Revision 2 of CoLiS language: formal syntax, semantics, concrete and symbolic interpreters

Benedikt Becker, Nicolas Jeannerod, Claude Marché, Ralf Treinen

CoLiS project

http://colis.irif.fr/

Deliverable 3.3 - Revision 2 of CoLiS language: formal syntax, semantics, concrete and symbolic interpreters

B. Becker, C. Marché, N. Jeannerod, R. Treinen

October 2019

Contents

This report aims at providing the detailed technical informations about the second version of the CoLiS language, designed in the context of the CoLiS project.

The first version was presented in 2017 [3]. This second version extends the language by some constructs that were identified as missing for the analysis of installation shell scripts of Debian packages. Moreover, a few important fix were made in the formal semantics. Section 1 summarizes the changes made with respect to the first version of the language.

As for the first version, the second version of the CoLiS language is formalized using the Why3 environment for formal design and verification [2]. Section 2 details the formalisation and the proof of a *concrete interpreter* for CoLiS, which is used for executing CoLiS scripts in a concrete state of a file system. Section 3 is dedicated to the formalisation of a *symbolic interpreter*, aiming at executing a script in a *symbolic* state, which represents a (possibly infite) set of concrete states by a symbolic formula. The symbolic interpreter is a basic block for a static analysis of the execution of a script.

The implementation of this version 2 of the CoLiS language is available in the repository `https://github.com/colis-anr/colis-language`. The implementation is heavily used in current experimentations of analysis of Debian installation scripts. Data and conclusions of those experimentations are reported in a separate document (under preparation).

For completeness, the appendix of this document presents the complete syntax of the CoLiS language and the complete set of its semantic rules. It is worth to notice that the set of semantics rules is in fact automaticcaly generated from the Why3 implementation, so as to guarantee that the present documentation is accurate. Technically this is obtain by the Why3 command

```
why3 pp --output=latex --kind=inductive
```

which was implemented in Why3 for this precise purpose, and is now distributed with Why3 itself for convenience of other users of Why3.

1 Summary of Changes and Fixes

We detail the additions and changes with respect to the first CoLiS version.

1.1 Changes and Additions in the syntax

The new complete syntax is given in Appendix A.

Distinction between function calls and utility The syntax now makes an explicit distinction between a call to a function in the script, and a call to a utility (e.g., `mkdir`, `rmdir`, `mv`, `cp`). Syntactically, a function call is marked by the keyword `call`, whereas a utility is called just by its name and arguments. For example:

```
call f "a" "b"  # a function call
rm "-f" "*.log" # a call to "rm" utility
```

In other words, when writing a CoLiS script the user must explicitly tell when they intend to call a function or a system's utility.

Exporting variables As in the POSIX shell, an explicit instruction of the form

```
export VAR
```

can be used to export variable VAR, that is to transmit it to sub-shells or utilities.

Built-in instruction for changing current directory Due to the specific nature of `cd`, we added a instruction in the CoLiS language

```
cd "/new/path"
```

Discarding standard output For the purpose of translating a shell script that redirects standard output to `/dev/null` or standard error, we added the construct

```
nooutput <instruction>; ... ; <instruction> endnooutput
```

to discard the output of the given instructions.

Result states The `exit` and `return` instructions now receive as argument one of three possible values: `success`, `failure` or `previous`, grosso-modo corresponding to `exit 0`, `exit 1` and `exit $?`.

Last but not least, we reconsidered the distinction between string expressions and list expressions. The keyword `split` was introduced to explicitly ask for the splitting of a string expression when used as a list fragment.

1.2 Changes and Additions in the Semantics

Several modifications to the semantics were required to support the syntactic changes described above. This update to the CoLiS language makes two major changes in the semantics:

Strict mode The first important change is a correction of the behaviour of a failing instructions and utilties when inside or outside of a test of an `if` statement or a loop. The first version of the semantics did not correspond to the shell semantics in the so-called strict mode (invoked by `set -e`). For example, for the code below

```
set -e

foo () {
  false ;
  echo 'here'
}

if foo ; then
    echo 'yes'
else
    echo 'no'
fi
```

the shell semantics produces the output

```
here
yes
```

with exit code 0, whereas the first version of CoLiS semantics was producing no output and exit code 1. The mistake was to evaluate `false` always as an equivalent to `exit 1`, whereas in the shell that "strict" mode is disabled when under conditions.

The semantics of CoLiS was changed to match the behavior of the shell by adding a boolean flag in the evaluation context to tell whether the evaluation must be done in strict mode or not. Failing utilities (including `false`) only trigger an error when in strict mode and not when the failure happend within a test.

Bound on the number of loop iterations The second important change in the semantics was required for modelling the symbolic execution. In that context, we have to limit a priori the number of loop iterations to make symbolic execution terminating. This was realised by an additional parameter in the execution judgement: the desired bound on the number of iterations, that may be either a non-negative number, or ∞ to denote unbounded execution.

The principle of using a bound on the number of loop iterations is described in more detail in a paper showing a formalization of a symbolic interpreter for a basic IMP language [1]. In the case of the CoLiS language, which accepts recursive functions, we also added a bound on the number of nested (recursive) function calls. The resulting semantic judgements and rules are given in Appendix B.

Prover	VCs	Fastest	Slowest	Average
CVC4 1.6	230	0.10	0.68	0.26
Alt-Ergo 2.2.0	13	0.12	2.45	0.54
Z3 4.6.0	2	0.12	0.25	0.18

Table 1: The use of different automatic theorem provers in the verification conditions (VCs) of the concrete interpreter with processing time in seconds.

2 Concrete Interpreter

A concrete interpreter of CoLiS script was already formalized and proved for the first version of CoLiS [3]. With the multiple changes in the syntax and semantics, this interpreter must have been change accordingly, but also its formal specification had to be updated, and the formal proof updated too.

We give below the formal contract for main function of that interpreter, as is written in the programming language of Why3. The correctness of that interpreter is expressed as formal post-conditions as follows.

```
let ghost ref s = 0

let rec interp_instruction (I:input) (sta:concrete_state) (i:instruction) : unit
 requires { I.loop-limit = ∞ ∧ I.stack-size = ∞ }
 ensures { s = (old s) }
 ensures { (I, C(old sta), S(old sta)), i ⇓ᴵ_old s (S(sta), C(sta), Normal)) }
 raises { Exit|Return as ξ → (I, C(old sta), S(old sta)), i ⇓ᴵ_old s (S(sta), C(sta), ξ)) }
 diverges
= ...
```

The main thing to note is that for the concrete interpreter, we don't set limits to the number of loop iterations and functions calls (as shown in the precondition). In essence, the normal post-condition expresses that the instruction evaluates normally, that is produces a normal behaviour in the sense of the formal semantics, and moreover the relation between the initial and the final states are allowed by the formal semantics. The interpret may also raise exceptions `Exit` or `Return`, and they are associated with exceptional post-conditions which tells that in that case the formal semantics accordingly says that the instruction produces an abnormal behaviour, and again the initial and final state are correctly related by the formal semantics.

Notice however that this specification says nothing about termination, indeed the interpreter will loop in presence of infinite loop or infinite recursion, and in that case the contract says nothing.

The proofs had to be updated accordingly, and Table 1 summarizes the VCs and with which prover they are discharged.

3 Symbolic Interpreter

The design and formalization of the symbolic interpreter follows as general approach that was described on a simple IMP language [1]. We reused from that other work all the ideas about bounding the number of loop iterations, and also the usage of ghost code so as to express the important expected properties *over-approximation* and *under-approximation*. In essence, over-approximation is the property that tells that a symbolic execution from a given symbolic state Σ covers all concrete execution that start from a concrete state instance of Σ, where the under-approximation property tells that ni spurious concrete execution is covered by symbolic execution.

First one should notice that CoLiS proposes significantly more control structures than IMP which has only sequence, `if` and `while` [1]. Yet, the extension to other control structures could be done routinely: they pose no specific issues that where not already consider for supporting `if` statements or `while` loops. Hence, the major difference between the symbolic interpreter for CoLiS and the symbolic interpreter for IMP in relies on the data types: for IMP we only considered integer variables. In CoLiS the variables are holding string values or list of strings value, but they are not the major source of

Prover	VCs	Fastest	Slowest	Average
CVC4 1.6	702	0.13	1.92	0.48
Alt-Ergo 2.2.0	65	0.13	43.26	2.60
Z3 4.6.0	13	0.07	1.14	0.31

Table 2: The use of different automatic theorem provers in the verification conditions of the symbolic interpreter functions with processing time in seconds.

novelty: the novelty is that a script is operating on a mutable program state that contain *all the file system*.

In a symbolic interpreter engine, the data types must be supported by the constraint language that expresses the set of possible values for variables. A specificity of the context of analysing Debian installation scripts is that we know concretely the values of the variables. It is only the file sysmtem which is not known concretely, so the file system is the only part that must be treated symbolically. For that purpose, we had to use a contraint language dedicated to tree structures [5].

To summarize, our symbolic engine for CoLiS relies on a constraint engine dedicated to the file system. The semantics of utility is encoded in tree constraints as described in another CoLiS deliverable [4]. The handling of the control structures is then done in a similar way as for the IMP language. It should be noted that the symbolic engine is highly generic with respect to the specification of utilities.

The formal contract for the main function of interpreting CoLiS instructions is as follows

```
let rec sym_interp_instruction (s) (I,C,S) (i : instruction) : set of (S' × C')_β
  requires { s ≤ I.stack-size ≠ ∞ ∧ I.loop-limit ≠ ∞ }
  variant { I.stack-size − s, size(s), −1 }
  ensures { ∀ S',C',β. (I,C,S),i ⇓ˡ_s (S',C',β) → (S',C')_β ∈ result }
= ...
```

The symbolic interpreter for instructions takes the current stack height, a triple of an evaluation input (comprising the maximal stack height and loop limit), context, and state, and an instruction as argument and returns a set of pairs of result context and state, indexed by a behavior. The limits have to be finite to ensure the termination of the symbolic interpreter, where in each recursive call the remaining stack size, the size of the instruction, or the remaining loop iterations decrease (as stated in the variant). The post-condition expresses the over-approximation property: any concrete execution is covered by the symbolic execution. This property ensures that if the symbolic interpreter tells that no undesired symbolic states can be reached, then it is also true that no undesired concrete states can be reached by any concrete execution.

Statistics about the proof of the symbolic engine are given on Table 2.

4 Conclusions and Future Work

The version 2 of the CoLiS language, described in this report, is the one currently used for analysing installation scenarios of installation scripts.

It is likely that a few minor constructs will need to be added, such as the extension to parameter expansion (https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_06_02). Moreover the constraint language to model the filesystem will probably need to be modified. Though, these modifications are orthogonal to the current formalization of the semantics and only minor modifications are still expected, if any.

References

[1] Benedikt Becker and Claude Marché. Ghost code in action: Automated verification of a symbolic interpreter. In Supratik Chakraborty and Jorge A.Navas, editors, *Verified Software: Tools, Techniques*

and Experiments, Lecture Notes in Computer Science, New York, United States, July 2019. https://hal.inria.fr/hal-02276257.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. http://hal.inria.fr/hal-00967132/en.

[3] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A formally verified interpreter for a shell-like programming language. In Andrei Paskevich and Thomas Wies, editors, *Verified Software: Theories, Tools, and Experiments. Revised Selected Papers Presented at the 9th International Conference VSTTE*, number 10712 in Lecture Notes in Computer Science, Heidelberg, Germany, December 2017. Springer. https://hal.inria.fr/.

[4] Nicolas Jeannerod, Yann Régis-Gianas, Claude Marché, Mihaela Sighireanu, and Ralf Treinen. Specification of UNIX utilities. Technical report, HAL Archives Ouvertes, October 2019. https://hal.inria.fr/hal-02321691.

[5] Nicolas Jeannerod and Ralf Treinen. Deciding the first-order theory of an algebra of feature trees with updates. In *9th International Joint Conference on Automated Reasoning*, Oxford, United Kingdom, July 2018. https://hal.archives-ouvertes.fr/hal-01807474.

A Syntax

⟨*program*⟩ ::=
    ⟨*function-definition*⟩*
    `begin`
      ⟨*instr*⟩*
    `end` ⟨*eof*⟩

⟨*function-definition*⟩ ::=
    `function` ⟨*identifier*⟩ `begin` — *No argument names, use* `arg` *n*
      ⟨*instr*⟩*
    `end`

⟨*instr*⟩ ::= — *Program instruction*
 | ⟨*identifier*⟩ `:=` ⟨*string-expr*⟩ — *Variable assignment*
 | `export` ⟨*identifier*⟩ — *Export variable*
 | `cd` ⟨*string-expr*⟩ — *Change working directory*
 | `nooutput` ⟨*instr*⟩* `endnooutput` — *Suppress output*
 | `begin` ⟨*instr*⟩* `end` — *Group instrs*
 | `not` ⟨*instr*⟩ — *Negation of result*
 | `if` ⟨*instr*⟩ `then` ⟨*instr*⟩* `fi` — *Conditional*
 | `if` ⟨*instr*⟩ `then` ⟨*instr*⟩* `else` ⟨*instr*⟩* `fi`
 | `for` ⟨*identifier*⟩ `in` ⟨*list-expr*⟩ `do` ⟨*instr*⟩* `done` — *For loop*
 | `while` ⟨*instr*⟩ `do` ⟨*instr*⟩* `done` — *While loop*
 | `process` ⟨*instr*⟩* `endprocess` — *Subscope*
 | `pipe` ⟨*instr*⟩ (`into` ⟨*instr*⟩)* `endpipe` — *Pipe*
 | `call` ⟨*identifier*⟩ ⟨*list-expr*⟩? — *Function call*
 | ⟨*identifier*⟩ ⟨*list-expr*⟩? — *Utility call*
 | `exit` ⟨*result*⟩ — *Exit program*
 | `return` ⟨*result*⟩ — *Return from function*
 | `shift` ⟨*nat*⟩? — *Shift arguments list*

⟨*result*⟩ ::= — *Result for* `exit`/`return`
 | `success`
 | `failure`
 | `previous` — *Result of previous instr*

⟨*sfrag*⟩ ::= — *String fragment*
 | ⟨*literal*⟩ — *String literal*
 | ⟨*identifier*⟩ — *Variable*
 | `embed {` ⟨*instr*⟩ `}` — *Output from instr*
 | `arg` ⟨*nat*⟩ — *Program/function argument by index*

⟨*string-expr*⟩ ::= ⟨*sfrag*⟩+ — *String expression*

⟨*lfrag*⟩ ::= `split?` ⟨*string-expr*⟩ — *List fragment*

⟨*list-expr*⟩ ::= `[` (⟨*lfrag*⟩ (`,` ⟨*lfrag*⟩)*)? `]` — *Non-empty, bracket-delimited, comma-separated list of fragments*

B Semantic judgements and rules

$$
\begin{array}{lrll}
\text{Behaviour:} & \beta = & \mathsf{Normal} & \text{— Behaviours of an instruction} \\
& & \mid \mathsf{Return} & \\
& & \mid \mathsf{Exit} & \\
& & \mid \mathsf{Failure} & \text{— Loop limit or stack exceeded} \\
\text{Result:} & \mathit{result}\ \alpha = & \mathsf{Success}\ \alpha \mid \mathsf{Error} & \text{— Possibly failing result of type } \alpha \\
\text{Option:} & \mathit{option}\ \alpha = & \mathsf{Some}\ \alpha \mid \mathsf{None} & \text{— Optional value of type } \alpha \\
\text{List:} & \mathit{list}\ \alpha = & \alpha :: (\mathit{list}\ \alpha) \mid [\,] & \text{— List of values of type } \alpha
\end{array}
$$

$$
\begin{array}{lrll}
\text{Top-down context:} & \mathcal{I} = \{ & \mathsf{under\text{-}condition} : \mathbb{B}\ ; & \text{— Evaluation under test/condition} \\
& & \mathsf{argument}_0 : \mathbb{S}\ ; & \text{— First argument} \\
& & \mathsf{loop\text{-}limit} : \mathbb{N} \cup \{\infty\}\ ; & \text{— Maximal loop count} \\
& & \mathsf{stack\text{-}size} : \mathbb{N} \cup \{\infty\}\ \} & \text{— Maximal stack size} \\
\text{Program context:} & \mathcal{C} = \{ & \mathsf{vars} : \mathit{var} \to \mathit{option}\ \mathbb{S} \times \mathbb{B}\ ; & \text{— Variable environment} \\
& & \mathsf{funs} : \mathit{var} \to \mathit{option}\ \mathit{instruction}\ ; & \text{— Function environment} \\
& & \mathsf{arguments} : \mathit{list}\ \mathbb{S}\ ; & \text{— Other arguments} \\
& & \mathsf{result} : \mathbb{B}\ ; & \text{— (Previous) result} \\
& & \mathsf{cwd} : \mathit{cwd}\ \} & \text{— Current working directory (abstract)} \\
\text{System state:} & \mathcal{S} = \{ & \mathsf{file\text{-}system} : \mathit{file\text{-}system}\ ; & \text{— File system (abstract)} \\
& & \mathsf{stdin} : \mathit{stdin}\ ; & \text{— Standard input} \\
& & \mathsf{stdout} : \mathit{stdout}\ \} & \text{— Standard output}
\end{array}
$$

$$
\begin{array}{lr}
\text{Instruction:} & (\mathcal{I},\mathcal{C},\mathcal{S}), i \Downarrow^{\mathsf{I}}_{s} (\mathcal{S}',\mathcal{C}',\beta) \\
\text{List expression:} & (\mathcal{I},\mathcal{C},\mathcal{S}), le \Downarrow^{\mathsf{L}}_{s} (\mathcal{S}', \mathit{result}\ (\mathit{list}\ \mathbb{S})) \\
\text{String expression:} & s, b, (\mathcal{I},\mathcal{C},\mathcal{S}), se \Downarrow^{\mathsf{S}} (\mathcal{S}', \mathit{result}\ (\mathbb{S},\mathbb{B})) \\
\text{While-loop:} & b, (\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{while}\ i\ i \Downarrow^{\mathsf{W}}_{s,n} (\mathcal{S}',\mathcal{C}',\beta), b' \\
\text{For-loop:} & b, (\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{for}\ id\ le\ i \Downarrow^{\mathsf{F}}_{s} (\mathcal{S}',\mathcal{C}',\beta), b' \\
\text{Program:} & (\mathcal{I},\mathcal{C},\mathcal{S}), p \Downarrow^{\mathsf{P}} (\mathcal{S}', \mathit{result}\ \mathbb{B}) \\
& \mathcal{C}, \mathit{defs} \Downarrow^{\mathsf{D}} \mathcal{C}'
\end{array}
$$

The semantic judgements define that an evaluation has behaviour $\beta = \mathsf{Failure}$ if the loop limit or stack height is exceeded. The variable environment maps identifiers to pairs composed of an optional string value of the variable and its export status, with default $(\mathsf{None}, \mathsf{False})$. The filesystem (and current working directory) is left abstract in the semantics because the changes to the filesystem are not introduced by language constructs but only by the utilities that are not covered by the semantics.

Auxiliary definitions

A number of auxiliary functions and notations are used in the formalisation of the semantic rules:

- $o \,||\, x$: get value from option o, or x by default
- $r \,||\, b$: result r as boolean or b if $r = \texttt{previous}$
- $f[id \leftarrow v]$: Update function f for id by value v
- $vars[id]'$: Get string value of identifier id in $vars$ (and ignore its boolean export flag)
- $vars[id \leftarrow s]'$: Update string value of id in $vars$ to s, retaining the boolean export flag (or False)
- inject a return value into a behaviour

$$
bhv_{\mathcal{I}}(b) = \begin{cases} \mathsf{Normal} & \text{when } \mathcal{I}.\mathsf{under\text{-}condition} = \mathsf{True} \vee b = \mathsf{True} \\ \mathsf{Exit} & \text{when } \mathcal{I}.\mathsf{under\text{-}condition} = \mathsf{False} \wedge b = \mathsf{False} \end{cases}
$$

- $\mathit{filter}(vars)$: Create a mapping of bindings in $vars$ that are marked as $\mathsf{export} = \mathsf{True}$.

B.1 Evaluation of instructions, $\Downarrow^{\mathsf{I}}$

EXIT
$$\frac{\mathcal{C}' = \{\mathcal{C} \texttt{ with result=}c \,||\, \mathcal{C}.\mathsf{result}\}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{exit } c \Downarrow^{\mathsf{I}}_s (\mathcal{S},\mathcal{C}',\mathsf{Exit})}$$

RETURN
$$\frac{\mathcal{C}' = \{\mathcal{C} \texttt{ with result=}c \,||\, \mathcal{C}.\mathsf{result}\}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{return } c \Downarrow^{\mathsf{I}}_s (\mathcal{S},\mathcal{C}',\mathsf{Return})}$$

SHIFT
$$\frac{\mathsf{shift}(1 \,||\, bn, \mathcal{C}.\mathsf{arguments}) = \mathsf{Some}\ args \qquad \mathcal{C}' = \{\mathcal{C} \texttt{ with arguments=}args\texttt{;result=True}\} \qquad \beta = bhv_{\mathcal{I}}(\mathcal{C}'.\mathsf{result})}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{shift } bn \Downarrow^{\mathsf{I}}_s (\mathcal{S},\mathcal{C}',\beta)}$$

SHIFT-ERROR
$$\frac{\mathsf{shift}(1 \,||\, bn, \mathcal{C}.\mathsf{arguments}) = \mathsf{None} \qquad \mathcal{C}' = \{\mathcal{C} \texttt{ with result=False}\} \qquad \beta = bhv_{\mathcal{I}}(\mathcal{C}'.\mathsf{result})}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{shift } bn \Downarrow^{\mathsf{I}}_s (\mathcal{S},\mathcal{C}',\beta)}$$

EXPORT
$$\frac{varval = \{\mathcal{C}.\mathsf{vars}[id] \texttt{ with export=True}\} \qquad vars = \mathcal{C}.\mathsf{vars}[id \leftarrow varval] \qquad \mathcal{C}' = \{\mathcal{C} \texttt{ with vars=}vars\texttt{;result=True}\}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{export } id \Downarrow^{\mathsf{I}}_s (\mathcal{S},\mathcal{C}',\mathsf{Normal})}$$

CD-ARG-FAILURE
$$\frac{s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), se \Downarrow^{\mathsf{S}} (\mathcal{S}', \mathsf{Error})}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{cd } se \Downarrow^{\mathsf{I}}_s (\mathcal{S}',\mathcal{C},\mathsf{Failure})}$$

CD-NO-DIR
$$\frac{\begin{array}{c} s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), se \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Success}\,(s,b)) \\ cwd = \mathit{norm\text{-}path}_{\mathcal{C}.\mathsf{cwd}}(s) \qquad \mathit{interp}\,(\mathcal{C}.\mathsf{cwd}, \mathit{filter}(\mathcal{C}.\mathsf{vars}), \texttt{"-d"} :: cwd :: \texttt{[]})\,(\texttt{"test"}, \mathcal{S}_1) = (\mathcal{S}_2, \mathsf{False}) \\ \mathcal{C}_1 = \{\mathcal{C} \texttt{ with result=False}\} \qquad \beta = bhv_{\mathcal{I}}(\mathsf{False}) \end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{cd } se \Downarrow^{\mathsf{I}}_s (\mathcal{S}_2,\mathcal{C}_1,\beta)}$$

CD
$$\frac{\begin{array}{c} s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), se \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Success}\,(s,b)) \\ cwd = \mathit{norm\text{-}path}_{\mathcal{C}.\mathsf{cwd}}(s) \qquad \mathit{interp}\,(\mathcal{C}.\mathsf{cwd}, \mathit{filter}(\mathcal{C}.\mathsf{vars}), \texttt{"-d"} :: cwd :: \texttt{[]})\,(\texttt{"test"}, \mathcal{S}_1) = (\mathcal{S}_2, \mathsf{True}) \\ \mathcal{C}_1 = \{\mathcal{C} \texttt{ with result=True;vars=}\mathcal{C}.\mathsf{vars}[\texttt{"PWD"} \leftarrow cwd]'\texttt{;cwd=}cwd\} \qquad \beta = bhv_{\mathcal{I}}(\mathsf{True}) \end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{cd } se \Downarrow^{\mathsf{I}}_s (\mathcal{S}_2,\mathcal{C}_1,\beta)}$$

ASSIGNMENT
$$\frac{s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), e \Downarrow^{\mathsf{S}} (\mathcal{S}', \mathsf{Success}\,(s,b)) \qquad \mathcal{C}' = \{\mathcal{C} \texttt{ with vars=}\mathcal{C}.\mathsf{vars}[id \leftarrow s]'\texttt{;result=}b\} \qquad \beta = bhv_{\mathcal{I}}(\mathcal{C}'.\mathsf{result})}{(\mathcal{I},\mathcal{C},\mathcal{S}), id := e \Downarrow^{\mathsf{I}}_s (\mathcal{S}',\mathcal{C}',\beta)}$$

ASSIGNMENT-FAILURE
$$\frac{s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), e \Downarrow^{\mathsf{S}} (\mathcal{S}', \mathsf{Error})}{(\mathcal{I},\mathcal{C},\mathcal{S}), id := e \Downarrow^{\mathsf{I}}_s (\mathcal{S}',\mathcal{C},\mathsf{Failure})}$$

SEQUENCE
$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}), i_1 \Downarrow^{\mathsf{I}}_s (\mathcal{S}_1,\mathcal{C}_1,\mathsf{Normal}) \qquad (\mathcal{I},\mathcal{C}_1,\mathcal{S}_1), i_2 \Downarrow^{\mathsf{I}}_s (\mathcal{S}_2,\mathcal{C}_2,\beta_2)}{(\mathcal{I},\mathcal{C},\mathcal{S}), i_1 \ ; \ i_2 \Downarrow^{\mathsf{I}}_s (\mathcal{S}_2,\mathcal{C}_2,\beta_2)}$$

SEQUENCE-ABORT
$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}), i_1 \Downarrow^{\mathsf{I}}_s (\mathcal{S}_1,\mathcal{C}_1,\beta) \qquad \beta \neq \mathsf{Normal}}{(\mathcal{I},\mathcal{C},\mathcal{S}), i_1 \ ; \ i_2 \Downarrow^{\mathsf{I}}_s (\mathcal{S}_1,\mathcal{C}_1,\beta)}$$

SUBSHELL

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}),i \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}',\beta) \qquad \beta \neq \mathsf{Failure} \qquad \mathcal{C}'' = \{\mathcal{C} \texttt{ with result=}\mathcal{C}'.\mathsf{result}\} \qquad \beta' = bhv_{\mathcal{I}}(\mathcal{C}'.\mathsf{result})}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{process } i \texttt{ endprocess} \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}'',\beta')}$$

SUBSHELL-FAILURE

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}),i \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}',\mathsf{Failure}) \qquad \mathcal{C}'' = \{\mathcal{C} \texttt{ with result=}\mathcal{C}'.\mathsf{result}\}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{process } i \texttt{ endprocess} \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}'',\mathsf{Failure})}$$

NOT

$$\frac{\begin{array}{c}(\{\mathcal{I} \texttt{ with under-condition=True}\},\mathcal{C},\mathcal{S}),i \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}',\beta) \\ \beta = \mathsf{Normal} \vee \beta = \mathsf{Return} \qquad \mathcal{C}'' = \{\mathcal{C}' \texttt{ with result=}\neg\,\mathcal{C}'.\mathsf{result}\}\end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{not } i \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}'',\beta)}$$

NOT-TRANSMIT

$$\frac{(\{\mathcal{I} \texttt{ with under-condition=True}\},\mathcal{C},\mathcal{S}),i \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}',\beta) \qquad \beta = \mathsf{Exit} \vee \beta = \mathsf{Failure}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{not } i \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}',\beta)}$$

IF-TRUE

$$\frac{\begin{array}{c}(\{\mathcal{I} \texttt{ with under-condition=True}\},\mathcal{C},\mathcal{S}),i_1 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1,\mathcal{C}_1,\mathsf{Normal}) \\ \mathcal{C}_1.\mathsf{result} = \mathsf{True} \qquad (\mathcal{I},\mathcal{C}_1,\mathcal{S}_1),i_2 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_2,\mathcal{C}_2,\beta_2)\end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{if } i_1 \texttt{ then } i_2 \texttt{ else } i_3 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_2,\mathcal{C}_2,\beta_2)}$$

IF-FALSE

$$\frac{\begin{array}{c}(\{\mathcal{I} \texttt{ with under-condition=True}\},\mathcal{C},\mathcal{S}),i_1 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1,\mathcal{C}_1,\mathsf{Normal}) \\ \mathcal{C}_1.\mathsf{result} = \mathsf{False} \qquad (\mathcal{I},\mathcal{C}_1,\mathcal{S}_1),i_3 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_3,\mathcal{C}_3,\beta_3)\end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{if } i_1 \texttt{ then } i_2 \texttt{ else } i_3 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_3,\mathcal{C}_3,\beta_3)}$$

IF-TRANSMIT-CONDITION

$$\frac{(\{\mathcal{I} \texttt{ with under-condition=True}\},\mathcal{C},\mathcal{S}),i_1 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1,\mathcal{C}_1,\beta_1) \qquad \beta_1 \neq \mathsf{Normal}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{if } i_1 \texttt{ then } i_2 \texttt{ else } i_3 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1,\mathcal{C}_1,\beta_1)}$$

NOOUTPUT

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}),i \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C}',\beta) \qquad \mathcal{S}'' = \{\mathcal{S}' \texttt{ with stdout=}\mathcal{S}.\mathsf{stdout}\}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{nooutput } i \Downarrow_s^{\mathsf{I}} (\mathcal{S}'',\mathcal{C}',\beta)}$$

PIPE

$$\frac{\begin{array}{c}\mathcal{S}' = \{\mathcal{S} \texttt{ with stdout=}\varepsilon\} \\ (\mathcal{I},\mathcal{C},\mathcal{S}'),i_1 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1,\mathcal{C}_1,\beta_1) \qquad \beta_1 \neq \mathsf{Failure} \qquad \mathcal{S}_1' = \{\mathcal{S}_1 \texttt{ with stdout=}\mathcal{S}.\mathsf{stdout}\texttt{;stdin=}\mathcal{S}_1.\mathsf{stdout}\} \\ (\mathcal{I},\mathcal{C},\mathcal{S}_1'),i_2 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_2,\mathcal{C}_2,\beta_2) \qquad \mathcal{S}_2' = \{\mathcal{S}_2 \texttt{ with stdin=}\mathcal{S}_1.\mathsf{stdin}\} \qquad \mathcal{C}' = \{\mathcal{C} \texttt{ with result=}\mathcal{C}_2.\mathsf{result}\}\end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{pipe } i_1 \texttt{ into } i_2 \texttt{ endpipe} \Downarrow_s^{\mathsf{I}} (\mathcal{S}_2',\mathcal{C}',\beta_2)}$$

PIPE-FAILURE

$$\frac{(\mathcal{I},\mathcal{C},\{\mathcal{S} \texttt{ with stdout=}\varepsilon\}),i_1 \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1,\mathcal{C}_1,\mathsf{Failure}) \qquad \mathcal{S}_1' = \{\mathcal{S}_1 \texttt{ with stdout=}\mathcal{S}.\mathsf{stdout}\}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{pipe } i_1 \texttt{ into } i_2 \texttt{ endpipe} \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1',\mathcal{C},\mathsf{Failure})}$$

CALL-UTILITY-ARGS-FAILURE

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}),le \Downarrow_s^{\mathsf{L}} (\mathcal{S}',\mathsf{Error})}{(\mathcal{I},\mathcal{C},\mathcal{S}),id\ le \Downarrow_s^{\mathsf{I}} (\mathcal{S}',\mathcal{C},\mathsf{Failure})}$$

CALL-UTILITY

$$\frac{\begin{array}{c}(\mathcal{I},\mathcal{C},\mathcal{S}),le \Downarrow_s^{\mathsf{L}} (\mathcal{S}',\mathsf{Success}\,ss) \\ interp\,(\mathcal{C}.\mathsf{cwd},filter(\mathcal{C}.\mathsf{vars}),ss)\,(id,\mathcal{S}') = (\mathcal{S}'',b) \qquad \mathcal{C}' = \{\mathcal{C} \texttt{ with result=}b\} \qquad \beta = bhv_{\mathcal{I}}(b)\end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}),id\ le \Downarrow_s^{\mathsf{I}} (\mathcal{S}'',\mathcal{C}',\beta)}$$

CALL-FUNCTION-ARGS-FAILURE

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_1, \mathsf{Error})}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{call}\ id\ le \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1, \mathcal{C}, \mathsf{Failure})}$$

CALL-FUNCTION-NOT-FOUND

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}', \mathsf{Success}\, ss) \qquad \mathcal{C}.\mathsf{funs}[id] = \mathsf{None} \qquad \mathcal{C}' = \{\mathcal{C}\ \texttt{with result=False}\} \qquad \beta = bhv_{\mathcal{I}}(\mathsf{False})}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{call}\ id\ le \Downarrow_s^{\mathsf{I}} (\mathcal{S}', \mathcal{C}', \beta)}$$

CALL-FUNCTION-STACK-LIMIT

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_1, \mathsf{Success}\, args) \qquad \mathcal{C}.\mathsf{funs}[id] = \mathsf{Some}\ i \qquad \mathcal{I}.\text{stack-size} = s}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{call}\ id\ le \Downarrow_s^{\mathsf{I}} (\mathcal{S}_1, \mathcal{C}, \mathsf{Failure})}$$

CALL-FUNCTION

$$\frac{\begin{array}{c}(\mathcal{I},\mathcal{C},\mathcal{S}), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_1, \mathsf{Success}\, args) \qquad \mathcal{C}.\mathsf{funs}[id] = \mathsf{Some}\ i \qquad \mathcal{I}.\text{stack-size} \neq s \\ \mathcal{I}_1 = \{\mathcal{I}\ \texttt{with argument}_0\texttt{=}id\} \qquad \mathcal{C}_1 = \{\mathcal{C}\ \texttt{with arguments=}args\} \qquad (\mathcal{I}_1,\mathcal{C}_1,\mathcal{S}_1), i \Downarrow_{s+1}^{\mathsf{I}} (\mathcal{S}_2, \mathcal{C}_2, \beta) \\ \beta' = \texttt{match}\ \beta\ \texttt{with}\ \mathsf{Normal} \mid \mathsf{Return} \rightarrow \mathsf{Normal} \mid \mathsf{Exit} \rightarrow \mathsf{Exit} \mid \mathsf{Failure} \rightarrow \mathsf{Failure} \\ \mathcal{C}' = \{\mathcal{C}_2\ \texttt{with arguments=}\mathcal{C}.\text{arguments}\}\end{array}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{call}\ id\ le \Downarrow_s^{\mathsf{I}} (\mathcal{S}_2, \mathcal{C}', \beta')}$$

FOREACH-ARGS-FAILURE

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}', \mathsf{Error})}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{for}\ id\ \texttt{in}\ le\ \texttt{do}\ i\ \texttt{done} \Downarrow_s^{\mathsf{I}} (\mathcal{S}', \mathcal{C}, \mathsf{Failure})}$$

FOREACH

$$\frac{(\mathcal{I},\mathcal{C},\mathcal{S}), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}', \mathsf{Success}\, ss) \qquad \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}'), \texttt{for}\ id\ ss\ i \Downarrow_s^{\mathsf{F}} (\mathcal{S}'', \mathcal{C}', \beta), b \qquad \mathcal{C}'' = \{\mathcal{C}'\ \texttt{with result=}b\}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{for}\ id\ \texttt{in}\ le\ \texttt{do}\ i\ \texttt{done} \Downarrow_s^{\mathsf{I}} (\mathcal{S}'', \mathcal{C}'', \beta)}$$

WHILE

$$\frac{\mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{while}\ i_1\ i_2 \Downarrow_{s,0}^{\mathsf{W}} (\mathcal{S}', \mathcal{C}', \mathsf{Normal}), b \qquad \mathcal{C}'' = \{\mathcal{C}'\ \texttt{with result=}b\}}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{while}\ i_1\ \texttt{do}\ i_2\ \texttt{done} \Downarrow_s^{\mathsf{I}} (\mathcal{S}', \mathcal{C}'', \mathsf{Normal})}$$

WHILE-ABORT

$$\frac{\beta \neq \mathsf{Normal} \qquad \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{while}\ i_1\ i_2 \Downarrow_{s,0}^{\mathsf{W}} (\mathcal{S}', \mathcal{C}', \beta), b}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{while}\ i_1\ \texttt{do}\ i_2\ \texttt{done} \Downarrow_s^{\mathsf{I}} (\mathcal{S}', \mathcal{C}', \beta)}$$

B.2 Evaluation of list expressions, $\Downarrow^{\mathsf{L}}$

LIST-EXPR-NIL

$$\frac{}{(\mathcal{I},\mathcal{C},\mathcal{S}), \texttt{[]} \Downarrow_s^{\mathsf{L}} (\mathcal{S}, \mathsf{Success}\ \texttt{[]})}$$

LIST-EXPR-FAILURE-HEAD

$$\frac{s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), se \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Error})}{(\mathcal{I},\mathcal{C},\mathcal{S}), (se, sp) :: le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_1, \mathsf{Error})}$$

LIST-EXPR-FAILURE-TAIL

$$\frac{s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), se \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Success}\, (s, b_1)) \qquad (\mathcal{I},\mathcal{C},\mathcal{S}_1), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_2, \mathsf{Error})}{(\mathcal{I},\mathcal{C},\mathcal{S}), (se, sp) :: le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_2, \mathsf{Error})}$$

LIST-EXPR-CONS

$$\frac{s, \mathsf{True}, (\mathcal{I},\mathcal{C},\mathcal{S}), se \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Success}\, (s, b_1)) \qquad (\mathcal{I},\mathcal{C},\mathcal{S}_1), le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_2, \mathsf{Success}\, l_2) \qquad l_3 = split_{sp}(s) +\!\!+ l_2}{(\mathcal{I},\mathcal{C},\mathcal{S}), (se, sp) :: le \Downarrow_s^{\mathsf{L}} (\mathcal{S}_2, \mathsf{Success}\, l_3)}$$

B.3 Evaluation of string expressions, $\Downarrow^{\mathsf{S}}$

STR-LITERAL
$$\frac{res = \mathsf{Success}\,(str, b)}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), str \Downarrow^{\mathsf{S}} (\mathcal{S}, res)}$$

STR-VARIABLE
$$\frac{str = \mathcal{C}.\mathsf{vars}[id]' \qquad res = \mathsf{Success}\,(str, b)}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), id \Downarrow^{\mathsf{S}} (\mathcal{S}, res)}$$

STR-ARG
$$\frac{str = nth\text{-}arg(\mathcal{I}.\mathsf{argument}_0 :: \mathcal{C}.\mathsf{arguments}, n) \qquad res = \mathsf{Success}\,(str, b)}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{arg}\ n \Downarrow^{\mathsf{S}} (\mathcal{S}, res)}$$

STR-SUBSHELL-FAILURE
$$\frac{(\mathcal{I}, \mathcal{C}, \{\mathcal{S}\ \mathtt{with\ stdout}{=}\varepsilon\}), i \Downarrow^{\mathsf{I}}_s (\mathcal{S}_1, \mathcal{C}_1, \mathsf{Failure}) \qquad \mathcal{S}'_1 = \{\mathcal{S}_1\ \mathtt{with\ stdout}{=}\mathcal{S}.\mathsf{stdout}\}}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{embed\ \{}\ i\ \mathtt{\}} \Downarrow^{\mathsf{S}} (\mathcal{S}'_1, \mathsf{Error})}$$

STR-SUBSHELL
$$\frac{\begin{array}{c}\beta_1 \neq \mathsf{Failure} \qquad (\mathcal{I}, \mathcal{C}, \{\mathcal{S}\ \mathtt{with\ stdout}{=}\varepsilon\}), i \Downarrow^{\mathsf{I}}_s (\mathcal{S}_1, \mathcal{C}_1, \beta_1) \\ \mathcal{S}'_1 = \{\mathcal{S}_1\ \mathtt{with\ stdout}{=}\mathcal{S}.\mathsf{stdout}\} \qquad res = \mathsf{Success}\,(\mathcal{S}_1.\mathsf{stdout}, \mathcal{C}_1.\mathsf{result})\end{array}}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{embed\ \{}\ i\ \mathtt{\}} \Downarrow^{\mathsf{S}} (\mathcal{S}'_1, res)}$$

STR-CONCAT-FAILURE1
$$\frac{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), e_1 \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Error})}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), e_1\ e_2 \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Error})}$$

STR-CONCAT-FAILURE2
$$\frac{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), e_1 \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Success}\,(str_1, b_1)) \qquad s, b_1, (\mathcal{I}, \mathcal{C}, \mathcal{S}_1), e_2 \Downarrow^{\mathsf{S}} (\mathcal{S}_2, \mathsf{Error})}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), e_1\ e_2 \Downarrow^{\mathsf{S}} (\mathcal{S}_2, \mathsf{Error})}$$

STR-CONCAT
$$\frac{\begin{array}{c}s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), e_1 \Downarrow^{\mathsf{S}} (\mathcal{S}_1, \mathsf{Success}\,(str_1, b_1)) \\ s, b_1, (\mathcal{I}, \mathcal{C}, \mathcal{S}_1), e_2 \Downarrow^{\mathsf{S}} (\mathcal{S}_2, \mathsf{Success}\,(str_2, b_2)) \qquad res = \mathsf{Success}\,(str_1\hat{}str_2, b_2)\end{array}}{s, b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), e_1\ e_2 \Downarrow^{\mathsf{S}} (\mathcal{S}_2, res)}$$

B.4 Evaluation of for loops, $\Downarrow^{\mathsf{F}}$

FOREACH-DONE
$$\frac{}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{for}\ id\ \mathtt{[]}\ i \Downarrow^{\mathsf{F}}_s (\mathcal{S}, \mathcal{C}, \mathsf{Normal}), b}$$

FOREACH-ABORT
$$\frac{\mathcal{C}' = \{\mathcal{C}\ \mathtt{with\ vars}{=}\mathcal{C}.\mathsf{vars}[id \leftarrow s]'\} \qquad (\mathcal{I}, \mathcal{C}', \mathcal{S}), i \Downarrow^{\mathsf{I}}_s (\mathcal{S}_1, \mathcal{C}_1, \beta_1) \qquad \beta_1 \neq \mathsf{Normal}}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{for}\ id\ s :: ss'\ i \Downarrow^{\mathsf{F}}_s (\mathcal{S}_1, \mathcal{C}_1, \beta_1), \mathcal{C}_1.\mathsf{result}}$$

FOREACH-STEP
$$\frac{\begin{array}{c}\mathcal{C}' = \{\mathcal{C}\ \mathtt{with\ vars}{=}\mathcal{C}.\mathsf{vars}[id \leftarrow s]'\} \\ (\mathcal{I}, \mathcal{C}', \mathcal{S}), i \Downarrow^{\mathsf{I}}_s (\mathcal{S}_1, \mathcal{C}_1, \mathsf{Normal}) \qquad \mathcal{C}_1.\mathsf{result}, (\mathcal{I}, \mathcal{C}_1, \mathcal{S}_1), \mathtt{for}\ id\ ss'\ i \Downarrow^{\mathsf{F}}_s (\mathcal{S}_2, \mathcal{C}_2, \beta_2), b_2\end{array}}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{for}\ id\ s :: ss'\ i \Downarrow^{\mathsf{F}}_s (\mathcal{S}_2, \mathcal{C}_2, \beta_2), b_2}$$

B.5 Evaluation of while loops, $\Downarrow^{\mathsf{W}}$

WHILE-LOOP-LIMIT

$$\frac{0 \leq n \qquad \mathcal{I}.\mathsf{loop\text{-}limit} = n}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{while}\ i_1\ i_2 \Downarrow^{\mathsf{W}}_{s,n} (\mathcal{S}, \mathcal{C}, \mathsf{Failure}), b}$$

WHILE-ABORT-CONDITION

$$\frac{0 \leq n \qquad \mathcal{I}.\mathsf{loop\text{-}limit} \neq n \qquad (\{\mathcal{I}\ \mathtt{with}\ \mathsf{under\text{-}condition}{=}\mathsf{True}\}, \mathcal{C}, \mathcal{S}), i_1 \Downarrow^{\mathsf{I}}_{s} (\mathcal{S}_1, \mathcal{C}_1, \beta_1) \qquad \beta_1 \neq \mathsf{Normal}}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{while}\ i_1\ i_2 \Downarrow^{\mathsf{W}}_{s,n} (\mathcal{S}_1, \mathcal{C}_1, \beta_1), b}$$

WHILE-FALSE

$$\frac{\begin{array}{c} 0 \leq n \\ \mathcal{I}.\mathsf{loop\text{-}limit} \neq n \qquad \mathcal{C}_1.\mathsf{result} = \mathsf{False} \qquad (\{\mathcal{I}\ \mathtt{with}\ \mathsf{under\text{-}condition}{=}\mathsf{True}\}, \mathcal{C}, \mathcal{S}), i_1 \Downarrow^{\mathsf{I}}_{s} (\mathcal{S}_1, \mathcal{C}_1, \mathsf{Normal}) \end{array}}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{while}\ i_1\ i_2 \Downarrow^{\mathsf{W}}_{s,n} (\mathcal{S}_1, \mathcal{C}_1, \mathsf{Normal}), b}$$

WHILE-ABORT-BODY

$$\frac{\begin{array}{c} 0 \leq n \qquad \mathcal{I}.\mathsf{loop\text{-}limit} \neq n \qquad (\{\mathcal{I}\ \mathtt{with}\ \mathsf{under\text{-}condition}{=}\mathsf{True}\}, \mathcal{C}, \mathcal{S}), i_1 \Downarrow^{\mathsf{I}}_{s} (\mathcal{S}_1, \mathcal{C}_1, \mathsf{Normal}) \\ \mathcal{C}_1.\mathsf{result} = \mathsf{True} \qquad (\mathcal{I}, \mathcal{C}_1, \mathcal{S}_1), i_2 \Downarrow^{\mathsf{I}}_{s} (\mathcal{S}_2, \mathcal{C}_2, \beta_2) \qquad \beta_2 \neq \mathsf{Normal} \end{array}}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{while}\ i_1\ i_2 \Downarrow^{\mathsf{W}}_{s,n} (\mathcal{S}_2, \mathcal{C}_2, \beta_2), b}$$

WHILE-LOOP

$$\frac{\begin{array}{c} 0 \leq n \\ \mathcal{I}.\mathsf{loop\text{-}limit} \neq n \qquad (\{\mathcal{I}\ \mathtt{with}\ \mathsf{under\text{-}condition}{=}\mathsf{True}\}, \mathcal{C}, \mathcal{S}), i_1 \Downarrow^{\mathsf{I}}_{s} (\mathcal{S}_1, \mathcal{C}_1, \mathsf{Normal}) \qquad \mathcal{C}_1.\mathsf{result} = \mathsf{True} \\ (\mathcal{I}, \mathcal{C}_1, \mathcal{S}_1), i_2 \Downarrow^{\mathsf{I}}_{s} (\mathcal{S}_2, \mathcal{C}_2, \mathsf{Normal}) \qquad \mathcal{C}_2.\mathsf{result}, (\mathcal{I}, \mathcal{C}_2, \mathcal{S}_2), \mathtt{while}\ i_1\ i_2 \Downarrow^{\mathsf{W}}_{s,n+1} (\mathcal{S}_3, \mathcal{C}_3, \beta_3), b_3 \end{array}}{b, (\mathcal{I}, \mathcal{C}, \mathcal{S}), \mathtt{while}\ i_1\ i_2 \Downarrow^{\mathsf{W}}_{s,n} (\mathcal{S}_3, \mathcal{C}_3, \beta_3), b_3}$$

B.6 Evaluation of function definitions, $\Downarrow^{\mathsf{D}}$

FUNCTION-DEFINITIONS-DONE

$$\frac{}{\mathit{funs}, [\,] \Downarrow^{\mathsf{D}} \mathit{funs}}$$

FUNCTION-DEFINITION

$$\frac{e[\mathit{id} \leftarrow \mathsf{Some}\ i], \mathit{defs} \Downarrow^{\mathsf{D}} e'}{e, (\mathit{id}, i) :: \mathit{defs} \Downarrow^{\mathsf{D}} e'}$$

B.7 Evaluation of programs, $\Downarrow^{\mathsf{P}}$

PROGRAM

$$\frac{\mathcal{C}.\mathsf{funs}, p.\mathsf{funs} \Downarrow^{\mathsf{D}} \mathit{funs} \qquad (\mathcal{I}, \{\mathcal{C}\ \mathtt{with}\ \mathsf{funs}{=}\mathit{funs}\}, \mathcal{S}), p.\mathsf{instruction} \Downarrow^{\mathsf{I}}_{0} (\mathcal{S}', \mathcal{C}', \beta) \qquad \beta \neq \mathsf{Failure}}{(\mathcal{I}, \mathcal{C}, \mathcal{S}), p \Downarrow^{\mathsf{P}} (\mathcal{S}', \mathsf{Success}\, \mathcal{C}'.\mathsf{result})}$$

PROGRAM-FAILURE

$$\frac{\mathcal{C}.\mathsf{funs}, p.\mathsf{funs} \Downarrow^{\mathsf{D}} \mathit{funs} \qquad (\mathcal{I}, \{\mathcal{C}\ \mathtt{with}\ \mathsf{funs}{=}\mathit{funs}\}, \mathcal{S}), p.\mathsf{instruction} \Downarrow^{\mathsf{I}}_{0} (\mathcal{S}', \mathcal{C}', \mathsf{Failure})}{(\mathcal{I}, \mathcal{C}, \mathcal{S}), p \Downarrow^{\mathsf{P}} (\mathcal{S}', \mathsf{Error})}$$