A symbolic execution semantics for TopHat

Nico Naus
Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
n.naus@uu.nl

Tim Steenvoorden
Software Science
Radboud University
Nijmegen, The Netherlands
tim@cs.ru.nl

Markus Klinik
Software Science
Radboud University
Nijmegen, The Netherlands
m.klinik@cs.ru.nl

ABSTRACT

Task-Oriented Programming (TOP) is a programming paradigm that allows declarative specification of workflows. TOP is typically used in domains where functional correctness is essential, and where failure can have financial or strategical consequences. In this paper we aim to make formal verification of software written in TOP easier. Currently, only testing is used to verify that programs behave as intended. We use symbolic execution to guarantee that no aberrant behaviour can occur. In previous work we presented TopHat, a formal language that implements the core aspects of TOP. In this paper we develop a symbolic execution semantics for TopHat. Symbolic execution allows to prove that a given property holds for all possible execution paths of TopHat programs.

We show that the symbolic execution semantics is consistent with the original TopHat semantics, by proving soundness and completeness. We present an implementation of the symbolic execution semantics in Haskell. By running example programs, we validate our approach. This work represents a step forward in the formal verification of TOP software.

ACM Reference Format:
Nico Naus, Tim Steenvoorden, and Markus Klinik. 2020. A symbolic execution semantics for TopHat. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Task-Oriented Programming paradigm (TOP) is an abstraction over workflow specifications. The idea of TOP is to describe the work that needs to be done, in which order, by which person. From this specification, an application can be generated that helps to coordinate people and machines to execute the work. The iTasks framework [Plasmeijer et al. 2012] is an implementation of the paradigm in the functional programming language Clean. In earlier work [Steenvoorden et al. 2019], we presented the programming language TopHat, written $\widehat{\text{TOP}}$, to distill the core features of TOP into a language suitable for formal treatment. The usefulness of TOP has been demonstrated in several projects that applied it to implement various applications. It has been used by the Netherlands Royal Navy [Jansen et al. 2018], the Dutch Tax Office [Stutterheim et al. 2017] and the Dutch Coast Guard [Lijnse et al. 2012]. Furthermore, it can potentially be applied in domains like healthcare and Internet of Things [Koopman et al. 2018].

Applications in these kinds of domains are often mission critical, where programming mistakes can have severe consequences. To verify that a $\widehat{\text{TOP}}$ program behaves as intended, we would like to show that it satisfies a given property. A common way to do this is to write test cases, or to generate random input, and verify that all outcomes satisfy the property. Writing tests manually is time consuming and cumbersome. Testing interactive applications needs people to operate the application, maybe making use of a way to record and replay interactions. With this kind of testing there is no guarantee that all possible execution paths are covered.

To overcome these issues, we apply symbolic execution. Instead of executing tasks with test input, or letting a user interactively test the application, we run tasks on symbolic input. Symbolic input consists of tokens that represent any value of a certain type. When a program branches, the execution engine records the conditions over the symbolic input that lead to the different branches. These conditions can then be compared to a given predicate to check if the predicate holds under all conditions. We let an SMT solver verify these statements.

In this way we can guarantee that given predicates over the outcome of a TOP program always hold. Since iTasks is not suitable for formal reasoning, we instead apply symbolic execution to $\widehat{\text{TOP}}$ [Steenvoorden et al. 2019], by systematically changing the semantic rules of the original language.

1.1 Contributions

This paper makes the following contributions.

- We present a symbolic execution semantics for $\widehat{\text{TOP}}$, a programming language for workflows embedded in the simply typed λ-calculus.
- We prove soundness and completeness of the symbolic semantics with respect to the original $\widehat{\text{TOP}}$ semantics.
- We present an implementation of the symbolic execution semantics in Haskell.

1.2 Structure

Section 2 gives a brief overview of $\widehat{\text{TOP}}$ and its concepts. Section 3 introduces some examples to demonstrate the goal of our symbolic execution analysis. In Section 4, the $\widehat{\text{TOP}}$ language is defined. Section 5 goes on to define the formal semantics of the symbolic execution. In Section 6, soundness and completeness are shown for the symbolic execution semantics with respect to the original $\widehat{\text{TOP}}$ semantics. In Section 7 related work is discussed, and Section 8 concludes.

2 TOPHAT

This section briefly introduces the task-oriented programming language $\widehat{\text{TOP}}$, and discusses our vision about symbolic evaluation of this language.

IFL'19, September 2019, Singapore
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The $\widehat{\text{TOP}}$ language consists of two parts, the host language and the task language. Programs in $\widehat{\text{TOP}}$ are called *tasks*. The basic elements of tasks are editors. Using combinators, tasks can be combined into larger tasks.

The task language is embedded in a simply typed lambda calculus with references, conditionals, booleans, integers, strings, pairs, lists and unary and binary operations on these types. References allow tasks to communicate with each other, sharing information across task boundaries. The simply typed λ-calculus does not have recursion. By restricting references to only hold basic types, strong normalisation of the calculus is guaranteed. The full syntax of the host language is listed in Section 4. Next, we discuss the main constructs of the task language.

2.1 Editors

Editors are the most basic tasks. They are used to communicate with the outside world. Editors are an abstraction over widgets in a GUI library or on webpage forms. Users can change the value held by an editor, in the same way they can manipulate widgets in a GUI.

When a TOP implementation generates an application from a task specification, it derives user interfaces for the editors. The appearance of an editor is influenced by its type. For example, an editor for a string can be represented by a simple input field, a date by a calendar, and a location by a pin on a map.

There are three different editors in $\widehat{\text{TOP}}$.

$\square\, v$ Valued editor.
: This editor holds a value v of a certain type. The user can replace the value by a new value of the same type.

$\boxtimes\, \tau$ Unvalued editor.
: This editor holds no value, and can receive a value of type τ. When that happens, it turns into a valued editor.

$\blacksquare\, l$ Shared editor.
: This editor refers to a store location l. Its observable value is the value stored at that location. When it receives a new value, this value will be stored at location l.

2.2 Combinators

Editors can be combined into larger tasks using combinators. Combinators describe the way people collaborate. Tasks can be performed in sequence or in parallel, or there is a choice between two tasks.

The following combinators are available in $\widehat{\text{TOP}}$. Here, t stands for tasks and e for arbitrary expressions. The concrete syntax of the language is described in Section 4.1

$t \blacktriangleright e$ Step.
: Users can work on task t. As soon as t has a value, that value is passed on to the right hand side e. The expression e is a function, taking the value as an argument, resulting in a new task.

$t \triangleright e$ User Step.
: Users can work on task t. When t has a value, the step becomes enabled. Users can then send a continue event to the combinator. When that happens, the value of t is passed to the right hand side, with which it continues.

$t_1 \bowtie t_2$ Composition.
: Users can work on tasks t_1 and t_2 in parallel.

$t_1 \blacklozenge t_2$ Choice.
: The system chooses between t_1 or t_2, based on which task first has a value. If both tasks have a value, the system chooses the left one.

$e_1 \lozenge e_2$ User choice.
: A user has to make a choice between either the left or the right hand side. The user continues to work on the chosen task.

In addition to editors and combinators, $\widehat{\text{TOP}}$ also contains the fail task ($\lightning$). Programmers can use this task to indicate that a task is not reachable or viable. When the right hand side of a step combinator evaluates to $\lightning$, the step will not proceed to that task.

2.3 Observations

Several observations can be made on tasks. Using the value function $\mathcal{V}$, the current value of a task can be determined. The value function is a partial function, since not all tasks have a value. For example empty editors and steps do not have a value.

One can also observe whether or not a task is failing, by means of the failing function $\mathcal{F}$. The task $\lightning$ is failing, as is a parallel combination of failing tasks ($\lightning \bowtie \lightning$).

The step combinator makes use of both functions in order to determine if it can step. First, it uses $\mathcal{V}$ to see if the left hand side produces a value. If that is the case, it uses the $\mathcal{F}$ function to see if it is safe to step to the right hand side. The complete definition of the value and failing function are discussed in Section 5.2.

2.4 Input

Input events drive evaluation of tasks. Because tasks are typed, input is typed as well. Editors only accept input of the correct type. Examples are replacing a value in an editor, or sending a continue event to a user step. When the system receives a valid event, it gives this event to the current task, which reduces to a new task. Everything in between interaction steps is evaluated atomically with respect to inputs.

Input events are synchronous, which means the order of execution is completely determined by the order of the events. In particular, the order of input events determine the progression of parallel branches.

3 EXAMPLES

In this section we study three examples to illustrate how the language $\widehat{\text{TOP}}$ works and what kind of properties we would like to prove.

3.1 Positive value

This example demonstrates how we can prove that the first observable value of a program can only be a positive number. Consider the program in Listing 1.

Listing 1: A task that only steps on a positive input value.

$\boxtimes\text{INT} \blacktriangleright \lambda x.$ **if** $x > 0$ **then** $\square x$ **else** $\lightning$

It asks the user to input a value of type INT. This value is then passed on to the right hand side. If the value is greater than zero, an editor containing the entered value is returned. At this point, the task has an observable value, and we consider it done. Otherwise the step does not proceed and the task does not have an observable value. The user can enter a different input value.

Imagine that we want to prove that no matter which value is given as input, the first observable value is a value greater than zero.

Symbolic execution of this program proceeds as follows. The symbolic execution engine generates a fresh symbolic input s for the editor on the left. The engine then arrives at the conditional. To take the then-branch, the condition $s > 0$ needs to hold. This branch will then result in $\Box\, s$, in which case the program has an observable value. The engine records this endpoint together with its path condition $s > 0$. The else-branch applies if the condition does not hold, but this leads to a failing task. Therefore, the step is not taken and the task expression is not altered. No additional program state is generated.

Symbolic execution returns a list of all possible program end states, together with the path conditions that led to them. If all end states satisfy the desired property, it is guaranteed that the property holds for all possible inputs.

In this example, the only end state is the expression $\Box\, s$ with path condition $s > 0$. From that we can conclude that no matter what input is given, the only result value possible is greater than zero.

3.2 Tax subsidy request

Stutterheim et al. [2017] worked with the Dutch tax office to develop a demonstrator for a fictional but realistic law about solar panel subsidies. In this section we study a simplified version of this, translated to $\widehat{\text{TOP}}$, to illustrate how symbolic execution can be used to prove that the program implements the law.

This example proves that a citizen will get subsidy only under the following conditions.

- The roofing company has confirmed that they installed solar panels for the citizen.
- The tax officer has approved the request.
- The tax officer can only approve the request if the roofing company has confirmed, and the request is filed within one year of the invoice date.
- The amount of the granted subsidy is at most 600 EUR.

Listing 2: Subsidy request and approval workflow at the Dutch tax office.

```
let today = 25 Sept 2020 in                                              1
let provideDocuments = ⊠Amount ⋈ ⊠Date in                                2
let companyConfirm = □True ◊ □False in                                   3
let officerApprove = λinvoiceDate. λdate. λconfirmed.                    4
  □False ◊ if (date − invoiceDate < 365 ∧ confirmed)                     5
    then □True                                                           6
    else ↯ in                                                            7
provideDocuments ⋈ companyConfirm ►                                      8
  λ⟨⟨invoiceAmount, invoiceDate⟩ , confirmed⟩ .                          9
officerApprove invoiceDate today confirmed ► λapproved.                  10
let subsidyAmount = if approved                                          11
  then min 600 (invoiceAmount / 10) else 0 in                            12
□⟨subsidyAmount, approved, confirmed, invoiceDate, today⟩                13
```

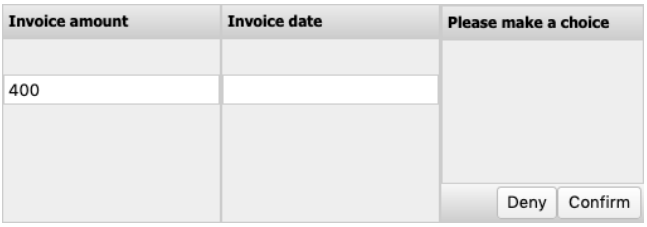

Figure 1: Graphical user interface for the task in Listing 2. In parallel, the citizen is asked to enter the invoice amount and the invoice date of the installed solar panels, and the roofing company is asked to deny or confirm they actually installed the solar panels.

Listing 2 shows the program. To enhance readability of the example, we omit type annotations and make use of pattern matching on tuples. The program works as follows. First, the citizen has to enter their personal information (Line 2). In the original demonstrator this included the citizen service number, name, and home address. Here, we simplified the example so that the citizen only has to enter the invoice date. A date is specified using an integer representing the number of days since 1 January 2000.

In the next step (Line 8), in parallel the citizen has to provide the invoice documents of the installed solar panels, while the roofing company has to confirm that they have actually installed solar panels at the citizen's address. Once the invoice and the confirmation are there, the tax officer has to approve the request (Line 10). The officer can always decline the request, but they can only approve it if the roofing company has confirmed and the application date is within one year of the invoice date (Line 5). The result of the program is the amount of the subsidy, together with all information needed to prove the required properties (Line 13). The graphical user interface belonging to two steps in this process are shown in Fig. 1.

The result of the overall task is a tuple with the subsidy amount, the officer's approval, the roofing company's confirmation, the invoice amount, the invoice date, and today's date. Returning all this information allows the following predicate to be stated, which verifies the correctness of the implementation. The predicate has 5 free variables, which correspond to the returned values.

$$\begin{aligned}
\psi(s,a,c,i,t) &= s \geq 0 \supset c && (1)\\
&\wedge\ s > 0 \supset a && (2)\\
&\wedge\ a \supset (c \wedge t - i < 365) && (3)\\
&\wedge\ s \leq 600 && (4)\\
&\wedge\ \neg a \supset s \equiv 0 && (5)
\end{aligned}$$

The predicate ψ states that (1) if subsidy s has been payed, the roofing company must have confirmed c, (2) if subsidy has been payed, the officer must have approved a, (3) the officer can approve only if the roofing company has confirmed and today's date t is within 356 days of the invoice date i, and (4) the subsidy is maximal 600 EUR. Finally, (5) if the officer has not approved, the subsidy must be 0.

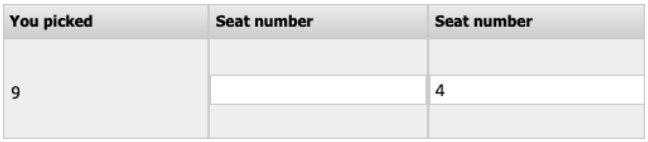

Figure 2: Graphical user interface generated from the specification in Listing 3. Three users are booking seats in parallel. The first user booked seat 9, the second did not enter a seat number yet, and the third is about to book seat 4.

3.3 Flight booking

In this section we develop a small flight booking system. The purpose of this example is to demonstrate how symbolic execution handles references and lists. We prove that when the program terminates, every passenger has exactly one seat, and that no two passengers have the same seat. This program is a simplified version of what we presented in earlier work [Steenvoorden et al. 2019].

Listing 3: Flight booking.

```
let maxSeats = 50 in                                              1
let bookedSeats = ref [] in                                       2
let bookSeat = ⊠Int ▶ λx .                                        3
  if not (x ∈ !bookedSeats) ∧ x ≤ maxSeats                        4
    then bookedSeats := x :: !bookedSeats ▶ λ_ . □x               5
    else ↯ in                                                     6
bookSeat ⋈ bookSeat ⋈ bookSeat ▶ λ_ .                             7
□(!bookedSeats)                                                   8
```

The program, shown in Listing 3, consists of three parallel seat booking tasks (Line 7). There is a shared list that stores all booked seats so far (Line 2). To book a seat, a passenger has to enter a seat number (Line 3). A guard expression makes sure that only free seats can be booked (Line 4). The exclamation mark denotes dereferencing. When the guard is satisfied, the list of booked seats is updated, and the user can see his booked seat (Line 5). The main expression runs the seat booking task three times in parallel (Line 7), simulating three concurrent customers. The program returns the list of booked seats. The graphical user interface, generated from the specification in Listing 3, is shown in Fig. 2.

With the returned list, we can state the predicate to verify the correctness of the booking process.

$$\psi(l) = \mathsf{len}\, l \equiv 3 \tag{1}$$
$$\wedge\ \mathsf{uniq}\, l \tag{2}$$

The predicate specifies that all three passengers booked exactly one seat (1), and that all seats are unique (2), which means that no two passengers booked the same seat. The unary operators for list length (len) and uniqueness (uniq) are available in the predicate language. List length is a capability of SMT-LIB, while uniq is our own addition.

4 LANGUAGE

The language presented in this section is nearly identical to the original $\widehat{\text{TOP}}$ language presented by Steenvoorden et al. [2019]. The main difference with the original grammar is the addition of symbolic values.

Symbolic execution for functional programming languages struggles with higher order features. This topic is under active study, and is not the focus of our work [Hallahan et al. 2019, 2017]. Therefore, we restrict symbols to only represent values of basic types. This restriction is of little importance in the domains we are interested in. Allowing users to enter higher order values is not useful in most workflow applications. By restricting the input grammar to first-order values only, we ensure that no higher-order user input can be entered. Apart from input, all other higher order features are unrestricted.

The following subsections describe in detail how all elements of the $\widehat{\text{TOP}}$ language deal with the addition of symbols.

4.1 Expressions, values, and types

The syntax of Symbolic $\widehat{\text{TOP}}$ is listed in Fig. 3. Two main changes have been made with regards to the original $\widehat{\text{TOP}}$ grammar. The differences with the original syntax are highlighted in grey boxes. First, symbols s have been added to the syntax of expressions. However, they are not intended to be used by programmers, similar to locations l. Instead, they are generated by the semantics as placeholders for symbolic inputs. Second, unary and binary operations have been made explicit.

Expressions

$\tilde{e}$	::=	$\lambda x : \tau\,.\ \tilde{e} \mid \tilde{e}_1\ \tilde{e}_2$	– abstraction, application
	|	$x \mid c \mid \langle\rangle$	– variable, constant, unit
	|	$u\ \tilde{e}_1 \mid \tilde{e}_1\ o\ \tilde{e}_2$	– unary, binary operation
	|	**if** $\tilde{e}_1$ **then** $\tilde{e}_2$ **else** $\tilde{e}_3$	– conditional
	|	$\langle \tilde{e}_1, \tilde{e}_2 \rangle \mid \mathsf{fst}\ \tilde{e} \mid \mathsf{snd}\ \tilde{e}$	– pair, projections
	|	$[\,]_\beta \mid \tilde{e}_1 :: \tilde{e}_2$	– nil, cons
	|	$\mathsf{head}\ \tilde{e} \mid \mathsf{tail}\ \tilde{e}$	– first element, list tail
	|	**ref** $\tilde{e} \mid\, !\tilde{e} \mid \tilde{e}_1 := \tilde{e}_2 \mid l$	– references, location
	|	$\tilde{p} \mid s$	– pretask, symbol

Constants

c	::=	$B \mid I \mid S$	– boolean, integer, string

Unary Operations

u	::=	$\neg \mid - \mid \mathsf{len} \mid \mathsf{uniq}$	– not, negate, length, unique

Binary Operations

o	::=	$< \mid \leq \mid \equiv \mid \not\equiv \mid \geq \mid >$	– equational
	|	$+ \mid - \mid \times \mid /$	– numerical
	|	$\wedge \mid \vee$	– conjunction, disjunction
	|	$+\!\!+ \mid \in$	– append, elementhood

Pretasks

$\tilde{p}$	::=	$\square\,\tilde{e} \mid \boxtimes\beta \mid \blacksquare\,\tilde{e}$	– editors: valued, unvalued, shared
	|	$\tilde{e}_1 \blacktriangleright \tilde{e}_2 \mid \tilde{e}_1 \triangleright \tilde{e}_2$	– steps: internal, external
	|	$\lightning \mid \tilde{e}_1 \bowtie \tilde{e}_2$	– fail, composition
	|	$\tilde{e}_1 \blacklozenge \tilde{e}_2 \mid \tilde{e}_1 \lozenge \tilde{e}_2$	– choice: internal, external

Figure 3: Syntax of Symbolic $\widehat{\text{TOP}}$ expressions.

Symbols are treated as values (Fig. 4). They have therefore been added to the grammar of values. Also, every symbol has a type, and basic operations can take symbols as arguments. As a result, we must now also regard unary and binary operations as values. Therefore we make these operations explicit in this language description, where in the original they were left implicit.

Values

$$\begin{array}{lcll} \tilde{v} & ::= & \lambda x : \tau \,.\, \tilde{e} \mid \langle \tilde{v}_1, \tilde{v}_2 \rangle \mid \langle\rangle & \text{– abstraction, pair, unit} \\ & \mid & [\,]_\beta \mid \tilde{v}_1 :: \tilde{v}_2 \mid c & \text{- nil, cons, constant} \\ & \mid & l \mid \tilde{t} \mid s & \text{– location, task, symbol} \\ & \mid & u\ \tilde{v} \mid \tilde{v}_1\ o\ \tilde{v}_2 & \text{– unary/binary operation} \end{array}$$

Tasks

$$\begin{array}{lcll} \tilde{t} & ::= & \square\, \tilde{v} \mid \boxtimes \beta \mid \blacksquare\, l & \text{– editors} \\ & \mid & \tilde{t}_1 \blacktriangleright \tilde{e}_2 \mid \tilde{t}_1 \triangleright \tilde{e}_2 & \text{– steps} \\ & \mid & \lightning \mid \tilde{t}_1 \bowtie \tilde{t}_2 & \text{– fail, combination} \\ & \mid & \tilde{t}_1 \blacklozenge \tilde{t}_2 \mid \tilde{e}_1 \lozenge \tilde{e}_2 & \text{– choices} \end{array}$$

Figure 4: Syntax of values in Symbolic $\widehat{\text{TOP}}$.

Types

$$\begin{array}{lcll} \tau & ::= & \tau_1 \to \tau_2 \mid \beta & \text{– function, basic} \\ & \mid & \textsc{Ref}\ \tau \mid \textsc{Task}\ \tau & \text{– reference, task} \end{array}$$

Basic types

$$\begin{array}{lcll} \beta & ::= & \beta_1 \times \beta_2 \mid \textsc{List}\ \beta \mid \textsc{Unit} & \text{– product, list, unit} \\ & \mid & \textsc{Bool} \mid \textsc{Int} \mid \textsc{String} & \text{– boolean, integer, string} \end{array}$$

Figure 5: Syntax of Symbolic $\widehat{\text{TOP}}$ types.

$$\textsc{T-Sym}\ \frac{s : \beta \in \Gamma}{\Gamma, \Sigma \vdash s : \tau}$$

Figure 6: Additional typing rule for Symbolic $\widehat{\text{TOP}}$.

The types of Symbolic $\widehat{\text{TOP}}$ remain the same (Fig. 5). However, we do need an additional typing rule, T-Sym in Fig. 6, to type symbols, since they are now part of our expression syntax. The type of symbols is kept track of in the environment Γ.

4.2 Inputs

In symbolic execution, we do not know what the input of a program will be. In our case this means that we do not know which events will be sent to editors. This is reflected in the definition of symbolic inputs and actions in Fig. 7

Symbolic inputs

$$\begin{array}{lcll} \tilde{\imath} & ::= & \tilde{a} \mid \mathsf{F}\,\tilde{\imath} \mid \mathsf{S}\,\tilde{\imath} & \text{– symbolic action, to first, to second} \end{array}$$

Symbolic actions

$$\begin{array}{lcll} \tilde{a} & ::= & s & \text{– symbol} \\ & \mid & \mathsf{C} \mid \mathsf{L} \mid \mathsf{R} & \text{– continue, go left, go right} \end{array}$$

Figure 7: Syntax of inputs and actions in Symbolic $\widehat{\text{TOP}}$.

Inputs are still the same and consist of paths and actions. Paths are tagged with one or more F (first) and S (second) tags. Actions no longer contain concrete values, but only symbols. This means that instead of concrete values, editors can only hold symbols.

4.3 Path conditions

Concrete execution of $\widehat{\text{TOP}}$ programs is driven by concrete inputs, which select one branch of conditionals, or make a choice. Since no concrete information is available during symbolic execution, the symbolic execution semantics records how each execution path depends on the symbolic input. This is done by means of path conditions. Figure 8 lists the syntax of path conditions.

Path conditions

$$\begin{array}{lcll} \varphi & ::= & c \mid s & \text{– constant, symbol} \\ & \mid & \langle\rangle \mid \langle \varphi_1, \varphi_2 \rangle & \text{– unit, pairs} \\ & \mid & [\,]_\beta \mid \varphi_1 :: \varphi_2 & \text{– nil, cons} \\ & \mid & u\ \varphi \mid \varphi_1\ o\ \varphi_2 & \text{– symbolic unary/binary operation} \end{array}$$

Figure 8: Syntax of path conditions.

Path conditions are a subset of the values of basic type β. They can contain symbols, constants, pairs, lists, and operations on them.

5 SEMANTICS

In this section we discuss the symbolic execution semantics for $\widehat{\text{TOP}}$. The structure of the symbolic semantics closely resembles that of the concrete semantics. It consists of three layers, a big step symbolic evaluation semantics for the host language, a big step symbolic normalisation semantics for the task language, and a small step driving semantics that processes user inputs. Figure 9 gives an overview of the relations between the different semantics.

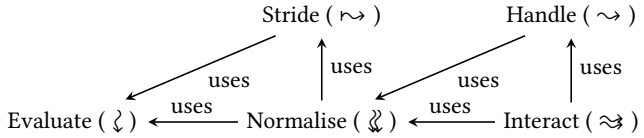

Figure 9: Semantic functions defined in this report and their relation.

They are described in the following sections. We will study their interesting aspects, and the changes made with respect to the concrete semantics.

5.1 Symbolic evaluation

The host language is a simply typed lambda calculus with references and basic operations. Most of the symbolic evaluation rules closely resemble the concrete semantics. The original evaluation relation ($\hat{\downarrow}$) had the form $e, \sigma \downarrow v, \sigma'$, where an expression e in a state $\hat{\sigma}$ evaluates to a value $\hat{v}$ in state $\hat{\sigma}'$. The new relation ($\downarrow$) adds path conditions φ to the output and has the form $\tilde{e}, \tilde{\sigma} \downarrow \overline{\tilde{v}, \tilde{\sigma}', \varphi}$. The hat distinguishes the old concrete and the new symbolic variants."

The symbolic semantics can generate multiple outcomes. This is denoted in the evaluation with a line over the result, which can be read as $\overline{v, \sigma', \varphi} = \{(v_1, \sigma_1', \varphi_1), \cdots, (v_n, \sigma_n', \varphi_n)\}$. The set that results from symbolic execution can be interpreted as follows. Each element is a possible endpoint in the execution of a task. It is guarded by a condition φ over the symbolic input. Execution only arrives at the symbolic value v and symbolic state σ' when the path condition φ is satisfied.

To illustrate the difference between concrete and symbolic evaluation, Fig. 10 lists one rule from the concrete semantics and its corresponding symbolic counterpart.

The E-Edit rule evaluates the expression held in an editor to a value. The SE-Edit does the same, but since it is concerned with

$$\textsc{E-Edit}\ \frac{e, \sigma \downarrow v, \sigma'}{\square\, e, \sigma \downarrow \square\, v, \sigma'} \qquad \textsc{SE-Edit}\ \frac{\tilde{e}, \tilde{\sigma} \swarrow \overline{\tilde{v}, \tilde{\sigma}', \varphi}}{\square\, \tilde{e}, \tilde{\sigma} \swarrow \overline{\square\, \tilde{v}, \tilde{\sigma}', \varphi}}$$

Figure 10: The evaluation rule from the concrete and the symbolic semantics for the editor expression.

symbolic execution, the expression can contain symbols. We therefore do not know beforehand which concrete value will be produced, or even which path the execution will take. If the expression contains a conditional that depends on a symbol, there can be multiple possible result values.

Most symbolic rules closely resemble their concrete counterparts, and follow directly from them. The rules are not listed here, a full overview can be found in **??**.

The only interesting rule is the one for conditionals, listed in Fig. 11. The concrete semantics has two separate rules for the **then**

$$\boxed{\tilde{e}, \tilde{\sigma} \swarrow \overline{\tilde{v}, \tilde{\sigma}', \varphi}}$$

$$\textsc{SE-If}\ \frac{\tilde{e}_1, \tilde{\sigma} \swarrow \overline{\tilde{v}_1, \tilde{\sigma}', \varphi_1} \qquad \tilde{e}_2, \tilde{\sigma}' \swarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_2} \qquad \tilde{e}_3, \tilde{\sigma}' \swarrow \overline{\tilde{v}_3, \tilde{\sigma}''', \varphi_3}}{\textbf{if}\ \tilde{e}_1\ \textbf{then}\ \tilde{e}_2\ \textbf{else}\ \tilde{e}_3, \tilde{\sigma} \swarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_1 \wedge \varphi_2 \wedge \tilde{v}_1} \cup \overline{\tilde{v}_3, \tilde{\sigma}''', \varphi_1 \wedge \varphi_3 \wedge \neg \tilde{v}_1}}$$

Figure 11: Part of the symbolic evaluation semantics.

and the **else** branch. The symbolic semantics has one combined rule SE-If. Since e_1 can contain symbols, it can evaluate to multiple values. The rule keeps track of all options. It calculates the **then**-branch, and records in the path condition that execution can only reach this branch if v_1 becomes True. The rule does the same for the **else**-branch, except it requires that v_1 becomes False. Note that both e_2 and e_3 are evaluated using the same state σ', which is the resulting state after evaluating e_1.

5.2 Observations

The symbolic normalisation and driving semantics make use of observations on tasks, just like the concrete semantics.

The partial function $\mathcal{V}$ can be used to observe the value of a task. Its definition is given in Fig. 12. It is unchanged with respect to the original.

$$\begin{aligned}
&\mathcal{V} : \text{Tasks} \times \text{States} \rightharpoonup \text{Values} \\
&\mathcal{V}(\square\, \tilde{v}, \tilde{\sigma}) = \tilde{v} \\
&\mathcal{V}(\boxtimes\, \beta, \tilde{\sigma}) = \bot \\
&\mathcal{V}(\blacksquare\, l, \tilde{\sigma}) = \tilde{\sigma}(l) \\
&\mathcal{V}(\lightning, \tilde{\sigma}) = \bot \\
&\mathcal{V}(\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}) = \bot \\
&\mathcal{V}(\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}) = \bot \\
&\mathcal{V}(\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}) = \begin{cases} \langle \tilde{v}_1, \tilde{v}_2 \rangle & \textbf{when } \mathcal{V}(\tilde{t}_1, \tilde{\sigma}) = \tilde{v}_1 \wedge \mathcal{V}(\tilde{t}_2, \tilde{\sigma}) = \tilde{v}_2 \\ \bot & \textbf{otherwise} \end{cases} \\
&\mathcal{V}(\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}) = \begin{cases} \tilde{v}_1 & \textbf{when } \mathcal{V}(\tilde{t}_1, \tilde{\sigma}) = \tilde{v}_1 \\ \tilde{v}_2 & \textbf{when } \mathcal{V}(\tilde{t}_1, \tilde{\sigma}) = \bot \wedge \mathcal{V}(\tilde{t}_2, \tilde{\sigma}) = \tilde{v}_2 \\ \bot & \textbf{otherwise} \end{cases} \\
&\mathcal{V}(\tilde{t}_1 \lozenge \tilde{t}_2, \tilde{\sigma}) = \bot
\end{aligned}$$

Figure 12: Task value observation function $\mathcal{V}$.

$$\begin{aligned}
&\mathcal{F} : \text{Tasks} \times \text{States} \rightarrow \text{Booleans} \\
&\mathcal{F}(\square\, \tilde{v}, \tilde{\sigma}) = \text{False} \\
&\mathcal{F}(\boxtimes\, \beta, \tilde{\sigma}) = \text{False} \\
&\mathcal{F}(\blacksquare\, l, \tilde{\sigma}) = \text{False} \\
&\mathcal{F}(\lightning, \tilde{\sigma}) = \text{True} \\
&\mathcal{F}(\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}) = \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \\
&\mathcal{F}(\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}) = \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \\
&\mathcal{F}(\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}) = \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \wedge \mathcal{F}(\tilde{t}_2, \tilde{\sigma}) \\
&\mathcal{F}(\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}) = \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \wedge \mathcal{F}(\tilde{t}_2, \tilde{\sigma}) \\
&\mathcal{F}(\tilde{e}_1 \lozenge \tilde{e}_2, \tilde{\sigma}) = \bigwedge \Big(\{ \mathcal{F}(\tilde{t}_1, \tilde{\sigma}_1') \mid \tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\tilde{t}_1, \tilde{\sigma}_1'} \} \cup \\
&\qquad \{ \mathcal{F}(\tilde{t}_2, \tilde{\sigma}_2') \mid \tilde{e}_2, \tilde{\sigma} \Downarrow \overline{\tilde{t}_2, \tilde{\sigma}_2'} \} \Big)
\end{aligned}$$

Figure 13: Task failing observation function $\mathcal{F}$.

The function $\mathcal{F}$ observes if a task is failing. Its definition is given in Fig. 13. A task is failing if it is the fail task ($\lightning$), or if it consists of only failing tasks. This function differs from its concrete counterpart in the clause for user choice. As symbolic normalisation can yield multiple results, all of the results must be failing to make a user choice failing.

5.3 Normalisation

Normalization ($\Downarrow$) reduces tasks until they are ready to receive input. Very little has to be changed to accommodate symbolic execution. Just like the evaluation semantics it now gathers sets of results, each element guarded by a path condition. Figure 14 lists the normalisation semantics.

$$\boxed{\tilde{e}, \tilde{\sigma} \Lsh \overline{\tilde{t}, \tilde{\sigma}', \varphi}}$$

$$\textsc{SN-Done}\ \frac{\tilde{e}, \tilde{\sigma} \swarrow \overline{\tilde{t}, \tilde{\sigma}', \varphi_1} \qquad \tilde{t}, \tilde{\sigma}' \rightsquigarrow \overline{\tilde{t}', \tilde{\sigma}'', \varphi_2}}{\tilde{e}, \tilde{\sigma} \Lsh \overline{\tilde{t}, \tilde{\sigma}', \varphi_1 \wedge \varphi_2}}\ \tilde{\sigma}' = \tilde{\sigma}'' \wedge \tilde{t} = \tilde{t}'$$

$$\textsc{SN-Repeat}\ \frac{\tilde{e}, \tilde{\sigma} \swarrow \overline{\tilde{t}, \tilde{\sigma}', \varphi_1} \qquad \tilde{t}, \tilde{\sigma}' \rightsquigarrow \overline{\tilde{t}', \tilde{\sigma}'', \varphi_2}\ \tilde{t}', \tilde{\sigma}'' \Lsh \overline{\tilde{t}'', \tilde{\sigma}''', \varphi_3}}{\tilde{e}, \tilde{\sigma} \Lsh \overline{\tilde{t}'', \tilde{\sigma}''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}}\ \tilde{\sigma}' \neq \tilde{\sigma}'' \vee \tilde{t} \neq \tilde{t}'$$

Figure 14: Symbolic normalisation semantics.

Normalisation makes use of the small step striding semantics ($\mapsto$). Its details are not important here. For more background, we refer to the appendix.

5.4 Handling

The handling semantics ($\rightarrow$) deals with user input. In the symbolic case there are symbols instead of concrete inputs. A complete overview of the rules can be found in **??**. Figure 15 lists the interesting rules of the symbolic handling semantics.

The three rules for the editors (SH-Change, SH-Fill, SH-Update) clearly show how symbols enter the symbolic execution. The first one for example generates a fresh symbol s and returns an editor containing it.

$$\boxed{\tilde{t},\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}',\ \tilde{\sigma}',\ \tilde{\imath},\ \varphi}}$$

SH-Change
$$\frac{\text{fresh } s}{\square\,\tilde{v},\ \tilde{\sigma}\ \rightsquigarrow\ \square\,s,\ \tilde{\sigma},\ s,\ \text{True}}\ \tilde{v},\ s:\beta$$

SH-Fill
$$\frac{\text{fresh } s \qquad s:\beta}{\boxtimes\,\beta,\ \tilde{\sigma}\ \rightsquigarrow\ \square\,s,\ \tilde{\sigma},\ s,\ \text{True}}$$

SH-Update
$$\frac{\text{fresh } s \qquad \tilde{\sigma}(l),\ s:\beta}{\blacksquare\,l,\ \tilde{\sigma}\ \rightsquigarrow\ \blacksquare\,l,\ \tilde{\sigma}[l\mapsto s],\ s,\ \text{True}}$$

SH-PickLeft
$$\frac{\tilde{e}_1,\ \tilde{\sigma}\ \not\Downarrow\ \overline{\tilde{t}_1,\ \tilde{\sigma}_1,\ \varphi_1} \qquad \tilde{e}_2,\ \tilde{\sigma}\ \not\Downarrow\ \overline{\tilde{t}_2,\ \tilde{\sigma}_2,\ \varphi_2}}{\tilde{e}_1 \lozenge \tilde{e}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \tilde{t}_1,\ \tilde{\sigma}_1,\ \mathsf{L},\ \varphi_1}\ \neg\mathcal{F}(\tilde{t}_1,\tilde{\sigma}_1)\wedge\mathcal{F}(\tilde{t}_2,\tilde{\sigma}_2)$$

SH-PickRight
$$\frac{\tilde{e}_1,\ \tilde{\sigma}\ \not\Downarrow\ \overline{\tilde{t}_1,\ \tilde{\sigma}_1,\ \varphi_1} \qquad \tilde{e}_2,\ \tilde{\sigma}\ \not\Downarrow\ \overline{\tilde{t}_2,\ \tilde{\sigma}_2,\ \varphi_2}}{\tilde{e}_1 \lozenge \tilde{e}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \tilde{t}_2,\ \tilde{\sigma}_2,\ \mathsf{R},\ \varphi_2}\ \mathcal{F}(\tilde{t}_1,\tilde{\sigma}_1)\wedge\neg\mathcal{F}(\tilde{t}_2,\tilde{\sigma}_2)$$

SH-Pick
$$\frac{\tilde{e}_1,\ \tilde{\sigma}\ \not\Downarrow\ \overline{\tilde{t}_1,\ \tilde{\sigma}_1,\ \varphi_1} \qquad \tilde{e}_2,\ \tilde{\sigma}\ \not\Downarrow\ \overline{\tilde{t}_2,\ \tilde{\sigma}_2,\ \varphi_2}}{\tilde{e}_1 \lozenge \tilde{e}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}_1,\ \tilde{\sigma}_1,\ \mathsf{L},\ \varphi_1}\cup\overline{\tilde{t}_2,\ \tilde{\sigma}_2,\ \mathsf{R},\ \varphi_2}}\ \neg\mathcal{F}(\tilde{t}_1,\tilde{\sigma}_1)\wedge\neg\mathcal{F}(\tilde{t}_2,\tilde{\sigma}_2)$$

SH-Next
$$\frac{\tilde{t}_1,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_1,\ \tilde{\sigma}_1,\ \tilde{\imath},\ \varphi_1} \qquad \tilde{e}_2\ \tilde{v}_1,\ \tilde{\sigma}_1\ \not\Downarrow\ \overline{\tilde{t}_2,\ \tilde{\sigma}_2,\ \varphi_2}}{\tilde{t}_1 \rhd \tilde{e}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_1 \rhd \tilde{e}_2,\ \tilde{\sigma}_1,\ \tilde{\imath},\ \varphi_1}\cup\overline{\tilde{t}_2,\ \tilde{\sigma}_2,\ \mathsf{C},\ \varphi_2}}\ \mathcal{V}(\tilde{t}'_1,\tilde{\sigma}_1)=\tilde{v}_1\wedge\neg\mathcal{F}(\tilde{t}_2,\tilde{\sigma}_2)$$

SH-And
$$\frac{\tilde{t}_1,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_1,\ \tilde{\sigma}_1,\ \tilde{\imath}_1,\ \varphi_1} \qquad \tilde{t}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_2,\ \tilde{\sigma}_2,\ \tilde{\imath}_2,\ \varphi_2}}{\tilde{t}_1 \bowtie \tilde{t}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_1 \bowtie \tilde{t}_2,\ \tilde{\sigma}_1,\ \mathsf{F}\,\tilde{\imath}_1,\ \varphi_1}\cup\overline{\tilde{t}_1 \bowtie \tilde{t}'_2,\ \tilde{\sigma}_2,\ \mathsf{S}\,\tilde{\imath}_2,\ \varphi_2}}$$

SH-Or
$$\frac{\tilde{t}_1,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_1,\ \tilde{\sigma}_1,\ \tilde{\imath}_1,\ \varphi_1} \qquad \tilde{t}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_2,\ \tilde{\sigma}_2,\ \tilde{\imath}_2,\ \varphi_2}}{\tilde{t}_1 \blacklozenge \tilde{t}_2,\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}'_1 \blacklozenge \tilde{t}_2,\ \tilde{\sigma}_1,\ \mathsf{F}\,\tilde{\imath}_1,\ \varphi_1}\cup\overline{\tilde{t}_1 \blacklozenge \tilde{t}'_2,\ \tilde{\sigma}_2,\ \mathsf{S}\,\tilde{\imath}_2,\ \varphi_2}}$$

Figure 15: Symbolic handling semantics.

$$\begin{aligned}
&\mathit{simulate} : \text{Tasks} \times \text{States} \times [\text{Inputs}] \times \text{Predicates} \to \mathcal{P}(\text{Values} \times [\text{Inputs}] \times \text{Predicates})\\
&\mathit{simulate}\ (t,\sigma,I,\varphi) = \bigcup\{\mathit{simulate}'\ (\text{True},t,t',\sigma',I\oplus[i'],\varphi\wedge\varphi') \mid t,\sigma \Rightarrow t',\sigma',i',\varphi'\}\\
&\mathit{simulate}' : \text{Booleans} \times \text{Tasks} \times \text{Tasks} \times \text{States} \times [\text{Inputs}] \times \text{Predicates} \to \mathcal{P}(\text{Values} \times [\text{Inputs}] \times \text{Predicates})\\
&\mathit{simulate}'\ (\mathit{again},t,t',\sigma',I,\varphi) =\\
&\begin{cases}
\varnothing & \neg\mathcal{S}(\varphi)\\
\{(v,I,\varphi)\} & \mathcal{S}(\varphi)\wedge\mathcal{V}(t',\sigma')=v\\
\mathit{simulate}\ (t',\sigma',I,\varphi) & \mathcal{S}(\varphi)\wedge\mathcal{V}(t',\sigma')=\bot\wedge t'\neq t\\
\bigcup\{\mathit{simulate}'\ (\text{False},t',t'',\sigma'',I\oplus[i'],\varphi\wedge\varphi') \mid t',\sigma' \twoheadrightarrow t'',\sigma'',i',\varphi'\} & \mathcal{S}(\varphi)\wedge\mathcal{V}(t',\sigma')=\bot\wedge t'=t\wedge\ \mathit{again}\\
\varnothing & \mathcal{S}(\varphi)\wedge\mathcal{V}(t',\sigma')=\bot\wedge t'=t\wedge\neg\mathit{again}
\end{cases}
\end{aligned}$$

Figure 16: Simulation function definition.

There are several task combinators where the result depends on user input. For example, the parallel combinator ($\bowtie$) receives an input for either the left or the right branch. To accommodate for all possibilities, the SH-And rule generates both cases. It tags the inputs for the first branch with F and inputs for the second branch with S.

The same principle applies to the external choice combinator ($\lozenge$). The three rules SH-PickLeft, SH-PickRight, and SH-Pick are needed to disallow choosing failing tasks. There is one rule for the case where only the right is failing, one rule when the left is failing, and one for when none of the options are failing.

After input has been handled, tasks are normalised. The combination of those two steps is taken care of by the driving ($\Rightarrow$) semantics, listed in Fig. 17.

$$\boxed{\tilde{t},\ \tilde{\sigma}\ \twoheadrightarrow\ \overline{\tilde{t}',\ \tilde{\sigma}',\ \tilde{\imath},\ \varphi}}$$

SI-Handle
$$\frac{\tilde{t},\ \tilde{\sigma}\ \rightsquigarrow\ \overline{\tilde{t}',\ \tilde{\sigma}',\ \tilde{\imath},\ \varphi_1} \qquad \tilde{t}',\ \tilde{\sigma}'\ \not\Downarrow\ \overline{\tilde{t}'',\ \tilde{\sigma}'',\ \varphi_2}}{\tilde{t},\ \tilde{\sigma}\ \twoheadrightarrow\ \overline{\tilde{t}'',\ \tilde{\sigma}'',\ \tilde{\imath},\ \varphi_1\wedge\varphi_2}}$$

Figure 17: Symbolic driving semantics.

5.5 Simulating

The symbolic driving semantics is a small step semantics. Every step simulates one symbolic input. To compute every possible execution, the driving semantics needs to be applied repeatedly, until the task is done. We define a task to be done when it has an observable value: $\mathcal{V}(t',\sigma')\neq\bot$. The simulation function listed in Fig. 16 is recursively called to produce a list of end states and path conditions. It accumulates all symbolic inputs and returns for each possible execution the observable task value v, the path condition φ, and the state σ. We consider a task, state and path condition to be an end state if the task value can be observed, and the path condition is satisfiable, represented by the function $\mathcal{S}$.

The recursion terminates when one of the following conditions is met.

- $\neg\mathcal{S}(\varphi)$ When the path condition cannot be satisfied, we know that all future steps will not be satisfiable either. All future steps will only add more restrictions to the path condition. No future path condition will be satisfiable, and we can therefore safely remove it.
- $\mathcal{V}(t,\sigma)$ When the current task has a value it is an end state, which we can return.
- $\mathcal{V}(t',\sigma')=\bot\wedge t=t'\wedge\neg\mathit{again}$ When the current task does not produce a value, and it is equal to the previous task except from symbol names in editors, the *simulate* function performs one

look-ahead step in case the task does proceed when a fresh symbol is entered. This one step look-ahead is encoded by the parameter *again*. When this parameter is set to False, one step look-ahead has been performed and *simulate* does not continue further. If the task has a value it is returned, otherwise the branch is pruned.

To better illustrate how the *simulate* function works, we study how it simulates Listing 1. Figure 18 gives a schematic overview of the application of *simulate*. First, it calls the drive semantics to calculate what input the task takes. Users can enter a fresh symbol s_0, as listed on the left. The symbolic execution then branches, since it reaches a conditional. Two cases are generated. Either $s_0 > 0$, the upper branch, or $s_0 \leq 0$, the branch to the right. In the first case, the resulting task has a value, and the symbolic execution ends returning that value and the input. In the second case, the resulting task does not have a value, and the new task is different from the previous task. Therefore, it recurses, and *simulate* is called again.

A fresh symbol s_1 is generated. Again, s_1 can either be greater than zero, or less or equal. In the first case, the resulting task has a value, and the execution ends. In the second case however, the task does not have a value, and we find that the task has not been altered (apart from the new symbol). This results in a recursive call to *simulate*′ with *again* set to False.

Once more a fresh symbol s_2 is generated, and s_2 can be greater than zero, or less or equal. In the first case, the task has a value and we are done. In the second case, it does not have a value, the task again has not changed, but *again* is False and therefore symbolic execution prunes this branch.

This example demonstrates a couple of things. From manual inspection, it is clear that only the first iteration returns an interesting result. When s_0 is greater than zero, the task results in a value that is greater than zero. When the input is less than or equal to zero, simulation continues with the task unchanged.

Why does the simulation still proceed then? Since the editor ⊠ changes to □, the tasks are not the same after the first step. This causes *simulate* to run an extra iteration. It finds that the task still does not have a value, but now the task has changed. Then *simulate* performs one look-ahead step, by setting the *again*-parameter to False. When this look-ahead does not return a value, the branch is pruned.

5.6 Solving

To check the satisfiability of path conditions $\mathcal{S}(\varphi)$, as well as the properties stated about a program, we make use of an external SMT solver. In the implementation we use z3, although any other SMT solver supporting SMT-LIB could be used.

For Listing 1, we would like to prove that after any input sequence I, the path conditions φ imply that the value v of the resulting task t' is greater than 0.

$$\varphi \supset v > 0 \qquad \textbf{where } v = \mathcal{V}(t', \sigma')$$

As shown in Fig. 18, there are three paths we need to verify. Therefore, we send the following three statements to the SMT solver for verification:

(1) $s_0 > 0 \supset s_0 > 0$
(2) $s_0 \leq 0 \wedge s_1 > 0 \supset s_1 > 0$
(3) $s_0 \leq 0 \wedge s_1 \leq 0 \wedge s_2 > 0 \supset s_2 > 0$

In this example all are trivially solvable.

5.7 Implementation

We implemented our language and its symbolic execution semantics in Haskell.[1] With the help of a couple of GHC extensions, the grammar, typing rules and semantics are almost one-to-one translatable into code. Our tool generates execution trees like the one shown in Fig. 18, which keep track of intermediate normalisations, symbolic inputs, and path conditions. All path conditions are converted to SMT-LIB compatible statements and are verified using the z3 SMT solver. As of now we do not have a parser, programs must be specified directly as abstract syntax trees.

As is usually the case with symbolic execution, the number of paths grows quickly. The examples in Listings 2 and 3 generate respectively 2112 and 1166 paths, which takes about a minute to calculate. Solving them, however, is almost instantaneous.

5.8 Outlook

Assertions. Other work on symbolic execution often uses assertions, which are included in the program itself. One could imagine an assertion statement **assert** ψ t in $\widehat{\text{TOP}}$ that roughly works as follows. First the SMT solver verifies the property ψ against the current path condition. If the assertion fails, an error message is generated. Then the program continues with task t.

Example 5.1. Consider the following small example program.

⊠INT ▶ λx . □(**ref** x) ▶ λl. **assert** (!$l \equiv x$) (□ "Done")

This program asks the user to enter an integer. The entered value is then stored in a reference. The assertion that follows ensures that the store has been updated correctly. Finally the string "Done" is returned.

Assertions have access to all variables in scope, unlike properties as we have currently implemented them. We can overcome this by returning all values of interest at the end of the program.

⊠INT ▶ λx . □(**ref** x) ▶ λstore . □"Done" ▶ $\lambda\_$. □(x,!store)

It is now possible to verify that the property $\psi(x, s) = x \equiv s$ holds. This demonstrates that our approach has expressive power similar to assertions. Having assertions in our language would be more convenient for programmers however, and we would like add them in the future.

Input-dependent predicates. Another feature we would like to support in the future are input-dependent predicates.

Example 5.2. Consider the following small program.

⊠INT ▶ λx . **if** $x > 0$ **then** □"Thank you" **else** □"Error"

The user inputs an integer. If the integer is larger than zero, the program prints a thank you message. If the integer is smaller than zero, an error is returned.

If we want to prove that given a positive input, the program never returns "Error", we need to be able to talk about inputs directly in predicates. Currently our symbolic execution does not support this.

[1] https://github.com/timjs/symbolic-tophat-haskell

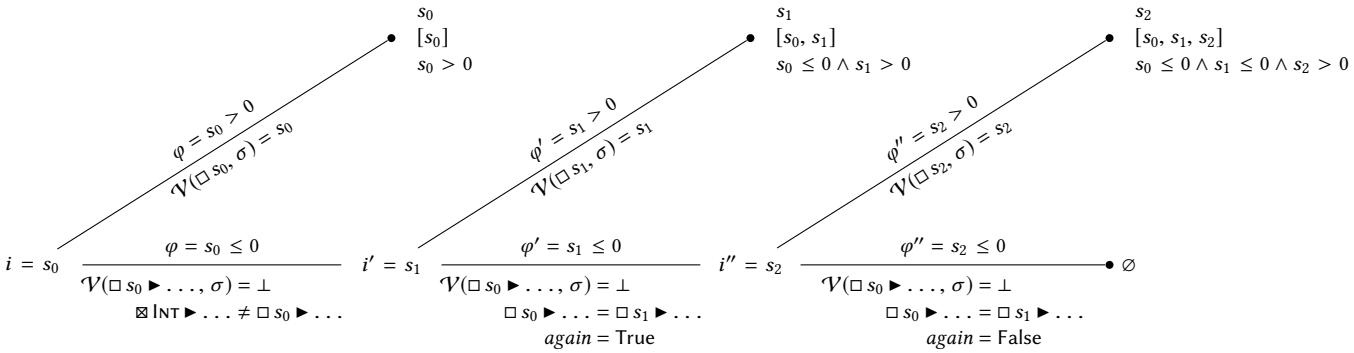

Figure 18: Application of the simulation function to Listing 1.

6 PROPERTIES

In this section we describe what it means for the symbolic execution semantics to be correct. We prove it sound and complete with respect to the concrete semantics of $\widehat{\text{TOP}}$.

To relate the two semantics, we use the concrete inputs listed in Fig. 19.

Concrete inputs

$$j ::= a \mid \mathsf{F}\, j \mid \mathsf{S}\, j \quad \text{– action, to first, to second}$$

Concrete actions

$$a ::= c \quad \text{– constant}$$
$$\mid \mathsf{C} \mid \mathsf{L} \mid \mathsf{R} \quad \text{– continue, go left, go right}$$

Figure 19: Syntax of concrete inputs.

6.1 Soundness

To validate the symbolic execution semantics, we want to show that for every individual symbolic execution step there exists a corresponding concrete one. This soundness property is expressed by Theorem 6.1.

Theorem 6.1 (Soundness of interact). *For all concrete tasks t, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}', \tilde{\sigma}', \tilde{\imath}, \varphi)$ in $t, \sigma \rightsquigarrow \overline{\tilde{t}', \tilde{\sigma}', \tilde{\imath}, \varphi}$ that $M\varphi$ implies $t, \sigma \overset{M\tilde{\imath}}{\Longrightarrow} t', \sigma''$ and $M\tilde{t}' \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma''$.*

The proof for this theorem is rather straightforward. Since the driving semantics makes use of the handling and the normalisation semantics, we require two lemmas. One showing that the handling semantics is sound, Lemma 6.2, and one showing that the normalisation semantics is sound, Lemma 6.3.

Lemma 6.2 (Soundness of handling). *For all concrete tasks t, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}', \tilde{\sigma}', \tilde{\imath}, \varphi)$ in $t, \sigma \rightsquigarrow \overline{\tilde{t}', \tilde{\sigma}', \tilde{\imath}, \varphi}$, that $M\varphi$ implies $t, \sigma \xrightarrow{M\tilde{\imath}} t', \sigma'$ and $M\tilde{t}' \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma'$.*

Lemma 6.2 is proven by induction over t. The full proof is listed in **??**.

Lemma 6.3 (Soundness of normalisation). *For all concrete expressions e, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}, \tilde{\sigma}', \varphi)$ in $e, \sigma \lessapprox \overline{\tilde{t}, \tilde{\sigma}', \varphi}$, that $M\varphi$ implies $e, \sigma \Downarrow t', \sigma''$ and $M\tilde{t} \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma''$.*

Since Lemma 6.3 makes use of both the striding and the evaluation semantics, we must show soundness for those too.

Lemma 6.4 (Soundness of striding). *For all concrete tasks t, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}', \tilde{\sigma}', \varphi)$ in $t, \sigma \mapsto \overline{\tilde{t}', \tilde{\sigma}', \varphi}$, that $M\varphi$ implies $t, \sigma \mapsto t', \sigma'$ and $M\tilde{t}' \equiv t' \wedge M\tilde{\sigma}' \equiv \sigma'$.*

Lemma 6.5 (Soundness of evaluation). *For all concrete expressions e, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{v}, \tilde{\sigma}', \varphi)$ in $e, \sigma \lesssim \overline{\tilde{v}, \tilde{\sigma}', \varphi}$, that $M\varphi$ implies $e, \sigma \downarrow v, \sigma' \wedge M\tilde{v} \equiv v \wedge M\tilde{\sigma}' \equiv \sigma'$.*

The full proofs of Lemmas 6.3 to 6.5 are listed in the appendix.

6.2 Completeness

We also want to show that for every concrete execution, a symbolic one exists.

To state this Theorem, we require a simulation relation $i \sim j$, which means that the symbolic input i follows the same direction as the concrete input j. This relation is defined below.

Definition 6.6 (Input simulation). A symbolic input $\tilde{\imath}$ simulates a concrete input i denoted as $\tilde{\imath} \sim i$ in the following cases.
$s \sim a$, where s is a symbol and a a concrete action.
$\tilde{\imath} \sim i \supset \mathsf{F}\,\tilde{\imath} \sim \mathsf{F}\, i$
$\tilde{\imath} \sim i \supset \mathsf{S}\,\tilde{\imath} \sim \mathsf{S}\, i$

This allows us to define the completeness property as listed in Theorem 6.7.

Theorem 6.7 (Completeness of interact). *For all concrete tasks t, concrete states σ and concrete inputs i such that $t, \sigma \overset{i}{\Rightarrow} t', \sigma'$ there exists an $\tilde{\imath} \sim i$, $\tilde{t}$, $\tilde{\sigma}$ and φ such that $(\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi)$ in $t, \sigma \overset{\sim}{\rightsquigarrow} \overline{\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi}$.*

The proof of Theorem 6.7 is rather simple. We show that handling is complete (Lemma 6.8) and that the subsequent normalisation is complete (Lemma 6.9).

Lemma 6.8 (Completeness of handling). *For all concrete tasks t, concrete states σ and concrete inputs i such that $t, \sigma \xrightarrow{i} t', \sigma'$ there exists an $\tilde{\imath} \sim i$, $\tilde{t}$, $\tilde{\sigma}$ and φ such that $(\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi)$ in $t, \sigma \rightsquigarrow \overline{\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi}$.*

Lemma 6.8 is proved by induction over t. We only need to show that every concrete execution is also a symbolic one. The only change needed to convert from concrete to symbolic is the adaption of the input.

Since handling makes use of normalisation and evaluation, we need to prove that they too are complete. These properties are listed in Lemmas 6.9 and 6.10

Lemma 6.9 (Completeness of normalisation). *For all concrete expressions e and concrete states σ such that $e, \sigma \Downarrow t, \sigma$ there exists a symbolic execution result $(t, \sigma, \mathsf{True})$ in $e, \sigma \lessapprox \overline{\tilde{t}, \tilde{\sigma}, \varphi}$.*

Lemma 6.10 (Completeness of evaluation). *For all concrete expressions e and concrete states σ such that $e, \sigma \downarrow v, \sigma$ there exists a symbolic execution result $(v, \sigma, \mathsf{True})$ in $e, \sigma \gtrsim \overline{\tilde{v}, \tilde{\sigma}, \varphi}$.*

Lemmas 6.9 and 6.10 follow trivially, since every concrete execution in these semantics is also a symbolic one.

7 RELATED WORK

Symbolic execution. Symbolic execution [Boyer et al. 1975; King 1975] is typically being applied to imperative programming languages, for example Bucur et al. [2014] prototype a symbolic execution engine for interpreted imperative languages. Cadar et al. [2008] use it to generate test cases for programs that can be compiled to LLVM byte-code. Jaffar et al. [2012] use it for verifying safety properties of C programs.

In recent years it has been used for functional programming languages as well. To name some examples, there is ongoing work by Hallahan et al. [2017] and Xue [2019] to implement a symbolic execution engine for Haskell. Giantsios et al. [2017] use symbolic execution for a mix of concrete and symbolic testing of programs written in a subset of Core Erlang. Their goal is to find executions that lead to a runtime error, either due to an assertion violation or an unhandled exception. Chang et al. [2018] present a symbolic execution engine for a typed lambda calculus with mutable state where only some language constructs recognize symbolic values. They claim that their approach is easier to implement than full symbolic execution and simplifies the burden on the solver, while still considering all execution paths.

The difficulty of symbolic execution for functional languages lies in symbolic higher-order values, that is functions as arguments to other functions. Hallahan et al solve this with a technique called *defunctionalization*, which requires all source code to be present, so that a symbolic function can only be one of the present lambda expressions or function definitions. Giantosis et al also require all source code to be present, but they only analyze first-order functions. They can execute higher-order functions, but only with concrete arguments. Our method also requires closed well-typed terms, so we never execute a higher-order function in isolation. Furthermore, we currently do not allow functions and tasks as task values. Together, this means that symbolic values can never be functions.

Contracts. Another method for guaranteeing correctness of programs are *contracts*. Contracts refine static types with additional conditions. They are enforced at runtime. Contracts were first presented by Meyer [1992] for the Eiffel programming language. Findler and Felleisen [2002] applied this technique to functional programming by implementing a contract checker for Scheme. Their contracts are assertions for higher-order programs. Contracts can be used to specify properties more fine-grained than what a static type system could check. It is possible, for example, to refine the arguments or return values of functions to numbers in a certain range, to positive numbers or non-empty lists.

Nguyen et al. [2017] combine contracts and symbolic execution to provide *soft contract checking*. The two ideas go hand in hand in that contracts aid symbolic execution with a language for specifications and properties for symbolic values, and symbolic execution provides compile-time guarantees and test case generation. They present a prototype implementation to verify Racket programs.

Axiomatic program verification. One of the classical methods of proving partial correctness of programs is Hoare's axiomatic approach [Hoare 1969], which is based on pre- and postconditions. See Nielson and Nielson [1992] for a nice introduction to the topic. The axiomatic approach is usually applied to imperative programs, requires manually stating loop invariants, and manually carrying out proofs.

Some work has been done to bring the axiomatic method to functional programming. The current state of SMT solving allows for automated extraction and solving of a large amount of proof obligations. Notable works in this field are for example the Hoare Type Theory by Nanevski et al. [2006], the Hoare and Dijkstra Monads by Nanevski et al. [2008]; Swamy et al. [2013], or the Hoare logic for the state monad by Swierstra [2009].

The difference between the work cited here and our work is that the axiomatic method focuses on stateful computations, while we try to incorporate input as well.

8 CONCLUSION

In this paper, we have demonstrated how to apply symbolic execution to $\widehat{\text{TOP}}$ to verify individual programs. We have developed both a formal system and an implementation of a symbolic execution semantics. Our approach has been validated by proving the formal system correct, and by running the implementation on example programs. For these two example programs, a subsidy request workflow and a flight booking workflow, we have verified that they adhere to their specifications.

8.1 Future work

There are many ways in which we would like to continue this line of work.

First, we believe that more can be done with symbolic execution. Our current approach only allows proving predicates over task results and input values. We cannot, however, prove properties that depend on the order of the inputs. Since the symbolic execution currently returns a list of symbolic inputs, we think this extension is feasible.

Second, our symbolic execution only applies to $\widehat{\text{TOP}}$. We would like to see if we can fit it to iTasks. This poses several challenges. iTasks does not have a formal semantics in the sense that $\widehat{\text{TOP}}$ has. The current implementation in Clean is the closest thing available

to a formal specification. There are also a few language features in iTasks that are not covered by $\widehat{\text{TOP}}$, for example loops.

Third, we would like to apply different kinds of analyses altogether. Can a certain part of the program be reached? Does a certain property hold at every point in the program? Are two programs equal? And what does it mean for two programs to be equal? We think that these properties require a different approach.

ACKNOWLEDGMENTS

This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

REFERENCES

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT - a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 239–254. https://doi.org/10.1145/2541940.2541977

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings 8th USENIX Symposium on Operating Systems Design and Implementation (8th OSDI'08)*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, San Diego, California, USA, 209–224.

Stephen Chang, Alex Knauth, and Emina Torlak. 2018. Symbolic types for lenient symbolic execution. *PACMPL* 2, POPL (2018), 40:1–40:29.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ICFP*. 48–59.

Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2017. Concolic testing for functional languages. *Sci. Comput. Program.* 147 (2017), 109–134. https://doi.org/10.1016/j.scico.2017.04.008

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'19*. 411–424.

William T. Hallahan, Anton Xue, and Ruzica Piskac. 2017. Building a Symbolic Execution Engine for Haskell. In *Proceedings of TAPAS 17*.

Tony Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM: Communications of the ACM* 12 (1969).

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. 2012. TRACER: A Symbolic Execution Tool for Verification. In *Computer Aided Verification (23ed CAV'12)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Lecture Notes in Computer Science (LNCS), Vol. 7358. Springer-Verlag (New York), Berkeley, CA, USA, 758–766.

JM Jansen, F Bolderheij, et al. 2018. Dynamic Resource and Task Management. In *NL ARMS Netherlands Annual Review of Military Studies 2018*. Springer, 91–105.

James C. King. 1975. A New Approach to Program Testing. *SIGPLAN Not.* 10, 6 (April 1975), 228–233. https://doi.org/10.1145/390016.808444

Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop, RWDSL@CGO 2018, Vienna, Austria, February 24-24, 2018*. ACM, 4:1–4:11. https://doi.org/10.1145/3183895.3183902

Bas Lijnse, Jan Martin Jansen, Rinus Plasmeijer, et al. 2012. Incidone: A task-oriented incident coordination tool. In *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM*, Vol. 12.

Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and separation in Hoare type theory. *ACM SIGPLAN Notices* 41, 9 (Sept. 2006), 62–73. https://doi.org/10.1145/1159803.1159812

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: dependent types for imperative programs. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 229–240. https://doi.org/10.1145/1411204.1411237

Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2017. Higher order symbolic execution for contract verification and refutation. *J. Funct. Program* 27 (2017).

Hanne Riis Nielson and Flemming Nielson. 1992. *Semantics With Applications - A Formal Introduction*. Wiley.

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2012. Task-oriented programming in a pure functional language. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, Danny De Schreye, Gerda Janssens, and Andy King (Eds.). ACM, 195–206. https://doi.org/10.1145/2370776.2370801

Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A formal foundation for task-oriented programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019.*, Ekaterina Komendantskaya (Ed.). ACM, 17:1–17:13. https://doi.org/10.1145/3354166.3354182

Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2017. Maintaining Separation of Concerns Through Task Oriented Software Development. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK*.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 387–398. https://doi.org/10.1145/2491956.2491978

Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Vol. 5674. Springer, 440–451.

Anton Xue. 2019. Lazy Counterfactual Symbolic Execution. *(in submission)* (2019).