on the same shared data source. It demonstrates reference creation, assignment, and the step and parallel combinators.

Example 2.5 (Stopwatch). Shared editors can also be used to represent sensors or clocks. For example, we can represent the current time as a shared editor $vn.\boxplus^n\mathsf{time}$. While sensors and clocks are not explicitly modelled in $\widehat{\text{TOP}}$, we can assume that they exists as external users which periodically send update events to the system. By using shared editors as clocks, we could write a task-program that reacts to a timeout:

```
1  let wait : Int → Task τ → Task τ = λm. λt.
2    vn. ⊞ⁿtime ▶ λstart : Int.
3    vn. ⊞ⁿtime ▶ λnow : Int.
4      if now ≥ start + m then t else ↯ in
5  let stopwatch : Task String = ((λs. 1000 × s) ▲ vn. ⊠ⁿInt) ▶ λm.
6    (vn. ⊠ⁿ Unit ▶ λx.■ "Stopped") ♦ wait m (■ "Done")
```

The wait function (lines 1-4) is an example of a higher order task. It takes an integer m and a task t as arguments. The first step is immediately taken, so that the variable *start* holds the initial time (line 2). The next step will

remain guarded until m milliseconds have passed, after which the task t is returned (line 3-4).

We use this function to define a stopwatch in $\widehat{\text{TOP}}$. Suppose that the user should enter the number of seconds s for which the stopwatch should run. Because we defined wait on milliseconds, we use transform (▲) to convert seconds to milliseconds (line 5). After the user enters a value, the step is taken, which starts the task wait m (■"Done") (line 6). This task displays "Done" on the screen after m milliseconds, but so long as this task is still running, it has no value. We give the user a choice (⧫) to interrupt the stopwatch by sending an input event to the task on the left-hand side of the choice combinator (line 6). If done so before m milliseconds have passed, the left-hand side of the choice combinator will have a value before the right-hand side, and it displays "Stopped" on the screen.

Example 2.6 (Flight booking). We model a simple multi-user system where users can book a flight:

```
1  ref 42 ▶ λfreeSeats.
2  let validPassenger = λp. not (fst p ≡ "") ∧ snd p ≥ 18 in
3  let chooseSeats = λp. (νn. ⊠^n Int ⋈ νn. ⊞^n freeSeats) ▶ λ⟨s, fs⟩.
4    if fs ≥ s then freeSeats := fs − s; ■⟨p, s⟩ else ↯ in
5  let bookFlight = νn. ⊠^n(String × Int) ▶ λp.
6    if validPassenger p then chooseSeats p else ↯ in
7  bookFlight ⋈ bookFlight ⋈ bookFlight
```

When booking a flight, passengers should first enter their name and age into the system (line 5). Only when they enter a valid name and are at least 18 years old (line 2), are they allowed to proceed. Next, they have to choose how many tickets they want to buy. We create a shared reference *freeSeats* to keep track of how many seats are still available, and set its initial value to 42 (line 1). A user is only allowed to buy a certain amount of tickets if it does not exceed the number of tickets available. We can get the current value of *freeSeats* by using a shared editor. Because we want to get this value at the same moment as the user enters the amount of tickets he wants to buy, we set these two editors in parallel (line 3). If all went well, the system updates the value of *freeSeats*, and displays the passenger and the amount of tickets bought (line 4).

The parallel combinator (⋈) allows multiple bookFlight instances to run in parallel (line 7). This way, multiple users can book tickets at the same time. Their input events can interleave, and the order of execution is determined by the order of input events.

Chapter 3

Semantic equivalence in TopHat

In this chapter, we will examine when two $\widehat{\text{TOP}}$-programs are semantically equivalent. Let us first consider what it means in general for two programs to be semantically equivalent. According to Sewell, a "good" definition of semantic equivalence should satisfy the following properties [5]:

(i) Programs that result in observably-different values must not be equivalent.
(ii) Programs that terminate should not be equivalent to programs that do not terminate.
(iii) The relation $\cong$ should be an equivalence relation: it is reflexive, symmetric and transitive.
(iv) The relation $\cong$ should be a *congruence*, that is if $e_1 \cong e_2$, then for all contexts $C[-] : C[e_1] \cong C[e_2]$.
(v) $\cong$ should relate as many programs as possible subject to the above properties.

It should be obvious that the first three properties are desirable. The fourth property about congruence states that if two programs e_1 and e_2 are semantically equivalent, then we should be able to use e_1 and e_2 interchangeably within any program without changing its meaning. Finally, the last property ensures that $\cong$ is not just the empty relation.

We will keep these properties in mind when giving our definition of semantic equivalence. In Section 3.1, we will give a definition for the semantic equivalence of two expressions in the host language. The next section, Section 3.2, will look at the semantic equivalence of two tasks. Finally, Section 3.3 will present a set of properties that we believe hold true for $\widehat{\text{TOP}}$-programs with our definition. We will use the symbol $\simeq$ for the semantic equivalence of two expressions in the host language, and the symbol $\cong$ for the semantic equivalence of two tasks.

3.1 Expression equivalence

Before we will consider semantic equivalence of tasks, we will first look at semantic equivalence of expressions in the host language. We will start with an example. Consider the following two expressions:

$$e_1 := \lambda x : \textsc{Int}.\, \textbf{if}\, x < 0\, \textbf{then}\, -x\, \textbf{else}\, x$$
$$e_2 := \lambda y : \textsc{Int}.\, \textbf{if}\, y \geq 0\, \textbf{then}\, y\, \textbf{else}\, -y$$

It should be obvious that these two functions are equivalent: they both return the absolute value of their argument. Therefore, we should be able to use them interchangeably within any $\widehat{\text{TOP}}$-program without changing its behavior. So even though the functions e_1 and e_2 are different, we will never detect a difference between them when they are being used within a $\widehat{\text{TOP}}$-program, because for all possible arguments, e_1 and e_2 evaluate to the same result. So, when deciding if two expressions in the host language are equivalent, it is not enough to just look at the resulting value after evaluation. We need to consider all *contexts* that an expression can be used in. This leads to the definition of *contextual equivalence*. Pitts defines contextual equivalence informally as follows [4]:

> Two phrases of a programming language are *contextually equivalent* if any occurrences of the first phrase in a complete program can be replaced by the second phrase without affecting the observable results of executing the program.

This kind of equivalence is also called *operational*, or *observational* equivalence. To formally define such a notion of contextual equivalence for a given programming language, we must answer two questions: "What is a complete program?", and: "What are the observable results?". Depending on the answers to these two questions, this can result in different definitions of semantic equivalence for the same programming language [4].

For expressions in $\widehat{\text{TOP}}$, we answer these two questions as follows: we will consider an expression in the host-language a complete program if it does not contain any free variables, and the only observation we are interested in is the resulting value after evaluation. We also need a way to substitute an expression in a program by another. For this, we use the notion of a program context. A *context* $C[-]$ is a complete program that can contain "holes", denoted by the symbol '$-$', which can be filled. We write $C[e]$ for the expression that results from replacing all occurrences of $-$ in C by e. Figure 3.1 gives the context grammar for expressions. For the definition of expression equivalence, we will only quantify over all contexts of basic type β, because we can only observe the equivalence of two basic values. If we would allow all types, then we would have the same problem as introduced at the beginning of this section. Because then the context C can also be a

Contexts	C	::=	$- \mid \lambda x : \tau.\ e \mid C_1\ C_2$	– hole, abstraction, application
		|	$x \mid l \mid c$	– variable, location, constant
		|	$u\ C_1 \mid C_1\ o\ C_2$	– operations: unary, binary
		|	$\mathsf{if}\, C_1\, \mathsf{then}\, C_2\, \mathsf{else}\, C_3 \mid \langle\rangle$	– conditional, unit
		|	$\langle C_1, C_2\rangle \mid \mathsf{fst}\, C \mid \mathsf{snd}\, C$	– pair, projections
		|	P	– pretask
Pretasks	P	::=	$\nu n.C \mid \boxtimes^n \beta \mid \square^n C \mid \boxplus^n C$	– label allocation, editors
		|	$C_1 \bowtie C_2 \mid C_1 \blacklozenge C_2$	– pairing, choosing
		|	$\lightning \mid \blacksquare\, C$	– fail, internal value
		|	$C_1 \blacktriangle C_2 \mid C_1 \blacktriangleright C_2$	– transform, step
		|	$\mathsf{ref}\, C \mid C_1 := C_2$	– new shared value, assignment

Figure 3.1: Context grammar

lambda function, and we can only determine that two lambda functions are equivalent by considering all contexts that they can be used in. This leads to the following definition of expression equivalence:

Definition 3.1 (Expression equivalence). Given two expressions $e_1, e_2 : \tau$ (where $\tau \neq \textsc{Task}\ \tau'$ for some τ'), we say that e_1 and e_2 are semantically equivalent ($e_1 \simeq e_2$) if for all contexts $C[-] : \beta$, and for all values $b : \beta$:

$$C[e_1] \downarrow b \Leftrightarrow C[e_2] \downarrow b$$

Actually proving that two expressions are contextually equivalent is hard, as we would need to quantify over all contexts. That is, we would need to consider all possible ways that a program can use an expression. However, showing that two expressions e_1 and e_2 are contextually *in*equivalent is straightforward. All we have to do is find one context $C[-] : \beta$ such that $C[e_1] \downarrow b_1$ and $C[e_2] \downarrow b_2$ with $b_1 \neq b_2$.

Example 3.2 (Expression inequivalence). The expressions

$$e_1 := \lambda x : \textsc{Int}.\, \mathsf{if}\ x < 0\, \mathsf{then}\ 2\, \mathsf{else}\ 3$$
$$e_2 := \lambda x : \textsc{Int}.\, \mathsf{if}\ x > 0\, \mathsf{then}\ 3\, \mathsf{else}\ 2$$

are not contextually equivalent.
Proof: Take the context $C[-] : \textsc{Int}$ with $C[e] = e\,0$, then:

$$C[e_1] = (\lambda x : \textsc{Int}.\, \mathsf{if}\ x < 0\, \mathsf{then}\ 2\, \mathsf{else}\ 3)\, 0 \downarrow 3$$
$$C[e_2] = (\lambda x : \textsc{Int}.\, \mathsf{if}\ x > 0\, \mathsf{then}\ 3\, \mathsf{else}\ 2)\, 0 \downarrow 2$$

3.2 Task equivalence

For expression equivalence, we needed contexts to determine the equivalence of two lambda functions, whose results can only be observed after evaluation. For tasks however, we do not need contexts to view their results. A task's value can be determined at any point during execution, whereupon it either has a value, or it is undefined. On the other hand, tasks do allow user interaction, and depending on what inputs are send, the resulting task may be different. So while a lambda function can produce a different result depending on its arguments, so can a task produce different results depending on what inputs are send to it. So in a sense, for tasks, the "contexts" are user input.

The first property at the beginning of this chapter states that programs that result in observably different values must not be equivalent. Before we look at observations however, we should first fully normalise the tasks whose equivalence we want to determine. Normalisation keeps track of a state σ. For semantic equivalence, we do not want that normalisation results in two different states. So, given two tasks $t_1, t_2 : \textsc{Task}\ \tau$, we want that for all states σ, normalisation of t_1 and t_2 end in the same state σ':

$$t_1, \sigma, \varnothing \Downarrow t_1', \sigma' \Leftrightarrow t_2, \sigma, \varnothing \Downarrow t_2', \sigma'$$

We use the fixing semantics ($\Downarrow$) to ensure that the tasks are fully normalised. After normalisation, we need to decide what observations t_1' and t_2' must have in common for them to be considered equivalent. So let us recall what observations can be made on tasks. There is the value function $\mathcal{V}$ which returns the value v of a task, or $\bot$ if it is undefined; there is the failing function $\mathcal{F}$ which returns whether a task is failing; and there is the inputs function $\mathcal{I}$ which returns the set of all possible input events that a task accepts. The value and failing functions are used in the normalisation rules of $\widehat{\text{TOP}}$, and different observations for these functions can result in different derivation rules being triggered. Therefore, we can say that tasks for which the value or failing function return a different result must not be semantically equivalent.

Similarly, tasks whose inputs function $\mathcal{I}$ return a different set of input events can also not be semantically equivalent, because that would mean that the types of interaction that can be done with the tasks are different. Recall that by the fixing rules, presented in Figure 2.26, t_1' and t_2' should be in ν-standard form. Recall also that input events should be labeled by the same label as the editor it is meant for. So if we require that $\mathcal{I}(t_1') = \mathcal{I}(t_2')$, then this also implies that t_1' and t_2' must have the same label names for all extruded ν-expressions and their editors. If this is not possible by the congruence rules, then we can never have that $\mathcal{I}(t_1') = \mathcal{I}(t_2')$, and the tasks cannot be semantically equivalent.

Given these observations, we say that at least the following property must hold:

$$t \cong t' \Rightarrow \mathcal{F}(t) = \mathcal{F}(t') \wedge \mathcal{V}(t, \sigma) = \mathcal{V}(t', \sigma) \wedge \mathcal{I}(t) = \mathcal{I}(t')$$

For any of these task observations, we can distinguish two cases: either a task fails or does not fail, it either has a value or its value is undefined, and it either accepts input or it accepts no more input. Based on these case distinctions, we will say that a task is in either one of five states after normalisation. These task states, and some examples, are shown in Figure 3.2. The next subsections will describe each task state into more detail.

$\mathcal{F}$	$\mathcal{V}$	$\mathcal{I}$	**Task state**	**Examples**
✓	–	–	Failing	$\lightning,\ \lightning \bowtie \lightning,\ \lightning \blacklozenge \lightning,\ \lambda x.x \blacktriangle \lightning,\ \lightning \blacktriangleright \lambda x.\blacksquare x$
–	✓	–	Finished (stable)	$\blacksquare 2,\ \blacksquare 2 \bowtie \blacksquare 3,\ \blacksquare\langle 2,3\rangle,\ \lambda x.x+1 \blacktriangle \blacksquare 2$
–	✓	✓	Finished (unstable)	$\nu n.\,\Box^n 2,\ \nu n.\,\Box^n 2 \bowtie \nu n.\,\Box^n 3,\ \nu n.\,\Box^n\langle 2,3\rangle$
–	–	–	Stuck	$\blacksquare 2 \blacktriangleright \lambda x.\lightning,\ \blacksquare 2 \bowtie \lightning,\ \lightning \bowtie \blacksquare 2$
–	–	✓	Running (looping)	$\nu n.\ \boxtimes^n \textsc{Int} \bowtie \lightning,\ \nu n.\,\Box^n 2 \blacktriangleright \lambda x.\lightning$
–	–	✓	Running (branching)	$\nu n.\ \boxtimes^n \textsc{Int} \blacktriangleright \lambda x.\,\textsf{if}\ x \leq 2\ \textsf{then}\ \blacksquare x\ \textsf{else}\ \lightning$

Figure 3.2: Different states a task can be in after normalisation. Checkmarks (✓) for $\mathcal{F}$, $\mathcal{V}$ and $\mathcal{I}$ for a task t and a state σ indicate that $\mathcal{F}(t)$, $\mathcal{V}(t,\sigma) = v$ for $v \neq \bot$, and $\mathcal{I}(t) \neq \varnothing$ respectively.

3.2.1 Failing tasks

A *failing* task t is a task for which the failing function $\mathcal{F}(t)$ yields true. In the original $\widehat{\text{TOP}}$ paper, theorem 6.5 states that a task fails if and only if it accepts no more user input [6]. However, with the introduction of internal value in *TopHat Next* [7], this is no longer the case, because $\blacksquare e$ does not fail, and neither does it accept user input. What we can still say however is that if a task is failing, then we know that it has no value and accepts no more user input (Conjecture 3.4).

Definition 3.3 (FAILING TASK). We call a task $t : \textsc{Task}\ \tau$ *failing* iff $\mathcal{F}(t)$.

Conjecture 3.4. For all failing tasks t and states σ, we have that $\mathcal{V}(t,\sigma) = \bot$ and $\mathcal{I}(t) = \varnothing$.

Once a task fails, it will always remain failing, because by Conjecture 3.4, no more user interaction is possible, and by assumption, the task is already fully normalised. Failing can thus be regarded as one type of termination, and we will consider all tasks that fail to be equivalent. Hence, we will say that the tasks $\lightning,\ \lightning \bowtie \lightning,\ \lightning \blacklozenge \lightning,\ \lambda x.x \blacktriangle \lightning,\ \lightning \blacktriangleright \lambda x.\blacksquare x$, and all other failing tasks, are semantically equivalent to each other.

3.2.2 Finished tasks

A *finished* task t is a task which yields a value $\mathcal{V}(t, \sigma) = v$ for $v \neq \bot$. This value can either be *stable* when no more user input is possible, or *unstable* when the task still accepts user input. An example of a finished task with a stable value is the task $\blacksquare 42$. Because this task accepts no more user input, its value will always remain equal to 42. An example of a finished task with an unstable value is the task $\nu n.\, \square^n 42$. This task still accepts user input, and thus its value can keep on changing. Even though both tasks yield the same value, they should not be equivalent, since one's value can be changed and the other one's value cannot.

Definition 3.5 (FINISHED TASK). We call a task $t : \textsc{Task}\ \tau$ in state σ *finished* iff $\mathcal{V}(t, \sigma) = v$, for $v \neq \bot$. Furthermore, we call a finished task *stable* iff $\mathcal{I}(t) = \varnothing$, and *unstable* iff $\mathcal{I}(t) \neq \varnothing$.

A stable task can thus never be semantically equivalent to an unstable task, even if their values are (initially) the same. But just looking at the resulting values, and whether the resulting value is stable or not, is not enough to determine semantic equivalence of finished tasks. Consider for example the following tasks:

$$t_1 := \blacksquare\langle 2, 3\rangle \qquad t_3 := \nu n.\, \square^n \langle 2, 3\rangle$$
$$t_2 := \blacksquare\, 2 \bowtie \blacksquare\, 3 \qquad t_4 := \nu n.\, \square^n 2 \bowtie \nu n.\, \square^n 3$$

These are all finished tasks with value $\langle 2, 3\rangle$. Tasks t_1 and t_2 are both stable, and thus $t_1 \cong t_2$. We have already concluded that stable tasks cannot be equivalent to unstable tasks, so $t_1 \ncong t_3$, $t_1 \ncong t_4$, $t_2 \ncong t_3$, and $t_2 \ncong t_4$. But we will also say that $t_3 \ncong t_4$, because we have that $t_4 \equiv \nu n.\, \nu m.\, \square^n 2 \bowtie \square^m 3$ by ν-congruence, so:

$$\mathcal{I}(t_3) = \{E^n b \mid b : \textsc{Int} \times \textsc{Int}\}$$
$$\mathcal{I}(t_4) = \{E^n b \mid b : \textsc{Int}\} \cup \{E^m b \mid b : \textsc{Int}\}$$

Meaning that the types of interaction that can be done with t_3 differ from the types of interaction that can be done with t_4. In the case of t_3, the user can only alter the pair in one go, whereas for t_4, the user can partially update it. So for finished tasks, we also need to require that $\mathcal{I}(t) = \mathcal{I}(t')$ if we want to conclude that $t \cong t'$.

And yet this is still not enough. Consider the following two tasks:

$$t_5 := (\lambda x.\, \textbf{if}\ x < 0\ \textbf{then}\ -x\ \textbf{else}\ x) \blacktriangle \nu n.\, \square^n 42$$
$$t_6 := \nu n.\, \square^n 42$$

Both tasks have the value 42 and accept the same input. However for negative values, the resulting values diverge, because the transform function in

task t_5 ensures that its output is always positive. So if a task still accepts user input, not only do we need to look at a task's current value, but we also need to consider all values that a task can have after user interaction. We claim that a finished task will always remain finished with the same input space, only its value may change (Conjecture 3.6).

Conjecture 3.6. If t is a finished task, then for all inputs $i \in \mathcal{I}(t)$: if $t, \sigma \overset{i}{\Rightarrow} t', \sigma'$, then t' is again a finished task. Moreover, if we do not allow α-conversion for ν-expressions, then we also have that $\mathcal{I}(t') = \mathcal{I}(t)$.

3.2.3 Stuck tasks

A *stuck* task t is a task which does not fail, does not have a value, and does not accept user input. Such tasks are essentially "broken". Examples of stuck tasks are $\blacksquare\, 2 \blacktriangleright \lambda x.\, \lightning$, $\blacksquare\, 2 \bowtie \lightning$, and $\lightning \bowtie \blacksquare\, 2$. The first example is stuck because the right-hand side always fails, and thus the step can never be taken. For pairing, we have that both sides must fail before $\bowtie$ fails, and both sides must have a value before $\bowtie$ has a value. So, if one side fails and the other side has no value, then neither observation is true, and the task is stuck.

Definition 3.7 (Stuck task). We call a task $t : \textsc{Task}\ \tau$ in state σ *stuck* iff $\neg\mathcal{F}(T)$, $\mathcal{V}(t, \sigma) = \bot$, and $\mathcal{I}(t) = \varnothing$.

A stuck task will always remain stuck, because no more user interaction is possible, and by assumption it is already fully normalised. Similar to failing, we will consider stuck tasks as another type of termination, and we will say that all stuck tasks are semantically equivalent to each other.

3.2.4 Running tasks

A *running* task t is a task which does not fail, does not have a value, but still accepts user input. Because there is still user interaction possible, it may be the case that with the right input, it transitions to one of the previously described task states. The simplest example of a running task is the empty editor $\nu n.\ \boxtimes^n \beta$, which becomes a finished (unstable) task once it receives a valid input event. There also exist running tasks that only transition to a another task state for some inputs, or for no inputs at all. For example, the tasks $\nu n.\ \boxtimes^n \textsc{Int} \blacktriangleright \lambda x.\, \lightning$, $\nu n.\ \boxtimes^n \bowtie \lightning$, and $\lightning \bowtie \nu n.\, \square^n 2$ are all running tasks which will forever remain running; and the task $\nu n.\ \boxtimes^n \textsc{Int} \blacktriangleright \lambda x.\, \mathbf{if}\ x \leq 2\ \mathbf{then}\ \blacksquare\, x\ \mathbf{else}\ \lightning$ will only transition to another task state for some inputs, but not for others.

Definition 3.8 (Running task). We call a task $t : \textsc{Task}\ \tau$ in state σ *running* iff $\neg\mathcal{F}(T)$, $\mathcal{V}(t, \sigma) = \bot$, and $\mathcal{I}(t) \neq \varnothing$.

To determine the equivalence of two running tasks, we therefore need to look at all possible user interactions, and check that they affect the two tasks in the same way. To do this, we need to have a way to talk about sequences of input events, instead of just single input events. We give the following definition for this:

Definition 3.9 (Input sequences). An input sequence $I = i_0, ..., i_n$ is a finite sequence of input events. Given a task $t_0 : \text{Task } \tau$ and a state σ_0, we say that I is a *valid* input sequence for t_0, σ_0 iff:

$$t_0, \sigma_0 \overset{i_0}{\Rightarrow} t_1, \sigma_1 \overset{i_1}{\Rightarrow} \ldots \overset{i_n}{\Rightarrow} t_{n+1}, \sigma_{n+1}$$

with $i_j \in \mathcal{I}(t_j)$, for all $j \in \{0, ..., n\}$. We will use the shorthand notation $t_0, \sigma_0 \overset{I}{\Rightarrow}{}^* t_{n+1}, \sigma_{n+1}$ to denote the above derivation. We also consider the empty input sequence, denoted by Λ, to be a valid input sequence, and for all tasks t and states σ we have that: $t, \sigma \overset{\Lambda}{\Rightarrow}{}^* t, \sigma$.

We will make a distinction between running tasks that forever remain running, no matter what inputs you send to it; and running tasks for which there exists at least one input sequence which "escapes", i.e. which transitions to another task state. We will call the former class *looping* tasks, and the latter class *branching* tasks, formally:

Definition 3.10 (Looping and branching). We call a running task $t : \text{Task } \tau$ in state σ *looping* iff for all valid input sequences I: $t, \sigma \overset{I}{\Rightarrow}{}^* t', \sigma'$ and t' is again a running task. If there exists at least one valid input sequence for which t' is not a running task, then we call t *branching*.

For branching tasks, it is possible to transition to either a finished or a stuck task state. Take for example the task $\nu n.\ \boxtimes^n \text{Int} \blacktriangleright \lambda x.t$, which is a running task that transitions to t after a valid input event. So long as t is not failing, the step can be taken, and so t can be any non-failing task. It is also possible that for some input events, it transitions to one task state, and for others, that it transitions to a different task state. For example the task $\nu n.\ \boxtimes^n \text{Int} \blacktriangleright \lambda x.\ \textbf{if } x \leq 2 \textbf{ then } t \textbf{ else } t'$ transitions to t for some inputs, and to t' for other inputs. We will claim that a running task can never transition to a failing task (Conjecture 3.11).

Conjecture 3.11. If $t : \text{Task } \tau$ is a running task, then for all states σ and for all valid input sequences I, if $t, \sigma \overset{I}{\Rightarrow}{}^* t', \sigma'$, then $\neg\mathcal{F}(t')$.

We say that two running tasks t_1 and t_2 in state σ are semantically equivalent if for all valid input sequences I: $t_1, \sigma \overset{I}{\Rightarrow}{}^* t_1', \sigma' \Leftrightarrow t_2, \sigma \overset{I}{\Rightarrow}{}^* t_2', \sigma'$, with $\mathcal{V}(t_1', \sigma') = \mathcal{V}(t_2', \sigma')$, and $\mathcal{I}(t_1') = \mathcal{I}(t_2')$. That is, for all possible user interactions, t_1 and t_2 are not observably different. Because of Conjecture 3.11, we do not need to check whether the tasks fail or not.

3.2.5 Task equivalence

Figure 3.3 shows all possible task states and their transitions. In this diagram, a looping task is a task which never leaves the running state, i.e. which always takes the transition back to the running state, no matter what input is given. A branching task is a running task for which there exists at least one input sequence which will transition to either stuck, finished (stable), or finished (unstable). We claim that this state diagram is correct (Conjecture 3.12).

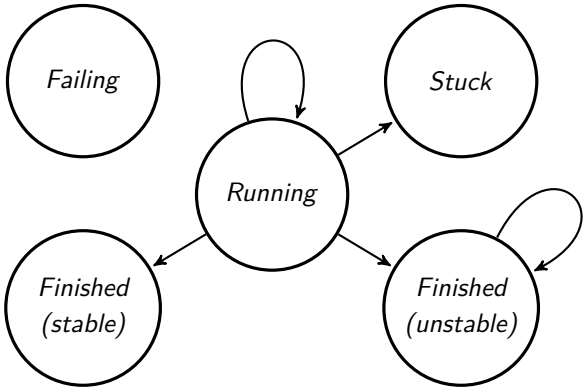

Figure 3.3: Possible task states and their transitions, where a transition indicates user interaction($\overset{i}{\Rightarrow}$) for some input i.

Conjecture 3.12. The state diagram in Figure 3.3 is correct. That is, for any two task states S and S', there is a transition from S to S' iff there exist two tasks $t \in S$ and $t' \in S'$ such that $t, \sigma \overset{i}{\Rightarrow} t', \sigma'$ for some input $i \in \mathcal{I}(t)$ and states σ and σ'.

In Section 3.2.1 on failing tasks, we noted that with the addition of internal value (■) in *TopHat Next* [7], that it is no longer the case that a task t is failing iff it accepts no more user input, as is shown by theorem 6.5 in the original $\widehat{\text{TOP}}$ paper [6]. Were this theorem still true, then we could not have the stuck and finished (stable) states, because they contain tasks that do not fail and do not accept user input. We will therefore claim if we remove ■ from the $\widehat{\text{TOP}}$ language as presented here, then we no longer have the stuck and finished (stable) states (Conjecture 3.13).

Conjecture 3.13. If we remove internal value (■) from $\widehat{\text{TOP}}$ then we are left with only three possible task states after normalisation: failing, running and finished (unstable).

Based on the five task states, we give the following definition for the semantic equivalence of two tasks:

Definition 3.14 (Task equivalence). Given two tasks $t_1, t_2 : \textsc{Task}\ \tau$, we say that t_1 and t_2 are semantically equivalent ($t_1 \cong t_2$) if for all states σ:

$$t_1, \sigma, \varnothing \Downarrow t_1', \sigma' \;\Leftrightarrow\; t_2, \sigma, \varnothing \Downarrow t_2', \sigma'$$

and for t_1' and t_2' one of the following holds:

(1) both tasks are *failing*;
(2) both tasks are *finished* with $\mathcal{V}(t_1', \sigma') = \mathcal{V}(t_2', \sigma')$, and this value is *stable*.
(3) both tasks are *finished* with $\mathcal{V}(t_1', \sigma') = \mathcal{V}(t_2', \sigma')$, and this value is *unstable*. Furthermore, for all valid input sequences I:

$$t_1', \sigma' \overset{I}{\Rightarrow}{}^* t_1'', \sigma'' \;\Leftrightarrow\; t_2', \sigma' \overset{I}{\Rightarrow}{}^* t_2'', \sigma''$$

with $\mathcal{V}(t_1'', \sigma'') = \mathcal{V}(t_2'', \sigma'')$;
(4) both tasks are *stuck*;
(5) both tasks are *running*, and for all valid input sequences I:

$$t_1', \sigma' \overset{I}{\Rightarrow}{}^* t_1'', \sigma'' \;\Leftrightarrow\; t_2', \sigma' \overset{I}{\Rightarrow}{}^* t_2'', \sigma''$$

with $\mathcal{V}(t_1'', \sigma'') = \mathcal{V}(t_2'', \sigma'') \wedge \mathcal{I}(t_1'') = \mathcal{I}(t_2'')$.

To determine a task's state, we use the task observation functions $\mathcal{F}$, $\mathcal{V}$, and $\mathcal{I}$. For some task states, we also need to take the interactive setting of $\widehat{\text{TOP}}$ into account. Namely, if the set of inputs given by $\mathcal{I}$ is not empty, as is the case for the finished (unstable) and running task states, then we also need to check the task observations after every possible input sequence. Because we also allow input sequences to be empty, we believe that we can generalize the above definition as follows:

Definition 3.15 (Task equivalence). Given two tasks $t_1, t_2 : \textsc{Task}\ \tau$, we say that t_1 and t_2 are semantically equivalent ($t_1 \cong t_2$) if for all states σ:

$$t_1, \sigma, \varnothing \Downarrow t_1', \sigma' \;\Leftrightarrow\; t_2, \sigma, \varnothing \Downarrow t_2', \sigma'$$

and for all valid input sequences I:

$$t_1', \sigma' \overset{I}{\Rightarrow}{}^* t_1'', \sigma'' \;\Leftrightarrow\; t_2', \sigma' \overset{I}{\Rightarrow}{}^* t_2'', \sigma''$$

with $\mathcal{F}(t_1'') = \mathcal{F}(t_2'') \wedge \mathcal{V}(t_1'', \sigma'') = \mathcal{V}(t_2'', \sigma'') \wedge \mathcal{I}(t_1'') = \mathcal{I}(t_2'')$.

Conjecture 3.16. Definition 3.15 is equivalent to Definition 3.14. That is, they describe the same relation:

$$t_1 \cong t_2 \text{ by Definition 3.14} \;\Leftrightarrow\; t_1 \cong t_2 \text{ by Definition 3.15}$$

3.3 Properties

Now that we have a definition of semantic equivalence for $\widehat{\text{TOP}}$-programs, we can look at some interesting properties. We will not give formal proofs in this section. At most, we will give some informal argumentation of why we think a certain equality holds, or we will provide a counterexample to show the inequality of two expressions.

3.3.1 Transforming (functor)

A functor in category theory and functional programming is an object that can be mapped over. In $\widehat{\text{TOP}}$, we can map a function over a task by using the transform ($\blacktriangle$) combinator. For an object F to be qualified as a functor, it needs to satisfy two laws: (1) the *identity law*, which says that if we map the identity function over F, then the result is again F; and (2) the *composition law*, which says that mapping the composition of two functions over F is the same as mapping the two functions over F in a row. In $\widehat{\text{TOP}}$, we can express these laws as follows:

$$\begin{aligned} \mathsf{id} \blacktriangle t &\cong t && \text{(identity)} \\ (e_1 \circ e_2) \blacktriangle t &\cong e_1 \blacktriangle (e_2 \blacktriangle t) && \text{(composition)} \end{aligned}$$

We think that these laws hold with our definition of semantic equivalence. We argue informally that this is true for the identity law. Given any task t and state σ, we have that $\mathcal{F}(\mathsf{id} \blacktriangle t) = \mathcal{F}(t)$ by definition (see Figure 2.18); $\mathcal{I}(\mathsf{id} \blacktriangle t) = \mathcal{I}(t)$ by definition (see Figure 2.15); and $\mathcal{V}(\mathsf{id} \blacktriangle t, \sigma) = \mathcal{V}(t, \sigma)$, because if t has no value, then neither does $\mathsf{id} \blacktriangle t$, and if $\mathcal{V}(t, \sigma) = v$ for $v \neq \bot$, then $\mathcal{V}(\mathsf{id} \blacktriangle t, \sigma) = \mathsf{id}\, v \downarrow v$. A similar argumentation can be made for the composition law. We therefore believe that the Task type former in $\widehat{\text{TOP}}$ is a functor.

3.3.2 Pairing (applicative/monoidal functor)

An applicative functor is a construct in functional programming, which is always also a functor. Traditionally, it needs an apply function which, given the functors $F\ (a \to b)$ and $F\ a$, returns $F\ b$. This function can also be expressed differently if we have a mapping and pairing function over F. In $\widehat{\text{TOP}}$, these are transform ($\blacktriangle$) and pairing ($\bowtie$). The laws that an applicative functor must satisfy can be written in $\widehat{\text{TOP}}$ as follows:

$$\begin{aligned} \blacksquare\langle\rangle \bowtie t &\ncong t && \text{(left identity)} \\ t \bowtie \blacksquare\langle\rangle &\ncong t && \text{(right identity)} \\ \mathsf{assoc} \blacktriangle (t_1 \bowtie (t_2 \bowtie t_3)) &\cong (t_1 \bowtie t_2) \bowtie t_3 && \text{(associativity)} \\ (\lambda\langle x_1, x_2\rangle.\langle e_1\ x_1, e_2\ x_2\rangle) \blacktriangle (t_1 \bowtie t_2) &\cong (e_1 \blacktriangle t_1) \bowtie (e_2 \blacktriangle t_2) && \text{(naturality)} \end{aligned}$$

However, if we take $t = \lightning$ for the left and right identity laws, then in both cases we have that the right-hand side is a failing task, but the left-hand side is not, because pairing requires that both sides fail before it fails itself. Therefore, we will say that the Task operation on types is not an applicative functor. We do however believe that the other two laws hold. For associativity, we need the function $\mathsf{assoc} := \lambda\langle a, \langle b, c\rangle\rangle.\langle\langle a, b\rangle, c\rangle$ to make sure that both sides have the same type. Lastly, the naturality law states that first pairing two tasks and then mapping two functions is the same as first mapping the functions separately and then pairing them.

3.3.3 Choosing

For choosing ($\blacklozenge$) we define the following properties:

$$
\begin{array}{rcll}
t_1 \blacklozenge (t_2 \blacklozenge t_3) & \cong & (t_1 \blacklozenge t_2) \blacklozenge t_3 & \text{(associativity)} \\
\blacksquare\, e \blacklozenge t & \cong & \blacksquare\, e & \text{(left catch)} \\
t \blacklozenge \blacksquare\, e & \ncong & \blacksquare\, e & \text{(right catch)} \\
 & & & \\
e \blacktriangle (t_1 \blacklozenge t_2) & \cong & (e \blacktriangle t_1) \blacklozenge (e \blacktriangle t_2) & \text{(distributivity)} \\
 & & & \\
t_0 \bowtie (t_1 \blacklozenge t_2) & \ncong & (t_0 \blacklozenge t_1) \bowtie (t_0 \blacklozenge t_2) & \text{(left pair distributivity)} \\
(t_1 \blacklozenge t_2) \bowtie t_0 & \ncong & (t_1 \blacklozenge t_0) \bowtie (t_2 \blacklozenge t_0) & \text{(right pair distributivity)} \\
 & & & \\
t_0 \blacktriangleright \lambda x.(t_1 \blacklozenge t_2) & \cong & (t_0 \blacktriangleright \lambda x.t_1) \blacklozenge (t_0 \blacktriangleright \lambda x.t_2) & \text{(left step distributivity)} \\
(t_1 \blacklozenge t_2) \blacktriangleright e & \cong & (t_1 \blacktriangleright e) \blacklozenge (t_2 \blacktriangleright e) & \text{(right step distributivity)}
\end{array}
$$

So we have associativity, a number of distributivity laws, and left and right catch. Catch only holds for the left side, because the choice combinator is left-biased. The first distributivity law shows that mapping a function over choice should be equivalent to mapping it separately over its component subtasks. The last two distributivity laws for step ($\blacktriangleright$) show that stepping to or from a choice is equivalent to choosing between steps.

We believe left and right distributivity for pairing ($\bowtie$) do not hold. Suppose $\mathcal{V}(t_0, \sigma) = \bot$, $\mathcal{V}(t_1, \sigma) = v_1$ and $\mathcal{V}(t_2, \sigma) = v_2$ for $v_1 \neq \bot$ and $v_2 \neq \bot$. Then in both cases, we have that the left-hand side of the inequality has no value, because pairing requires that both sides have a value before it has a value itself. However, the right-hand side does have a value in both cases, namely $\langle v_1, v_2\rangle$, because the choice combinator normalises t_0 away, and we are left with $t_1 \bowtie t_2$ in both cases.

3.3.4 Failing

When considering the failing task ($\lightning$), we can write down the following (in)equalities:

$$\begin{aligned}
\lightning \blacklozenge t &\cong t && \text{(left identity)}\\
t \blacklozenge \lightning &\cong t && \text{(right identity)}\\
\\
e \blacktriangle \lightning &\cong \lightning && \text{(annihilation)}\\
\\
\lightning \bowtie t &\ncong \lightning && \text{(left pair annihilation)}\\
t \bowtie \lightning &\ncong \lightning && \text{(right pair annihilation)}\\
\\
\lightning \blacktriangleright e &\cong \lightning && \text{(left step annihilation)}\\
t \blacktriangleright \lambda x.\, \lightning &\ncong \lightning && \text{(right step annihilation)}
\end{aligned}$$

We define the left and right identity for the choice combinator ($\blacklozenge$), which say that $\lightning$ can be cancelled out from the left- and right-hand side. For pairing, we will say that left and right pair annihilation do not hold, because $\mathcal{F}(t_1 \bowtie t_2) = \mathcal{F}(t_1) \wedge \mathcal{F}(t_2)$. Pairing therefore requires that both the left-hand and right-hand side fail before their pairing fails, and thus if only one side fails, it cannot be equivalent to the failing task $\lightning$.

We defined all failing tasks to be semantically equivalent, thus annihilation for $\blacktriangle$, and left step annihilation for $\blacktriangleright$ trivially hold. Right step annihilation does not however. Suppose that t is not a failing task, then we also have that $t \blacktriangleright \lambda x.\, \lightning$ is not a failing task, because $\mathcal{F}(t \blacktriangleright \lambda x.\, \lightning) = \mathcal{F}(t)$ by definition (see Figure 2.18). Neither can we normalise $t \blacktriangleright \lambda x.\, \lightning$, because the right-hand side fails (N-StepFail). So if t does not fail, then neither does $t \blacktriangleright \lambda x.\, \lightning$, and we cannot conclude that a non-failing is semantically equivalent to the failing task $\lightning$.

3.3.5 Stepping (monad)

A monad is a construct in category theory and functional programming. It consists of a type constructor M; a return function that wraps any value a into the monadic value $M\,a$; and a bind function that extracts the value a from the monad $M\,a$, and uses it to produce a new monad $M\,b$. Furthermore, for M to be considered a monad, it must satisfy three laws: (1) return is a *left identity* for bind, (2) return is a *right identity* for bind, and (3) bind is *associative*.

Steps ($\blacktriangleright$) in $\widehat{\text{TOP}}$ have a monadic flavor to them, and we can wonder whether the step combinator is a bind operation. So, if we consider Task to

be the monadic constructor, $\blacksquare$ the return function, and $\blacktriangleright$ the bind function, then we can express the three monadic laws in $\widehat{\text{TOP}}$ as follows:

$$\begin{array}{rcll} \blacksquare\, x \blacktriangleright g & \not\cong & g\ x & \text{(left identity)} \\ t \blacktriangleright (\lambda y.\, \blacksquare\, y) & \not\cong & t & \text{(right identity)} \\ (t \blacktriangleright g) \blacktriangleright h & \cong & t \blacktriangleright (\lambda y.g\ y \blacktriangleright h) & \text{(associativity)} \end{array}$$

Unfortunately, with our definition of program equivalence, we can show that neither the left nor the right identity laws hold. For the left identity, take for example $g = \lambda x.\, \lightning$, then we have that $g\ x$ is a failing task, because $g\ x = (\lambda x.\, \lightning)\ x \downarrow \lightning$. However, the left-hand side $\blacksquare\, x \blacktriangleright g = \blacksquare\, x \blacktriangleright \lambda x.\, \lightning$ is a stuck task, because it does not fail, and the step can never be taken.

For the right identity law we can take for example $t = \nu n.\ \boxtimes^n \textsc{Int}$. If we send, for example, the input event $E^n 42$ to both sides, then the left-hand side normalises to $\blacksquare\, 42$, while the right-hand side becomes $\nu n.\, \square^n 42$. As we have already seen, these two tasks are not semantically equivalent, because the value of $\blacksquare\, 42$ is stable and cannot be changed anymore, while the value of $\nu n.\, \square^n 42$ is unstable and can be updated through user input.

We argue informally that the associativity law holds. Given any task t and state σ, for $\mathcal{F}$ and $\mathcal{I}$ we have that:

$$\begin{aligned} \mathcal{F}((t \blacktriangleright g) \blacktriangleright h) &= \mathcal{F}(t \blacktriangleright g) = \mathcal{F}(t) = \mathcal{F}(t \blacktriangleright (\lambda y.g\ y \blacktriangleright h)) \\ \mathcal{I}((t \blacktriangleright g) \blacktriangleright h) &= \mathcal{I}(t \blacktriangleright g) = \mathcal{I}(t) = \mathcal{I}(t \blacktriangleright (\lambda y.g\ y \blacktriangleright h)) \end{aligned}$$

by definition (see Figure 2.18 and Figure 2.15). Furthermore, supposing that $\mathcal{V}(t, \sigma) = v$ for $v \neq \bot$, and $g\ v \downarrow t'$ with $\neg\mathcal{F}(t')$, then both sides normalise to $t' \blacktriangleright h, \sigma'$. On the other hand, if either $\mathcal{V}(t, \sigma) = \bot$ or $\mathcal{F}(t')$, then the step cannot be taken, and we have that $\mathcal{V}((t \blacktriangleright g) \blacktriangleright h) = \mathcal{V}(t \blacktriangleright (\lambda y.g\ y \blacktriangleright h)) = \bot$.

Chapter 4

Conclusion

In this thesis, we looked at the formal language $\widehat{\text{TOP}}$, which defines an operational semantics for reasoning about task-oriented programs. In Chapter 3, we gave a definition for the semantic equivalence of two $\widehat{\text{TOP}}$-programs. We split this definition into two classes: expression equivalence, and task equivalence. For task equivalence, we showed that a task can be in either one of five states after normalisation, and for every two tasks in the same state, we defined what it means for them to be semantically equivalent. We also noted that for task states that still accept user input, it is important to take the interactive setting of $\widehat{\text{TOP}}$ into account, and compare how both tasks react to user input. Finally, in Section 3.3, we presented a set of properties that we believe hold true for $\widehat{\text{TOP}}$-programs. We showed that the Task type former in $\widehat{\text{TOP}}$ is neither a monad nor an applicative functor, and although we did not prove it formally, we speculated that it is a functor.

4.1 Future work

We presented a number of conjectures whose proofs were out of scope for this thesis. To get more confidence in our results, it would be nice to prove these conjectures formally. It would also be nice to prove formally that Definition 3.14 satisfies the five desired properties of semantic equivalence presented at the beginning of Chapter 3. Likewise, the properties in Section 3.3 are missing formal proofs.

For our analysis of $\widehat{\text{TOP}}$, we left out some task constructs from the $\widehat{\text{TOP}}$-language as presented in *TopHat Next* [7]. We also noted in Section 3.2.1 that the addition of internal value (■) in *TopHat Next* changed some properties of $\widehat{\text{TOP}}$ as it was originally presented. Furthermore, we speculated in Conjecture 3.13 that without ■, we are left with only three task states after normalisation. It might therefore also be interesting to research what effects adding or removing certain task constructs has on our definition of semantic equivalence, and the properties presented in Section 3.3.

Bibliography

[1] Peter Achten, Pieter W. M. Koopman, and Rinus Plasmeijer. An introduction to task oriented programming. In Viktória Zsók, Zoltán Horváth, and Lehel Csató, editors, *Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, volume 8606 of *Lecture Notes in Computer Science*, pages 187–245. Springer, 2013.

[2] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 1 edition, 2012.

[3] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[4] Andrew M. Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000.

[5] Peter Sewell. Semantics of programming languages, computer science tripos, part 1b. `https://www.cl.cam.ac.uk/teaching/1718/Semantics/notes.pdf`, 2017. [Accessed 29-November-2019].

[6] Tim Steenvoorden, Nico Naus, and Markus Klinik. Tophat: A formal foundation for task-oriented programming. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 17:1–17:13. ACM, 2019.

[7] Tim Steenvoorden, Nico Naus, and Markus Klinik. Tophat next: Even more stylish task-oriented programming. `https://github.com/timjs/tophat-next`, forthcoming.