Calculating Certified Compilers for Non-Deterministic Languages*

Patrick Bahr

Department of Computer Science, University of Copenhagen, Denmark
paba@di.ku.dk

Abstract. Reasoning about programming languages with non-deterministic semantics entails many difficulties. For instance, to prove correctness of a compiler for such a language, one typically has to split the correctness property into a soundness and a completeness part, and then prove these two parts separately. In this paper, we present a set of proof rules to prove compiler correctness by a *single* proof in calculational style. The key observation that led to our proof rules is the fact that the soundness and completeness proof follow a similar pattern with only small differences. We condensed these differences into a single side condition for one of our proof rules. This side condition, however, is easily discharged automatically by a very simple form of proof search. We implemented this calculation framework in the Coq proof assistant. Apart from verifying a given compiler, our proof technique can also be used to formally derive – from the semantics of the source language – a compiler that is *correct by construction*. For such a derivation to succeed it is crucial that the underlying correctness argument proceeds as a single calculation, as opposed to separate calculations of the two directions of the correctness property. We demonstrate our technique by deriving a compiler for a simple language with interrupts.

1 Introduction

Formally verifying the correctness of compilers is a difficult and expensive endeavour [9]. However, the need for formally verified compilers is a corollary of the need for formal verification of critical pieces of software; for what good is your formally verified program if it is garbled by a defective compiler.

These challenges notwithstanding, we pursue an even more ambitious goal than post hoc verification: Not only do we wish to formally verify the correctness of a given compiler implementation. Beyond that we aim to derive a compiler implementation that is *correct by construction* [3]. That is, given the semantics of the source language and a high-level specification of the compiler, we wish to systematically derive a compiler implementation that satisfies the specification.

* This work was supported by the Danish Council for Independent Research, Grant 12-132365, "Efficient Programming Language Development and Evolution through Modularity".

This idea has been explored in the literature for quite some time; e.g. by Wand [14], Meijer [10], Ager et al. [1]. Recently, Bahr and Hutton [4] have shown an approach that derives a compiler *directly* from the statement of the compiler correctness property by performing a calculation proof. The parts of the setup that are not given up front, i.e. the compiler itself and the target language, reveal themselves during the calculation proof. Taken together, the final result of the calculation process is a compiler implementation and a (machine checked) formal proof of its correctness. In addition, also the virtual machine, which defines the semantics of the target language, falls out of the calculation proof.

A crucial ingredient of the calculation technique of Bahr and Hutton [4] is that the correctness argument proceeds "in one go", such that at any point during the proof we have a complete view of the computational context and its invariants. So far it was not known how this technique can be applied to languages with an inherent non-deterministic semantics. The problem that arises for these languages is that the compiler correctness property is typically split into a *soundness* and a *completeness* part, which are proved independently; for example, see Hutton and Wright [7]. Another technical challenge arises from the fact that the equational reasoning style used by Bahr and Hutton [4] is incompatible with non-deterministic semantics, which are typically given in the form of a relation (big-step/small-step operational semantics).

In this paper we improve and generalise the technique of Bahr and Hutton [4] for calculating compilers such that we can calculate compilers for non-deterministic languages. While our approach works also for proofs by hand, it works particularly well in a proof assistant such as Coq. In addition to calculating compilers that are correct by construction, our approach is also applicable to post hoc compiler verification.

The key contributions of this paper are the following:

- We devise a set of general proof rules for compiler correctness proofs for non-deterministic languages. These proof rules have been verified in Coq and are the basis for a calculation framework that we developed in Coq.
- We demonstrate the power of our calculation framework on a number of examples both in this paper and in the accompanying Coq development. In particular, we show its effectiveness for both post hoc verification of compilers as well as derivation of correct-by-construction compilers in the style of Bahr and Hutton [4].
- Apart from the ability to deal with non-determinism, the distinguishing feature of our approach is that we are able to *directly* derive a small-step operational semantics of the virtual machine (instead of a tail-recursive function).

To illustrate the problem we are trying to solve we begin with a simple non-deterministic toy language along with a compiler for it, which we define in section 2. In section 3, we present our calculation framework and apply it to the toy language to prove its compiler correct. In section 4, we illustrate how our framework can also be used to derive a correct-by-construction compiler from the specification of its correctness property. We then use this knowledge in section 5 in order to derive a compiler for a simple language with interrupts

$$\frac{}{\mathsf{Val\ n} \Downarrow \mathsf{n}}\ \textsc{Val} \qquad \frac{\mathsf{x} \Downarrow \mathsf{n} \qquad \mathsf{y} \Downarrow \mathsf{m}}{\mathsf{Add\ x\ y} \Downarrow (\mathsf{n}+\mathsf{m})}\ \textsc{Add} \qquad \frac{\mathsf{x} \Downarrow \mathsf{n} \qquad 0 \leqslant \mathsf{m} \leqslant |\mathsf{n}|}{\mathsf{Rnd\ x} \Downarrow \mathsf{m}}\ \textsc{Rnd}$$

Fig. 1: Semantics of the language.

(taken from Hutton and Wright [7]). Finally, in section 7 we discuss limitations of our approach and outline further work to address these limitations.

This paper uses Coq as a meta language and as a tool for proof automation. However, familiarity with Coq is not required to follow this paper. Section 6 covers some technical details of the use of proof automation in Coq, which does require some familiarity with Coq. Nonetheless, the core idea and the main contributions of this paper are independent of these technical details.

All calculation proofs in this paper can be found in the accompanying Coq source code, which is available from the author's web site[1]. The Coq development also includes calculations for languages that extend the interrupt language of Hutton and Wright [7] with state. Moreover, the source code contains proofs for all theorems presented in this paper including the correctness of the proof rules of our calculation framework.

2 A Simple Non-Deterministic Language

To illustrate the problem we aim to solve, we begin with a very simple toy language. The syntax is given by the following inductive type definition:

```
Inductive Expr : Set := Val (n : ℤ) | Add (e1 e2 : Expr) | Rnd (e : Expr).
```

Apart from integer literals (represented using the type $\mathbb{Z}$) and an addition operator Add, the language has a construct Rnd to generate a random number. The intended semantics of an expression Rnd e is that it evaluates e to a number n and then returns a number m with $0 \leq \mathsf{m} \leq |\mathsf{n}|$, where $|\mathsf{n}|$ denotes the absolute value of n. For instance, the expression Add (Rnd (Val 5)) (Val 42) generates a random number between 0 and 5 and adds 42 to it.

We formally define the semantics of this simple language by a big-step operational semantics [8], writing e ⇓ n to mean that the expression e evaluates to the number n. The binary relation ⇓ is given by the following inductive type definition:

```
Inductive eval : Expr → ℤ → Prop :=
| evalVal n         : Val n ⇓ n
| evalAdd x y m n : x ⇓ m → y ⇓ n            → Add x y ⇓ (m + n)
| evalRnd x n m   : x ⇓ n → 0 ⩽ m ⩽ |n| → Rnd x ⇓ m
where "x ⇓ y" := (eval x y).
```

[1] Or directly from `https://github.com/pa-ba/calc-comp-rel`.

A more readable form of this definition is given in Figure 1. From now on, we shall give inductive semantic definitions in this form, with the tacit understanding that it can be easily turned into an inductive type family definition in Coq.

Our compiler for the Expr language will target a simple machine that can store integer values on a stack and that executes the instruction set given by the following type definition:

```
Inductive Instr : Set := PUSH (n : ℤ) | ADD | RND .
```

The intuitive semantics of the three instructions above is that

- PUSH n pushes the number n onto the stack,
- ADD replaces the two topmost numbers on the stack with their sum, and
- RND replaces the topmost number n on the stack with a number m such that $0 \leqslant m \leqslant |n|$.

A program for this target machine is simply a sequence of these instructions, which is captured by the type Code defined below:

```
Definition Code := list Instr .
```

We will use the notation [] for the empty list, and x :: xs for the list with a head element x and a tail list xs.

Before we give the formal semantics of the target language, we present the compiler that translates expressions of the source language Expr into the target language Code. To this end, the compilation function takes an additional argument of type Code that represents a continuation (cf. Hutton [6, Chapter 13]):

```
Fixpoint comp' (e : Expr) (c : Code) : Code :=
  match e with
    | Val n   ⇒ PUSH n :: c
    | Add x y ⇒ comp' x (comp' y (ADD :: c))
    | Rnd x   ⇒ comp' x (RND :: c)
  end .
```

The final compiler is obtained by supplying the empty list of instructions [] as the initial value of the continuation argument:

```
Definition comp (e : Expr) : Code := comp' e [].
```

For instance, the expression Add (Rnd (Val 5)) (Val 42) compiles to the code [PUSH 5; RND; PUSH 42; ADD].

Finally, we give the semantics of the target language Code in the form of a *virtual machine* [2, 1]: a small-step operational semantics, given by the reflexive, transitive closure $\stackrel{*}{\Longrightarrow}$ of a binary relation $\Longrightarrow$ on the type of machine configurations Conf, which is defined below:

```
Definition Stack : Set := list ℤ.
Inductive Conf : Set  := conf (c : Code) (s : Stack).
```

$$\langle \mathsf{PUSH}\ n :: c\ ,\ s\rangle \Longrightarrow \langle c\ ,\ n :: s\rangle \qquad \text{(VM-Push)}$$

$$\langle \mathsf{ADD} :: c,\ m :: n :: s\rangle \Longrightarrow \langle c,\ (n + m) :: s\rangle \qquad \text{(VM-Add)}$$

$$\langle \mathsf{RND} :: c,\ n :: s\rangle \Longrightarrow \langle c,\ m :: s\rangle \quad \text{if } 0 \leqslant m \leqslant |n| \qquad \text{(VM-Rnd)}$$

Fig. 2: Definition of the virtual machine.

Notation "$\langle\ c\ ,\ s\ \rangle$" := (conf c s) .

A configuration describes the state of the virtual machine; it is a pair $\langle c,\ s\rangle$ consisting of a code c and a stack s. The binary relation $\Longrightarrow$ describes a single computation step in the virtual machine: if $C \Longrightarrow C'$, then the virtual machine transitions from configuration C to configuration C' in one step. The inductive definition of the relation $\Longrightarrow$ is presented in Figure 2.

For example, the code $[\mathsf{PUSH}\ 5; \mathsf{RND}; \mathsf{PUSH}\ 42; \mathsf{ADD}]$ is executed by the virtual machine starting with the empty stack as follows:

$$\begin{aligned}
&\langle [\mathsf{PUSH}\ 5; \mathsf{RND}; \mathsf{PUSH}\ 42; \mathsf{ADD}],\ []\rangle \\
&\quad \Longrightarrow \langle [\mathsf{RND}; \mathsf{PUSH}\ 42; \mathsf{ADD}],\ [5]\rangle && \text{(by VM-Push)} \\
&\quad \Longrightarrow \langle [\mathsf{PUSH}\ 42; \mathsf{ADD}],\ [3]\rangle && \text{(by VM-Rnd)} \\
&\quad \Longrightarrow \langle [\mathsf{ADD}],\ [42; 3]\rangle && \text{(by VM-Push)} \\
&\quad \Longrightarrow \langle [],\ [45]\rangle && \text{(by VM-Add)}
\end{aligned}$$

That is, $\langle [\mathsf{PUSH}\ 5; \mathsf{RND}; \mathsf{PUSH}\ 42; \mathsf{ADD}],\ []\rangle \overset{*}{\Longrightarrow} \langle [],\ [45]\rangle$. However, this is not the only possible execution. The rule for RND allows for more than one successor configuration after $\langle [\mathsf{RND}; \mathsf{PUSH}\ 42; \mathsf{ADD}],\ [5]\rangle$.

3 Correctness Property

Intuitively, the correctness property of the compiler comp states that for each expression e, running the code comp e produced by the compiler on the virtual machine yields the same result as evaluating e according to $\Downarrow$. Taking account of the non-determinism, the correctness property reads as follows: an expression e *may* evaluate to n iff running comp e on the virtual machine *may* produce the result n. The "only if" direction of this equivalence expresses completeness, whereas the "if" direction expresses soundness.

More formally, for the completeness property we want that whenever $e \Downarrow n$, then the virtual machine computes the result n as well, i.e. $\langle \mathsf{comp}\ e,\ []\rangle \overset{*}{\Longrightarrow} \langle [],\ [n]\rangle$. In order to prove this property by induction, we have to generalise it to arbitrary $c :: \mathsf{Code}$ and $s :: \mathsf{Stack}$ as follows:

$$\forall\, e\ n\ c\ s,\quad e \Downarrow n \rightarrow \langle \mathsf{comp'}\ e\ c,\ s\rangle \overset{*}{\Longrightarrow} \langle c,\ n :: s\rangle \qquad \text{(completeness)}$$

Conversely, for the soundness property we want that whenever we have $\langle \mathsf{comp}\ \mathsf{e},\ [] \rangle \stackrel{*}{\Longrightarrow} \mathsf{C}$, then we find some $\mathsf{C} \stackrel{*}{\Longrightarrow} \langle [], [\mathsf{n}] \rangle$ with $\mathsf{e} \Downarrow \mathsf{n}$. In other words, any run of the virtual machine can be extended such that it ends in a result value n such that $\mathsf{e} \Downarrow \mathsf{n}$. We have to generalise this property as well, in order to be able to prove it by induction. However, simply generalising it to arbitrary $\mathsf{c} :: \mathsf{Code}$ and $\mathsf{s} :: \mathsf{Stack}$ as follows is not enough:

$$\forall\, \mathsf{e}\ \mathsf{c}\ \mathsf{s}, \quad \langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c},\ \mathsf{s} \rangle \stackrel{*}{\Longrightarrow} \mathsf{C} \rightarrow \exists\, \mathsf{n}, \mathsf{C} \stackrel{*}{\Longrightarrow} \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle \wedge \mathsf{e} \Downarrow \mathsf{n}$$

Unfortunately, the above straightforward generalisation is not true. The problem is that the given run $\langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c},\ \mathsf{s} \rangle \stackrel{*}{\Longrightarrow} \mathsf{C}$ may have gone already beyond the configuration $\langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle$. That is, we have that

$$\langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c},\ \mathsf{s} \rangle \stackrel{*}{\Longrightarrow} \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle \stackrel{*}{\Longrightarrow} \mathsf{C}$$

Thus, we can in general not expect that $\mathsf{C} \stackrel{*}{\Longrightarrow} \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle$.

To take this situation into account, we follow an approach similar to Hutton and Wright [7]: we formulate the soundness property in terms of the notion that a machine configuration $\mathsf{C} : \mathsf{Conf}$ is *barred* by a set of configurations $\mathsf{S} : \mathsf{ConfSet}$, denoted $\mathsf{C} \lhd \mathsf{S}$, cf. Troelstra and van Dalen [13]. For the moment, we shall remain informal about what the type $\mathsf{ConfSet}$ of sets of configurations is and appeal to the intuitive notion of sets. Intuitively, $\mathsf{C} \lhd \mathsf{S}$ means that any sequence of $\Longrightarrow$-steps starting from C can be extended such that it *passes through* a configuration that is in S, i.e. for any sequence $\mathsf{C}_0 \Longrightarrow \mathsf{C}_1 \Longrightarrow \ldots \Longrightarrow \mathsf{C}_\mathsf{n}$ with $\mathsf{C} = \mathsf{C}_0$, we find a sequence $\mathsf{C}_\mathsf{n} \Longrightarrow \ldots \Longrightarrow \mathsf{C}_\mathsf{m}$ such that $\mathsf{C}_\mathsf{i} \in \mathsf{S}$ for some $0 \leqslant \mathsf{i} \leqslant \mathsf{m}$. Formally, we define $\lhd$ by the following inductive rules:

$$\frac{\mathsf{C} \in \mathsf{S}}{\mathsf{C} \lhd \mathsf{S}}\ \textsc{Here-}\lhd \qquad \frac{\forall\, \mathsf{D}, \mathsf{C} \Longrightarrow \mathsf{D} \rightarrow \mathsf{D} \lhd \mathsf{S} \qquad \exists\, \mathsf{D}, \mathsf{C} \Longrightarrow \mathsf{D}}{\mathsf{C} \lhd \mathsf{S}}\ \textsc{Step-}\lhd$$

We can then use the relation $\lhd$ to capture the soundness property of the compiler by the following statement:

$$\forall\, \mathsf{e}\ \mathsf{c}\ \mathsf{s}, \quad \langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c},\ \mathsf{s} \rangle \lhd \{\mathsf{n}, \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle \mid \mathsf{e} \Downarrow \mathsf{n}\} \qquad (\textsc{soundness})$$

Here we use the set comprehension notation $\{\mathsf{n}, \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle \mid \mathsf{e} \Downarrow \mathsf{n}\}$, which explicitly mentions the existentially quantified variable n, whose scope ranges over both the expression $\langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle$ and the predicate $\mathsf{e} \Downarrow \mathsf{n}$. That is, $\{\mathsf{n}, \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle \mid \mathsf{e} \Downarrow \mathsf{n}\}$ denotes the set of configurations of the form $\langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle$ such that $\mathsf{e} \Downarrow \mathsf{n}$ holds. In an informal set comprehension notation, one would typically write $\{\langle \mathsf{c},\ \mathsf{n} :: \mathsf{s} \rangle \mid \mathsf{e} \Downarrow \mathsf{n}\}$ instead.

The above property (SOUNDNESS) does indeed imply the desired soundness property, i.e. that $\langle \mathsf{comp}\ \mathsf{e},\ [] \rangle \stackrel{*}{\Longrightarrow} \mathsf{C}$ implies both $\mathsf{C} \stackrel{*}{\Longrightarrow} \langle [], [\mathsf{n}] \rangle$ and $\mathsf{e} \Downarrow \mathsf{n}$ for some n. This implication is a consequence of the following *general* property of the relation $\lhd$:

Proposition 1. *Let $S : \mathit{ConfSet}$ be such that all $C \in S$ are in normal form, i.e. there is no D with $C \Longrightarrow D$. If $C_1 \lhd S$ and $C_1 \stackrel{*}{\Longrightarrow} C_2$, then $C_2 \stackrel{*}{\Longrightarrow} C_3$ and $C_3 \in S$ for some C_3.*

In other words, any sequence of $\Longrightarrow$-steps starting from a configuration that is barred by a set of normal forms S can be extended such that it ends in a configuration in S. This property is general in the sense that it is independent of the definition of Conf and $\Longrightarrow$.

For the above proposition to be true, it is important that we have $\exists\ \mathsf{D},\ \mathsf{C} \Longrightarrow \mathsf{D}$ as a second antecedent in the Step-$\lhd$ rule. Without it, we would have $\mathsf{C} \lhd \mathsf{S}$ for any normal form C, i.e. any C for which there is no D with $\mathsf{C} \Longrightarrow \mathsf{D}$. In other words, we would be able to prove "soundness" even though some computations in the virtual machine might get stuck.[2] We will review an example that illustrates this in section 7.

Our goal is to prove soundness and completeness for the compiler in a calculational style. Such a proof was given by Hutton and Wright [7] (for a much more interesting language, which we will consider in section 5). They combine the two relations $\overset{*}{\Longrightarrow}$ and $\lhd$ into a single relation $\lhd\!\!\lhd$ by defining $\mathsf{C} \lhd\!\!\lhd \mathsf{S}$ iff $\mathsf{C} \overset{*}{\Longrightarrow} \mathsf{S}$ and $\mathsf{C} \lhd \mathsf{S}$, where $\overset{*}{\Longrightarrow}$ is lifted to sets by defining $\mathsf{C} \overset{*}{\Longrightarrow} \mathsf{S}$ iff $\mathsf{C} \overset{*}{\Longrightarrow} \mathsf{D}$ for all $\mathsf{D} \in \mathsf{S}$. Nonetheless, this calculational proof of Hutton and Wright [7] considers soundness and completeness separately. In order to combine the two proofs into a single calculational proof, we have to overcome two obstacles:

Firstly, the completeness proof proceeds by induction on (the proof of) $\mathsf{e} \Downarrow \mathsf{n}$, whereas the soundness proof proceeds by induction on e. This technical hurdle is overcome by transforming the completeness proof into an induction on e. While this approach may not be possible for every language, it does not restrict the applicability of the proof method as a whole, since the soundness proof is already an induction on e (cf. discussion in section 7).

Secondly, the two proofs utilise proof principles that are not valid for both $\overset{*}{\Longrightarrow}$ and $\lhd$. For example, the completeness proof makes use of the fact that $\mathsf{C} \Longrightarrow \mathsf{S}$ implies $\mathsf{C} \overset{*}{\Longrightarrow} \mathsf{S}$, whereas $\mathsf{C} \Longrightarrow \mathsf{S}$ does not necessarily imply that $\mathsf{C} \lhd \mathsf{S}$. Conversely, the soundness proof makes use of the fact that if $\mathsf{S} \subseteq \mathsf{T}$, then $\mathsf{C} \lhd \mathsf{S}$ implies $\mathsf{C} \lhd \mathsf{T}$, whereas $\mathsf{C} \overset{*}{\Longrightarrow} \mathsf{S}$ does not necessarily imply $\mathsf{C} \overset{*}{\Longrightarrow} \mathsf{T}$.

Overcoming the second hurdle is more difficult. But the underlying idea to solve this problem is quite simple. We take the completeness proof as our basis. The only thing that is missing in order to turn this proof into a soundness proof as well is the fact that we do not have that $\mathsf{C} \Longrightarrow \mathsf{S}$ implies $\mathsf{C} \lhd \mathsf{S}$ in general. However, the additional proof obligation necessary to conclude $\mathsf{C} \lhd \mathsf{S}$ can be discharged automatically by proof search on the $\Longrightarrow$ relation.

The automation of proofs of $\mathsf{C} \lhd \mathsf{S}$ is particularly important for our goal of deriving a compiler by calculation, which we will discuss in section 5. In that setting, both compiler and virtual machine are not fully defined in the beginning of the calculation. As the calculation progresses we flesh out the definition of the compiler and the virtual machine. In particular, we add new rules for $\Longrightarrow$. As a consequence, a statement of the form $\mathsf{C} \lhd \mathsf{S}$, might not hold anymore after we have extended the virtual machine relation $\Longrightarrow$. The proof search will extend

[2] The side condition $\exists\ \mathsf{D},\ \mathsf{C} \Longrightarrow \mathsf{D}$ is absent in the definition of $\lhd$ by Hutton and Wright [7]. However, their proofs need little change to account for this stronger version of $\lhd$, which then yields proper soundness and completeness for their compiler.

the proof to account for the modified definition of $\Longrightarrow$. If this fails, we are immediately informed that the added rule for $\Longrightarrow$ breaks the existing proof.

3.1 Proof Principles

In this section we will present the proof principles for constructing soundness and completeness proofs. These proof principles are general lemmas about the two relations $\stackrel{*}{\Longrightarrow}$ and $\triangleleft$, i.e. they are independent of the definition of Conf and $\Longrightarrow$. Our goal is to have a combined relation, similar to $\lessdot$ of Hutton and Wright [7], that will allow us to prove soundness and completeness in one go. To this end we will lift both relations to sets of configurations; recall that $\stackrel{*}{\Longrightarrow}$ is of type $\mathsf{Conf} \to \mathsf{Conf} \to \mathsf{Prop}$, whereas $\triangleleft$ is of type $\mathsf{Conf} \to \mathsf{ConfSet} \to \mathsf{Prop}$.

As a first step we have to define, what a set of configurations is. The corresponding type $\mathsf{ConfSet}$, has to provide the *element* relation $\in$ of type $\mathsf{Conf} \to \mathsf{ConfSet} \to \mathsf{Prop}$. Moreover, we need to be able to construct the empty set $\emptyset$, form the union $\mathsf{S} \cup \mathsf{T}$ of two sets, and define sets using set comprehension notation like the example $\{\mathsf{n}, \langle \mathsf{c}, \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n}\}$ above. In general, set comprehensions have the form $\{\overline{x}, \mathsf{C} \mid \mathsf{P}\}$, where $\overline{x}$ is a list of variables that are existentially quantified in C and P, which in turn are of type Conf and Prop, respectively. We shall return to the technical issue of representing this notation in Coq – and more importantly how to reason over it – in section 6.

The inductive rules below lift $\stackrel{*}{\Longrightarrow}$ and $\triangleleft$ to the relations $\Longrightarrow\!\!\!\!\!\Rightarrow$ and $\lessdot$, respectively, both of type $\mathsf{ConfSet} \to \mathsf{ConfSet} \to \mathsf{Prop}$:

$$\frac{\forall\, \mathsf{D}, \mathsf{D} \in \mathsf{T} \to (\exists\, \mathsf{C}, \mathsf{C} \in \mathsf{S} \wedge \mathsf{C} \stackrel{*}{\Longrightarrow} \mathsf{D})}{\mathsf{S} \Longrightarrow\!\!\!\!\!\Rightarrow \mathsf{T}} \qquad \frac{\forall\, \mathsf{C}, \mathsf{C} \in \mathsf{S} \to \mathsf{C} \triangleleft \mathsf{T}}{\mathsf{S} \lessdot \mathsf{T}}$$

That is, $\mathsf{S} \Longrightarrow\!\!\!\!\!\Rightarrow \mathsf{T}$ iff all configurations in T are reachable from some configuration in S; and $\mathsf{S} \lessdot \mathsf{T}$ iff all configurations in S are barred by T.

Using these two relations, we can reformulate the soundness and completeness properties as follows:

$$\forall\, \mathsf{e}\ \mathsf{c}\ \mathsf{s}, \quad \{\langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c}, \mathsf{s}\rangle\} \Longrightarrow\!\!\!\!\!\Rightarrow \{\mathsf{n}, \langle \mathsf{c}, \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n}\} \qquad \text{(COMPLETENESS)}$$

$$\forall\, \mathsf{e}\ \mathsf{c}\ \mathsf{s}, \quad \{\langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c}, \mathsf{s}\rangle\} \lessdot \{\mathsf{n}, \langle \mathsf{c}, \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n}\} \qquad \text{(SOUNDNESS)}$$

We then combine the two relations $\Longrightarrow\!\!\!\!\!\Rightarrow$ and $\lessdot$ into the relation $\Rightarrow\!\!\!\!\!\triangleright$ by forming their intersection, which we express by the following inductive rule:

$$\frac{\mathsf{S} \Longrightarrow\!\!\!\!\!\Rightarrow \mathsf{T} \qquad \mathsf{S} \lessdot \mathsf{T}}{\mathsf{S} \Rightarrow\!\!\!\!\!\triangleright \mathsf{T}}$$

This combined relation allows us to succinctly formulate the correctness property of the compiler:

$$\forall\, \mathsf{e}\ \mathsf{c}\ \mathsf{s}, \quad \{\langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c}, \mathsf{s}\rangle\} \Rightarrow\!\!\!\!\!\triangleright \{\mathsf{n}, \langle \mathsf{c}, \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n}\} \qquad \text{(CORRECTNESS)}$$

$$\frac{}{\mathsf{S} \Rrightarrow \mathsf{S}}\ \textsc{Refl} \qquad \frac{\mathsf{S} \Rrightarrow \mathsf{T} \quad \mathsf{T} \Rrightarrow \mathsf{U}}{\mathsf{S} \Rrightarrow \mathsf{U}}\ \textsc{Trans} \qquad \frac{\mathsf{S} \equiv \mathsf{T}}{\mathsf{S} \Rrightarrow \mathsf{T}}\ \textsc{Iff}$$

$$\frac{\mathsf{S} \Rrightarrow \mathsf{T} \quad \mathsf{S'} \Rrightarrow \mathsf{T'}}{\mathsf{S} \cup \mathsf{S'} \Rrightarrow \mathsf{T} \cup \mathsf{T'}}\ \textsc{Union}$$

$$\frac{\forall\, \bar{\mathsf{x}},\ \mathsf{P} \to \mathsf{C} \Longrightarrow \mathsf{D} \qquad \forall\, \bar{\mathsf{x}},\ \mathsf{P} \to \mathsf{C} \lhd \{\bar{\mathsf{x}},\ \mathsf{D} \mid \mathsf{P}\} \cup \mathsf{T}}{\{\bar{\mathsf{x}},\ \mathsf{C} \mid \mathsf{P}\} \cup \mathsf{T} \Rrightarrow \{\bar{\mathsf{x}},\ \mathsf{D} \mid \mathsf{P}\} \cup \mathsf{T}}\ \textsc{Step}$$

Fig. 3: Proof principles.

Figure 3 lists the proof rules that we can derive from the definition of $\Rrightarrow$. Most importantly we have reflexivity, transitivity and closure under union, which will allow us do calculational proofs. In addition, $\Rrightarrow$ is closed under extensional set equality $\equiv$, which is defined as follows:

$$\frac{\forall\, \mathsf{C},\ \mathsf{C} \in \mathsf{S} \leftrightarrow \mathsf{C} \in \mathsf{T}}{\mathsf{S} \equiv \mathsf{T}}$$

Lastly, the Step proof rule in Figure 3 is the only *non-structural* rule. It allows us to "advance" one step according to the $\Longrightarrow$ relation. In order to avoid syntactic clutter, the rule is stated somewhat informally: it is implicitly assumed that P, C and $\mathsf{C'}$ are *expressions* that may contain free variables from $\bar{\mathsf{x}}$. Formalising and proving this rule in Coq requires some technical effort. But we defer discussion of this aspect until section 6.

The Step rule should feel intuitively true: in order to derive

$$\{\bar{\mathsf{x}},\ \mathsf{C} \mid \mathsf{P}\} \cup \mathsf{T} \Rrightarrow \{\bar{\mathsf{x}},\ \mathsf{D} \mid \mathsf{P}\} \cup \mathsf{T},$$

we have to show a step $\mathsf{C} \Longrightarrow \mathsf{D}$ and that $\mathsf{C} \lhd \{\bar{\mathsf{x}},\ \mathsf{D} \mid \mathsf{P}\} \cup \mathsf{T}$ – both in a context of free variables $\bar{\mathsf{x}}$ and assuming that P is true. In practice, the latter proof obligation can be discharged automatically by a simple proof search: check whether C is in the set $\{\bar{\mathsf{x}},\ \mathsf{D} \mid \mathsf{P}\} \cup \mathsf{T}$; if not, then recursively check whether $\mathsf{C'} \lhd \{\bar{\mathsf{x}},\ \mathsf{D} \mid \mathsf{P}\} \cup \mathsf{T}$ for every $\mathsf{C'}$ with $\mathsf{C} \Longrightarrow \mathsf{C'}$.

3.2 Correctness Proof

To illustrate the proof rules we shall prove correctness of the compiler from section 2. Recall that the correctness property of the compiler is formulated using $\Rrightarrow$ as follows:

$$\forall\, \mathsf{e}\ \mathsf{c}\ \mathsf{s},\quad \{\langle \mathsf{comp'}\ \mathsf{e}\ \mathsf{c}, \mathsf{s}\rangle\} \Rrightarrow \{\mathsf{n}, \langle \mathsf{c}, \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n}\}$$

Unfortunately, this correctness property is not suitable for an induction proof. The induction hypothesis is not general enough, since the left-hand side of the

$\Rrightarrow$ is always a singleton set. To avoid this issue we generalise the correctness property as follows:

$$\forall\, e\ c\ P,\quad \{s, \langle \mathsf{comp'}\ e\ c, s\rangle \mid P\ s\} \Rrightarrow \{s\ n, \langle c, n :: s\rangle \mid e \Downarrow n \wedge P\ s\}$$

Instead of quantifying over stacks $s : \mathsf{Stack}$, we quantify over predicates on stacks $P : \mathsf{Stack} \to \mathsf{Prop}$. We can then prove the above property by induction on $e : \mathsf{Expr}$. The calculations are given in Figure 4. Note that we calculate "backwards", i.e. from the right-hand side to the left-hand side. This approach has the benefit that we can let the semantics $e \Downarrow n$ guide the calculation. Later in section 5, performing the calculation in this direction becomes crucial: it allows us to start the proof without having defined the compiler nor the virtual machine. Instead the definitions of the compiler and the virtual machine fall out of the calculation process itself.

Before we go into the details of the calculation, we review how it is built up using the proof rules of Figure 3. Each step of the calculation corresponds to a relation $X \lhd\!= Y$, together with its justification. Instead of $\lhd\!=$, we use the symbol $\equiv$ or $\Longleftarrow$, if the relation is justified by proof rule Iff or Step, respectively. For instance, in the calculation for $\mathsf{Val}\ n$, we first observe that we have the set equivalence

$$\{s\ n', \langle c, n' :: s\rangle \mid \mathsf{Val}\ n \Downarrow n' \wedge P\ s\} \equiv \{s, \langle c, n :: s\rangle \mid P\ s\}$$

This equivalence is justified by the rule Val of the semantics (cf. Figure 1), according to which $\mathsf{Val}\ n \Downarrow n'$ is equivalent to the equation $n = n'$. Applying rule Iff, then yields that

$$\{s\ n', \langle c, n' :: s\rangle \mid \mathsf{Val}\ n \Downarrow n' \wedge P\ s\} \lhd\!= \{s, \langle c, n :: s\rangle \mid P\ s\}$$

Similarly, the second step of the calculation indicates that

$$\langle c, n :: s\rangle \Longleftarrow \langle \mathsf{PUSH}\ n :: c, s\rangle$$

due to the rule VM-Push of the virtual machine (cf. Figure 2). Using proof rule Step we then obtain the desired relation

$$\{s, \langle c, n :: s\rangle \mid P\ s\} \lhd\!= \{s, \langle \mathsf{PUSH}\ n :: c, s\rangle \mid P\ s\}$$

given that the following side condition is met:

$$\forall\, s, P\ s \to \langle \mathsf{PUSH}\ n :: c, s\rangle \lhd \{s, \langle c, n :: s\rangle \mid P\ s\}$$

As we have mentioned earlier, these side conditions are trivial to check. In this particular case, $\langle c, n :: s\rangle$ is the only successor configuration of $\langle \mathsf{PUSH}\ n :: c, s\rangle$. Thus the side condition is met.

Finally, the individual calculation steps are combined via the Trans proof rule, which yields that

$$\{s, \langle \mathsf{comp'}\ (\mathsf{Val}\ n)\ c, s\rangle \mid P\ s\} \Rrightarrow \{s\ n', \langle c, n' :: s\rangle \mid \mathsf{Val}\ n \Downarrow n' \wedge P\ s\}$$

– Val n:

$$
\begin{array}{ll}
& \{\mathsf{s}\ \mathsf{n}',\ \langle \mathsf{c},\ \mathsf{n}' :: \mathsf{s}\rangle \mid \mathsf{Val}\ \mathsf{n} \Downarrow \mathsf{n}' \wedge \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ by } \textsc{Val} \} \\
& \{\mathsf{s},\ \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{P}\ \mathsf{s}\} \\
\Longleftarrow & \quad \{\text{ by VM-}\textsc{Push} \} \\
& \{\mathsf{s},\ \langle \mathsf{PUSH}\ \mathsf{n} :: \mathsf{c},\ \mathsf{s}\rangle \mid \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ definition of comp' } \} \\
& \{\mathsf{s},\ \langle \mathsf{comp'}\ (\mathsf{Val}\ \mathsf{n})\ \mathsf{c},\ \mathsf{s}\rangle \mid \mathsf{P}\ \mathsf{s}\}
\end{array}
$$

– Add e_1 e_2:

$$
\begin{array}{ll}
& \{\mathsf{s}\ \mathsf{n},\ \langle \mathsf{c},\ \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2 \Downarrow \mathsf{n} \wedge \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ by } \textsc{Add} \} \\
& \{\mathsf{s}\ \mathsf{n}\ \mathsf{m},\ \langle \mathsf{c},\ (\mathsf{n} + \mathsf{m}) :: \mathsf{s}\rangle \mid \mathsf{e}_1 \Downarrow \mathsf{n} \wedge \mathsf{e}_2 \Downarrow \mathsf{m} \wedge \mathsf{P}\ \mathsf{s}\} \\
\Longleftarrow & \quad \{\text{ by VM-}\textsc{Add} \} \\
& \{\mathsf{s}\ \mathsf{n}\ \mathsf{m},\ \langle \mathsf{ADD} :: \mathsf{c},\ \mathsf{m} :: \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e}_1 \Downarrow \mathsf{n} \wedge \mathsf{e}_2 \Downarrow \mathsf{m} \wedge \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ move existential quantifier } \} \\
& \{\mathsf{s'}\ \mathsf{m},\ \langle \mathsf{ADD} :: \mathsf{c},\ \mathsf{m} :: \mathsf{s'}\rangle \mid \mathsf{e}_2 \Downarrow \mathsf{m} \wedge (\exists\, \mathsf{s}\ \mathsf{n},\ \mathsf{e}_1 \Downarrow \mathsf{n} \wedge \mathsf{s'} = \mathsf{n} :: \mathsf{s} \wedge \mathsf{P}\ \mathsf{s})\} \\
\Longleftarrow & \quad \{\text{ induction hypothesis for } \mathsf{e}_2 \} \\
& \{\mathsf{s},\ \langle \mathsf{comp'}\ \mathsf{e}_2\ (\mathsf{ADD} :: \mathsf{c})\ ,\ \mathsf{s}\rangle \mid \exists\, \mathsf{s'}\ \mathsf{n},\ \mathsf{e}_1 \Downarrow \mathsf{n} \wedge \mathsf{s} = \mathsf{n} :: \mathsf{s'} \wedge \mathsf{P}\ \mathsf{s'}\} \\
\equiv & \quad \{\text{ move existential quantifier } \} \\
& \{\mathsf{s}\ \mathsf{n},\ \langle \mathsf{comp'}\ \mathsf{e}_2\ (\mathsf{ADD} :: \mathsf{c})\ ,\ \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e}_1 \Downarrow \mathsf{n} \wedge \mathsf{P}\ \mathsf{s}\} \\
\Longleftarrow & \quad \{\text{ induction hypothesis for } \mathsf{e}_1 \} \\
& \{\mathsf{s},\ \langle \mathsf{comp'}\ \mathsf{e}_1\ (\mathsf{comp'}\ \mathsf{e}_2\ (\mathsf{ADD} :: \mathsf{c}))\ ,\ \mathsf{s}\rangle \mid \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ definition of comp' } \} \\
& \{\mathsf{s},\ \langle \mathsf{comp'}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2)\ \mathsf{c},\ \mathsf{s}\rangle \mid \mathsf{P}\ \mathsf{s}\}
\end{array}
$$

– Rnd e:

$$
\begin{array}{ll}
& \{\mathsf{s}\ \mathsf{m},\ \langle \mathsf{c},\ \mathsf{m} :: \mathsf{s}\rangle \mid \mathsf{Rnd}\ \mathsf{e} \Downarrow \mathsf{m} \wedge \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ by } \textsc{Rnd} \} \\
& \{\mathsf{s}\ \mathsf{m}\ \mathsf{n},\ \langle \mathsf{c},\ \mathsf{m} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n} \wedge 0 \leqslant \mathsf{m} \leqslant |\mathsf{n}| \wedge \mathsf{P}\ \mathsf{s}\} \\
\Longleftarrow & \quad \{\text{ by VM-}\textsc{Rnd} \} \\
& \{\mathsf{s}\ \mathsf{m}\ \mathsf{n},\ \langle \mathsf{RND} :: \mathsf{c},\ \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n} \wedge 0 \leqslant \mathsf{m} \leqslant |\mathsf{n}| \wedge \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ eliminate tautology } \exists\, \mathsf{m},\ 0 \leqslant \mathsf{m} \leqslant |\mathsf{n}| \} \\
& \{\mathsf{s}\ \mathsf{n},\ \langle \mathsf{RND} :: \mathsf{c},\ \mathsf{n} :: \mathsf{s}\rangle \mid \mathsf{e} \Downarrow \mathsf{n} \wedge \mathsf{P}\ \mathsf{s}\} \\
\Leftarrow & \quad \{\text{ induction hypothesis for } \mathsf{e} \} \\
& \{\mathsf{s},\ \langle \mathsf{comp'}\ \mathsf{e}\ (\mathsf{RND} :: \mathsf{c})\ ,\ \mathsf{s}\rangle \mid \mathsf{P}\ \mathsf{s}\} \\
\equiv & \quad \{\text{ definition of comp' } \} \\
& \{\mathsf{s},\ \langle \mathsf{comp'}\ (\mathsf{Rnd}\ \mathsf{e})\ \mathsf{c},\ \mathsf{s}\rangle \mid \mathsf{P}\ \mathsf{s}\}
\end{array}
$$

Fig. 4: Correctness proof for compiler comp'

and therefore proves the correctness property for the case Val n.

The calculation for Add e_1 e_2 illustrates the need to generalise the correctness property in order to obtain an induction hypothesis that is strong enough. The induction hypotheses for this case are the following, for each $i \in \{1, 2\}$:

$$\forall\, c'\ P',\quad \{s, \langle comp'\ e_i\ c', s\rangle \mid P'\ s\} \Rightarrow \{s\ n, \langle c', n :: s\rangle \mid e_i \Downarrow n \wedge P'\ s\}$$

The calculation step that uses the induction hypothesis for e_2 instantiates c' with ADD :: c, and the predicate P' with

$$\exists\, s\ n, e_1 \Downarrow n \wedge s' = n :: s \wedge P\ s.$$

The instantiation of the predicate P' in the induction hypothesis allows us to preserve invariants of the stack. The ability to express such invariants is crucial for reasoning about compiler correctness.

Our Coq library that implements this calculational reasoning provides a syntax that is very close to the idealised syntax we used in Figure 4. To illustrate this, Figure 5 shows the full Coq proof of the correctness theorem.

4 Calculating a Compiler

In the previous section, we started out with the definition of the semantics of both the source and the target language of the compiler, together with the definition of the compiler itself. We then set out to to prove that this compiler is correct. The proof by calculation, however, also lends itself to a different setup: given the source language (including its semantics), we want to derive a suitable target language and a compiler that satisfies the correctness property. The idea of deriving a compiler from its specification has been explored in detail by a number of authors [14, 10, 1, 4]. Recently, Bahr and Hutton [4] have shown that such a derivation can be performed by simply stating the correctness property of the compiler and then performing the calculation proof. The parts of the setup that are not defined yet, i.e. the compiler itself and the target language, reveal themselves during the calculation proof.

To illustrate this idea of Bahr and Hutton [4] we reconsider the correctness proof from section 3.2. We need to prove the following property about comp':

$$\forall\, e\ c\ P,\quad \{s, \langle comp'\ e\ c, s\rangle \mid P\ s\} \Rightarrow \{s\ n, \langle c, n :: s\rangle \mid e \Downarrow n \wedge P\ s\}$$

To do so, the calculation proof "transforms" the right-hand side into the left-hand side using the proof rules for $\Rightarrow$. How can we do this, if the compiler comp' is not defined yet? The idea is to transform the right-hand side into the form $\{s, \langle c', s\rangle \mid P\ s\}$ for some c' : Code, and then simply take comp' e c = c' as a *defining* equation for comp'. The calculation proof in Figure 4, can be read this way by simply removing the last step in each case of the proof. These are the only steps that make reference to the definition of comp'. Instead of using the definition of comp', we can interpret the final calculation step as the discovery of how comp' must be defined such that the calculation proof can be completed.

```
Theorem correctness : forall e P c,
  {s, ⟨comp' e c, s ⟩ | P s} =|> {s n, ⟨c , n :: s ⟩ | e ⇓ n /\ P s}.
Proof.
induction e;intros.

begin
  ({s n', ⟨c, n' :: s ⟩ | Val n ⇓ n' /\ P s}).
=  { by_eval }
  ({s, ⟨c, n :: s ⟩ | P s}) .
<== { apply vm_push }
  ({s, ⟨PUSH n :: c, s ⟩ | P s}) .
[].

begin
  ({s n, ⟨c, n :: s ⟩ | Add e1 e2 ⇓ n /\ P s }) .
=  { by_eval }
  ({s n m, ⟨c, (n + m) :: s ⟩ | e1 ⇓ n /\ e2 ⇓ m /\ P s}) .
<== { apply vm_add }
  ({s n m, ⟨ADD :: c, m :: n :: s ⟩ | e1 ⇓ n  /\ e2 ⇓ m /\ P s}).
=  { eauto }
  ({s' m, ⟨ADD :: c, m :: s' ⟩ | e2 ⇓ m
                              /\ (exists s n, e1 ⇓ n /\ s' = n :: s /\ P s)}).
<|= { apply IHe2 }
  ({s, ⟨comp' e2 (ADD :: c), s ⟩ | exists s' n, e1 ⇓ n /\ s = n :: s' /\ P s'}).
=  { eauto }
  ({s n, ⟨comp' e2 (ADD :: c),  n :: s ⟩ | e1 ⇓ n /\ P s }).
<|= { apply IHe1 }
  ({s, ⟨comp' e1 (comp' e2 (ADD :: c)),  s ⟩ | P s }).
[].

begin
  ({s m, ⟨c, m :: s ⟩ | Rnd e ⇓ m /\ P s }) .
=  { by_eval }
  ({s m n, ⟨c, m :: s ⟩ | e ⇓ n /\ 0 <= m <= abs n /\ P s }) .
<== {apply vm_rnd}
  ({s (m:Z) n, ⟨RND :: c, n :: s ⟩ | e ⇓ n  /\ 0 <= m <= abs n /\ P s }).
=  {dist' auto}
  ({s n, ⟨RND :: c, n :: s ⟩ | e ⇓ n /\ P s }) .
<|= { apply IHe }
  ({s, ⟨comp' e (RND :: c), s ⟩ | P s }) .
[].
Qed.
```

Fig. 5: Correctness proof for compiler comp' in Coq.

However, we do not only derive the compiler from the calculation but also the target language and its semantics. The idea is quite simple: we introduce new elements into the type Instr of instructions and corresponding rules for $\Longrightarrow$ such that we can use the STEP proof rule to manipulate the state of the configuration such that we can apply the induction hypothesis or arrive at the target pattern $\{s, \langle c', s\rangle \mid P\ s\}$.

For example, consider the calculation for the case $e = \mathsf{Val}\ n$. We start, as in Figure 4, by using the semantics of the source language:

$$\begin{array}{ll} & \{s\ n', \langle c, n' :: s\rangle \mid \mathsf{Val}\ n \Downarrow n' \wedge P\ s\} \\ \equiv & \quad \{ \text{ by VAL } \} \\ & \{s, \langle c, n :: s\rangle \mid P\ s\} \end{array}$$

Our goal is to transform $\{s, \langle c, n :: s\rangle \mid P\ s\}$ by a sequence of calculation steps into the form $\{s, \langle c', s\rangle \mid P\ s\}$. That means that we have to get rid of the n on top of the stack. That is, we need to find a c' such that $\langle c', s\rangle \Longrightarrow \langle c, n :: s\rangle$. We could achieve that by simply adding a rule $\langle c, s\rangle \Longrightarrow \langle c, n :: s\rangle$ to the definition of $\Longrightarrow$, i.e. n is chosen non-deterministically. However, if we added this rule, we would not be able to derive

$$\{s, \langle c, s\rangle \mid P\ s\} \Rightarrow \{s, \langle c, n :: s\rangle \mid P\ s\}$$

since $\langle c, s\rangle$ is not barred by $\{s, \langle c, n :: s\rangle \mid P\ s\}$ for all s with $P\ s$; cf. rule STEP in Figure 3. The reason why this fails is that we have $\langle c, s\rangle \Longrightarrow \langle c, m :: s\rangle$ for *any* m, not only $m = n$.

Hence, we have to restrict the rule such that it only pushes integers m onto the stack that are equal to n. The only way we can achieve this is by "storing" the relevant information – i.e. n itself – in the code part of the configuration. Hence, we add a constructor $\mathsf{PUSH} : \mathbb{Z} \to \mathsf{Instr}$ to the type of instructions, which allows us to add the rule $\langle \mathsf{PUSH}\ n :: c, s\rangle \Longrightarrow \langle c, n :: s\rangle$ to the definition of $\Longrightarrow$. Using the STEP proof rule, we can then conclude the calculation as follows:

$$\begin{array}{ll} & \{s, \langle c, n :: s\rangle \mid P\ s\} \\ \Longleftarrow & \quad \{ \text{ define } \langle \mathsf{PUSH} :: c, s\rangle \Longrightarrow \langle c, n :: s\rangle \ \} \\ & \{s, \langle \mathsf{PUSH} :: c, n :: s\rangle \mid P\ s\} \\ \equiv & \quad \{ \text{ define } \mathsf{comp'}\ (\mathsf{Val}\ n)\ c = \mathsf{PUSH}\ n :: c \ \} \\ & \{s, \langle \mathsf{comp'}\ (\mathsf{Val}\ n)\ c, n :: s\rangle \mid P\ s\} \end{array}$$

The same can be done for the other two cases of the Expr language. The only difference is that we need to use the induction hypothesis. In order to be able to apply the induction hypothesis, we need to transform the configuration into the right shape. We do this by adding rules to the definition of $\Longrightarrow$ and then applying the STEP proof rule accordingly. In the end, we arrive at the very same calculation proof as in Figure 4. But instead of having comp' and $\Longrightarrow$ defined beforehand and using it in the proof, we discover the definition of comp' and $\Longrightarrow$ as we do the calculation. In fact, the compiler and the target language presented in section 2 have been derived using this approach.

The setup used by Bahr and Hutton [4] to do the calculation is slightly different: the semantics of the source language is given by a structurally recursive evaluation function eval of type $\mathsf{Exp} \to \mathsf{Value}$ and the virtual machine is formulated as a tail recursive function exec of type $\mathsf{Code} \to \mathsf{Conf'} \to \mathsf{Conf'}$. The compiler correctness property is then formulated as an equation that relates eval and exec. Moreover, the Code type is not part of the configuration type $\mathsf{Conf'}$. However, it is easy to translate eval into a big-step operational semantics – in fact, Bahr and Hutton [4] do this in their treatment of higher-order languages – and exec into a big-step operational semantics [5]. The methodology of Bahr and Hutton [4] can be adapted to the use of a small-step virtual machine $\Longrightarrow$ by reasoning over the reflexive transitive closure $\stackrel{*}{\Longrightarrow}$ instead of equational reasoning. But this approach does not work for non-deterministic languages, since the calculation would only prove completeness but not soundness.[3] The example below illustrates this.

Assume that during the calculation we derived a more general rule for the RND instruction (instead of the rule VM-Rnd in Figure 2):

$$\langle \mathsf{RND} :: \mathsf{c}, \mathsf{n} :: \mathsf{s} \rangle \Longrightarrow \langle \mathsf{c}, \mathsf{m} :: \mathsf{s} \rangle \qquad \text{(VM-Rnd')}$$

The above rule omits the side condition $0 \leqslant \mathsf{m} \leqslant |\mathsf{n}|$. As a result the compiler comp is not sound anymore. The virtual machine now allows the following execution, even though we do not have that $\mathsf{RND}\ (\mathsf{Val}\ 0) \Downarrow 1$:

$$\langle \mathsf{comp}\ (\mathsf{RND}\ (\mathsf{Val}\ 0)), [] \rangle = \langle [\mathsf{PUSH}\ 0; \mathsf{RND}], [] \rangle \Longrightarrow \langle [\mathsf{RND}], [0] \rangle \Longrightarrow \langle [], [1] \rangle$$

The calculation approach of Bahr and Hutton [4] (naively lifted to relational semantics as describes above) will not detect that the rule VM-Rnd' breaks the soundness property. This calculation approach roughly corresponds to the use of the proof rules in Figure 3 but with the "barred" side condition removed from the Step proof rule. The resulting proof system is admissible for $\Longrightarrow\!\!\!\!\!\triangleright$ but not for $\Rightarrow\!\!\triangleright$. Thus the corresponding calculation only proves the completeness property, but not the soundness property.

The problem illustrated above is easy to recognise for the simple toy language that we considered here. However, in the next section we will consider a more complex language, where such problems are much more subtle as we will see. The novelty of our approach lies in the proof rules for the $\Rightarrow\!\!\triangleright$ relation that combines the two relations $\stackrel{*}{\Longrightarrow}$ and $\triangleleft$. As a consequence, we are able to formulate soundness and completeness in one compact statement and calculate in a style similar to Bahr and Hutton [4].

[3] Bahr and Hutton [4] acknowledge that this problem already occurs if the semantics is not total, e.g. for the untyped lambda calculus. However, if the semantics is at least deterministic, soundness for the defined fragment of the language can be achieved easily by ensuring that the derived virtual machine is deterministic.

5 Calculating a Compiler for a Language with Interrupts

We now turn to a more interesting source language that features asynchronous exceptions, also known as *interrupts*. What distinguishes interrupts from ordinary exceptions is the fact that interrupts can potentially arise at (almost) any point in the execution of a program. As a consequence, the language's semantics is non-deterministic – a program's execution may proceed successfully or be interrupted by an asynchronous exception at any point. In addition, we consider language constructs that allow the programmer to limit the scope of interrupts, i.e. blocking interrupts from interfering with some parts of the program.

We consider the language with interrupts of Hutton and Wright [7]. The syntax of the language is given by the following inductive type:

```
Inductive Expr : Set := Val (n : ℤ) | Add (e1 e2 : Expr)
                      | Throw | Catch (e h : Expr) | Seqn (e1 e2 : Expr)
                      | Block (e : Expr) | Unblock (e : Expr) .
```

As before, we have integer literals (Val) and an addition operation (Add). Furthermore, the language allows us to throw (synchronous) exceptions with Throw and to catch (synchronous or asynchronous) exceptions with Catch. An expression of the form Catch e h behaves like e, in case e does not throw any exceptions; otherwise it behaves like h. We can also sequentially compose expressions using Seqn. Finally, Block and Unblock are used to control asynchronous exceptions: in an expression Block e or Unblock e, we allow respectively disallow interruption in e by asynchronous exceptions.

We give the semantics of the language as a big-step operational semantics as presented by Hutton and Wright [7]. To describe blocking and unblocking of interrupts, the semantics uses the following type to indicate the blocking status, where B indicates that interrupts are blocked and U that interrupts are allowed:

```
Inductive Status : Set := B | U .
```

The relation that embodies the big-step operational semantics is denoted by $\Downarrow_s$, where the subscript s indicates the status, i.e. either blocked or unblocked. The judgement e $\Downarrow_s$ v means that e : Expr evaluates to v : option ℤ given the status s : Status. Values of type option ℤ are either of the form Some n, indicating the result value n, or of the form None, indicating that an exception occurred. The inference rules for the semantics are shown in Figure 6.

Our goal is to derive a compiler and virtual machine such that the compiler is correct with respect to the source language's semantics. To this end, we follow the calculation approach outlined in section 4.

We start with a partially defined compiler comp:

```
Fixpoint comp' (e : Expr) (c : Code) : Code :=
  match e with
    | _ ⇒ Admit
  end .
```

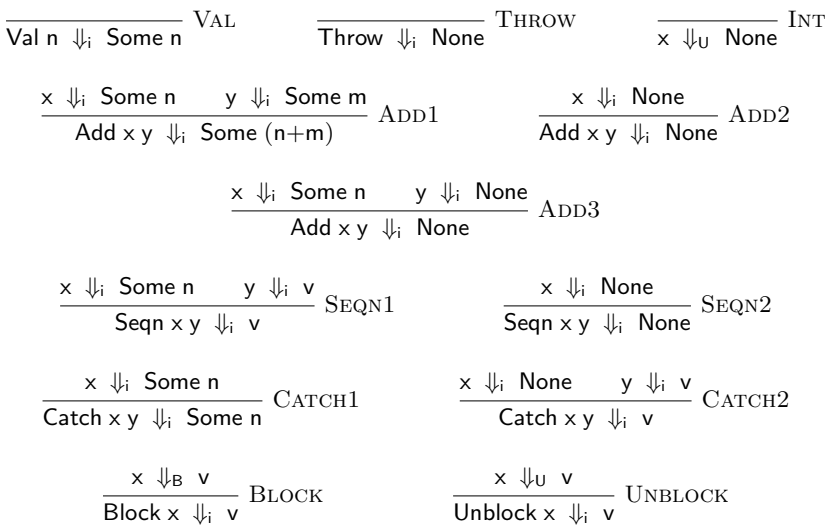

Fig. 6: Semantics of the language.

```
Definition comp (e : Expr) : Code := comp' e [].
```

Similarly to the compiler in section 2, the above compiler is defined with an additional argument representing the code that is supposed to be executed after the generated code. As we do the calculation proof, we will discover equations of the form `comp' p c = c'` for some pattern `p`. We will then add a corresponding clause `p ⇒ c'` to the **match** statement in the above definition. For now it uses the term `Admit` as a placeholder. It serves a similar role as the term `undefined` in Haskell.

Likewise, we start with an empty definition of the target language and the virtual machine:

```
Inductive Instr : Set :=.
Inductive VM : Conf → Conf → Prop :=
  where "x ==> y" := (VM x y) .
```

We then have to formulate the correctness property of the compiler. We adopt the same form of correctness property as in section 3:

```
∀ e c i P,  {s, ⟨comp' e c, s⟩ | P s} ⇒ {s n, ⟨c, n :: s⟩ | e ⇓ᵢ Some n ∧ P s}
```

We could now start the calculation. However, we would soon realise that the above property is not appropriate. We would encounter three problems:

1. The status indicator `i` only appears in the semantics of the source language but not the virtual machine. Thus, the above property is too general.

2. We only consider the case that $\mathsf{e} \Downarrow_{\mathsf{i}} \mathsf{Some\ n}$, but not the case that $\mathsf{e} \Downarrow_{\mathsf{i}} \mathsf{None}$. Thus, the above property is not general enough.
3. As we do the calculation, we realise that we need to store data other than just integers on the stack.

The calculation technique of Bahr and Hutton [4] anticipates these problems and suggests corresponding generalisations, which we shall adopt here as well. However, we have to translate their approach, which uses a tail-recursive function instead of a small-step relation for defining the virtual machine. We have to make the following amendments:

1. Extend the type of configurations with a component of type Status:

$$\textbf{Inductive}\ \mathsf{Conf}\ :\ \mathsf{Set} := \mathsf{conf}\ (\mathsf{c}\ :\ \mathsf{Code})\ (\mathsf{s}\ :\ \mathsf{Stack})\ (\mathsf{i}\ :\ \mathsf{Status}).$$
$$\textbf{Notation}\ \texttt{"⟨ c , s , i ⟩"} := (\mathsf{conf\ c\ s\ i})\ .$$

2. Add a case to the type of configurations that corresponds to the None case:[4]

$$\begin{aligned}&\textbf{Inductive}\ \mathsf{Conf}\ :\ \mathsf{Set} := \mathsf{conf}\ (\mathsf{c}\ :\ \mathsf{Code})\ (\mathsf{s}\ :\ \mathsf{Stack})\ (\mathsf{i}\ :\ \mathsf{Status})\\ &\qquad\qquad\qquad\qquad\ \ |\ \ \mathsf{fail}\ (\mathsf{s}\ :\ \mathsf{Stack})\ (\mathsf{i}\ :\ \mathsf{Status}).\\ &\textbf{Notation}\ \texttt{"⟨ c , s , i ⟩"} := (\mathsf{conf\ c\ s\ i}).\\ &\textbf{Notation}\ \texttt{"⟪ s , i ⟫"}\quad\ := (\mathsf{fail\ s\ i})\ .\end{aligned}$$

3. Generalise the type of stack elements and extend the type as necessary:

$$\textbf{Inductive}\ \mathsf{Elem}\ :\ \mathsf{Set} := \mathsf{VAL}\ (\mathsf{n}\ :\ \mathbb{Z}).$$
$$\textbf{Definition}\ \mathsf{Stack}\ :\ \mathsf{Set} := \mathsf{list\ Elem}\ .$$

The type Stack as defined above is isomorphic to the previous definition. However, we can now extend the type Elem with additional constructors to store other kinds of data on the stack.

With these changes we can formulate the correctness property as follows:

$$\begin{aligned}\forall\ \mathsf{e\ c\ P},\quad &\{\mathsf{s\ i}, \langle \mathsf{comp'\ e\ c}, \mathsf{s}, \mathsf{i}\rangle\ \mid\ \mathsf{P\ s\ i}\} \Rightarrow\\ &\qquad \{\mathsf{s\ i\ n}, \langle \mathsf{c}, \mathsf{VAL\ n} :: \mathsf{s}, \mathsf{i}\rangle \mid \mathsf{e} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge\ \mathsf{P\ s\ i}\}\\ &\cup \{\mathsf{s\ i},\quad \langle\!\langle \mathsf{s}, \mathsf{i}\rangle\!\rangle \qquad\quad \mid \mathsf{e} \Downarrow_{\mathsf{i}} \mathsf{None}\quad \wedge\ \mathsf{P\ s\ i}\}\end{aligned}$$

Note that the predicate P is now applied to s *and* i. We could have started the calculation with P only covering s, but we would soon realise that we need the more general version of P in order to apply the induction hypothesis.

An important difference between the above correctness property and the correctness property we have considered in section 3 is that it involves set union. As a consequence, we need to use the UNION proof rule from Figure 3. We will

[4] Bahr and Hutton [4] use tail-recursive functions to represent virtual machines. In their approach, one has to introduce an additional tail-recursive function fail. In our approach, this corresponds to a new constructor for the type of configurations.

always use the UNION rule together with the REFL rule, such that we only change one component of a union while the others remain constant. For example, given $\mathsf{X} \Rightarrow \mathsf{X'}$, we may derive $\mathsf{W} \cup \mathsf{X} \cup \mathsf{Y} \Rightarrow \mathsf{W} \cup \mathsf{X'} \cup \mathsf{Y}$.

Figure 7 shows the calculation for the two cases $\mathsf{e} = \mathsf{Val\ n}$ and $\mathsf{e} = \mathsf{Catch\ e_1\ e_2}$. We focus on these two cases to illustrate the calculation. The complete calculation covering the remaining cases as well can be found in the accompanying Coq code.

We begin with the case $\mathsf{e} = \mathsf{Val\ n}$. First, we use the definition of $\Downarrow_\mathsf{i}$: we can derive $\mathsf{Val\ n} \Downarrow_\mathsf{i} \mathsf{Some\ n'}$ iff $\mathsf{n} = \mathsf{n'}$; and we can derive $\mathsf{Val\ n} \Downarrow_\mathsf{i} \mathsf{None}$ iff $\mathsf{i} = \mathsf{U}$. We can then proceed to transform the configuration set such that it matches the left-hand side of the correctness statement, i.e. it is of the form $\{\mathsf{s\ i}, \langle \mathsf{c'}, \mathsf{s}, \mathsf{i} \rangle \mid \mathsf{P\ s\ i}\}$. In particular, we must get rid of the union construction.

The general strategy to achieve this is to transform the union into a form

$$\{\mathsf{s\ i}, \langle \mathsf{c'}, \mathsf{s}, \mathsf{i} \rangle \mid \mathsf{P_1\ s\ i}\} \cup \{\mathsf{s\ i}, \langle \mathsf{c'}, \mathsf{s}, \mathsf{i} \rangle \mid \mathsf{P_2\ s\ i}\}$$

such that we can replace it with a single set comprehension

$$\{\mathsf{s\ i}, \langle \mathsf{c'}, \mathsf{s}, \mathsf{i} \rangle \mid \mathsf{P_1\ s\ i} \vee \mathsf{P_2\ s\ i}\}$$

In most cases, we have that $\mathsf{P_1}$ implies $\mathsf{P_2}$, or vice versa such that we can replace the union with the right or the left component of the union, respectively.

We first consider the left component of the union. Similarly to the simple language of section 2, we introduce an instruction PUSH in order to get rid of the topmost stack element VAL n. In order to apply the above strategy to transform a union of set comprehensions into a single set comprehension, we must be able to transform the configuration $\langle\!\langle \mathsf{s}, \mathsf{U} \rangle\!\rangle$ into the form $\langle \mathsf{PUSH\ n} :: \mathsf{c}, \mathsf{s'}, \mathsf{i'} \rangle$ such that $\mathsf{P\ s\ U}$ implies $\mathsf{P\ s'\ i'}$. Hence, we must have $\mathsf{s'} = \mathsf{s}$ and $\mathsf{i'} = \mathsf{U}$. We thus decide to add the rule

$$\langle \mathsf{PUSH\ n} :: \mathsf{c}, \mathsf{s}, \mathsf{U} \rangle \Longrightarrow \langle\!\langle \mathsf{s}, \mathsf{U} \rangle\!\rangle$$

We could have also added the more general rule

$$\langle \mathsf{PUSH\ n} :: \mathsf{c}, \mathsf{s}, \mathsf{i} \rangle \Longrightarrow \langle\!\langle \mathsf{s}, \mathsf{i} \rangle\!\rangle$$

However, by adding this rule, the calculation step that we did before would not be correct anymore since we would not have that $\mathsf{P\ s\ i}$ implies that

$$\langle \mathsf{PUSH\ n} :: \mathsf{c}, \mathsf{s}, \mathsf{i} \rangle \triangleleft \{\mathsf{s\ i}, \langle \mathsf{c}, \mathsf{VAL\ n} :: \mathsf{s}, \mathsf{i} \rangle \mid \mathsf{P\ s\ i}\} \cup \{\mathsf{s}, \langle\!\langle \mathsf{s}, \mathsf{U} \rangle\!\rangle \mid \mathsf{P\ s\ U}\}$$

This phenomenon illustrates the utility of the proof automation of our calculation framework that discharges side conditions as the one above: by adding new rules to the definition of the virtual machine, applications of the STEP proof rule have to be reconsidered and may fail as the side condition is not fulfilled anymore.

The union is almost in the right form now. The right component can be equivalently written as

$$\{\mathsf{s\ i}, \langle \mathsf{PUSH\ n} :: \mathsf{c}, \mathsf{s}, \mathsf{i} \rangle \mid \mathsf{i} = \mathsf{U} \wedge \mathsf{P\ s\ i}\}$$

– Val n:

$$
\begin{array}{ll}
 & \{\mathsf{s\ i\ n'}, \langle \mathsf{c, VAL\ n' :: s, i}\rangle \mid \mathsf{Val\ n} \Downarrow_{\mathsf{i}} \mathsf{Some\ n'} \wedge \mathsf{P\ s\ i}\} \\
 & \cup \{\mathsf{s\ i}, \langle\!\langle \mathsf{s, i}\rangle\!\rangle \mid \mathsf{Val\ n} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i}\} \\
\equiv & \quad \{ \text{ by definition of } \Downarrow_{\mathsf{i}} \} \\
 & \{\mathsf{s\ i}, \langle \mathsf{c, VAL\ n :: s, i}\rangle \mid \mathsf{P\ s\ i}\} \cup \{\mathsf{s}, \langle\!\langle \mathsf{s, U}\rangle\!\rangle \mid \mathsf{P\ s\ U}\} \\
\Longleftarrow & \quad \{ \text{ define } \langle \mathsf{PUSH\ n :: c, s, i}\rangle \Longrightarrow \langle\ \mathsf{c, VAL\ n :: s, i}\ \rangle \} \\
 & \{\mathsf{s\ i}, \langle\ \mathsf{PUSH\ n :: c, s, i}\rangle \mid \mathsf{P\ s\ i}\} \cup \{\mathsf{s}, \langle\!\langle \mathsf{s, U}\rangle\!\rangle \mid \mathsf{P\ s\ U}\} \\
\Longleftarrow & \quad \{ \text{ define } \langle \mathsf{PUSH\ n :: c, s, U}\rangle \Longrightarrow \langle\!\langle \mathsf{s, U}\rangle\!\rangle \} \\
 & \{\mathsf{s\ i}, \langle \mathsf{PUSH\ n :: c, s, i}\rangle \mid \mathsf{P\ s\ i}\} \cup \{\mathsf{s}, \langle \mathsf{PUSH\ n :: c, s, U}\rangle \mid \mathsf{P\ s\ U}\} \\
\equiv & \quad \{ \text{ second set of the union is contained in first set } \} \\
 & \{\mathsf{s\ i}, \langle \mathsf{PUSH\ n :: c, s, i}\rangle \mid \mathsf{P\ s\ i}\}
\end{array}
$$

– Catch $\mathsf{e_1}$ $\mathsf{e_2}$:

$$
\begin{array}{ll}
 & \{\mathsf{s\ i\ n}, \langle \mathsf{c, VAL\ n :: s, i}\rangle \mid \mathsf{Catch\ e_1\ e_2} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge \mathsf{P\ s\ i}\} \\
 & \cup \{\mathsf{s\ i}, \langle\!\langle \mathsf{s, i}\rangle\!\rangle \mid \mathsf{Catch\ e_1\ e_2} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i}\} \\
\equiv & \quad \{ \text{ by definition of } \Downarrow_{\mathsf{i}} \} \\
 & \{\mathsf{s\ i\ n}, \langle \mathsf{c, VAL\ n :: s, i}\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge \mathsf{P\ s\ i}\} \\
 & \cup \{\mathsf{s\ i\ n}, \langle \mathsf{c, VAL\ n :: s, i}\rangle \mid \mathsf{e_2} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge (\mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i})\} \\
 & \cup \{\mathsf{s\ i}, \langle\!\langle \mathsf{s, i}\rangle\!\rangle \mid \mathsf{e_2} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge (\mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i})\} \\
\Longleftarrow & \quad \{ \text{ induction hypothesis for } \mathsf{e_2} \} \\
 & \{\mathsf{s\ i\ n}, \langle \mathsf{c, VAL\ n :: s, i}\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge \mathsf{P\ s\ i}\} \\
 & \cup \{\mathsf{s\ i}, \langle \mathsf{comp'\ e_2\ c, s, i}\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i}\} \\
\Longleftarrow & \quad \{ \text{ define } \langle\!\langle \mathsf{HAN\ h :: s, i}\rangle\!\rangle \Longrightarrow \langle \mathsf{h,s,i}\rangle \} \\
 & \{\mathsf{s\ i\ n}, \langle \mathsf{c, VAL\ n :: s, i}\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge \mathsf{P\ s\ i}\} \\
 & \cup \{\mathsf{s\ i}, \langle\!\langle \mathsf{HAN\ (comp'\ e_2\ c) :: s, i}\rangle\!\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i}\} \\
\Longleftarrow & \quad \{ \text{ define } \langle \mathsf{UNMARK :: c, VAL\ n :: HAN\ h :: s, i}\rangle \Longrightarrow \langle \mathsf{c, VAL\ n :: s, i}\rangle \} \\
 & \{\mathsf{s\ i\ n}, \langle \mathsf{UNMARK :: c, VAL\ n :: HAN\ (comp'\ e_2\ c) :: s, i}\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge \mathsf{P\ s\ i}\} \\
 & \cup \{\mathsf{s\ i}, \langle\!\langle \mathsf{HAN\ (comp'\ e_2\ c) :: s, i}\rangle\!\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i}\} \\
\equiv & \quad \{ \text{ move stack element } \mathsf{HAN\ (comp'\ e_2\ c)} \text{ into the predicate } \} \\
 & \{\mathsf{s'\ i\ n}, \langle \mathsf{UNMARK :: c, VAL\ n :: s', i}\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge \\
 & \qquad\qquad (\exists\, \mathsf{s, s'} = \mathsf{HAN\ (comp'\ e_2\ c) :: s} \wedge \mathsf{P\ s\ i})\} \\
 & \cup \{\mathsf{s'\ i}, \langle\!\langle \mathsf{s', i}\rangle\!\rangle \mid \mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge (\exists\, \mathsf{s, s'} = \mathsf{HAN\ (comp'\ e_2\ c) :: s} \wedge \mathsf{P\ s\ i})\} \\
\Longleftarrow & \quad \{ \text{ induction hypothesis for } \mathsf{e_1} \} \\
 & \{\mathsf{s'\ i}, \langle \mathsf{comp'\ e_1\ (UNMARK :: c)\ , s', i}\rangle \mid (\exists\, \mathsf{s, s'} = \mathsf{HAN\ (comp'\ e_2\ c) :: s} \wedge \mathsf{P\ s\ i})\} \\
\equiv & \quad \{ \text{ extract stack element } \mathsf{HAN\ (comp'\ e_2\ c)} \text{ from the predicate } \} \\
 & \{\mathsf{s\ i}, \langle \mathsf{comp'\ e_1\ (UNMARK :: c)\ , HAN\ (comp'\ e_2\ c) :: s, i}\rangle \mid \mathsf{P\ s\ i}\}. \\
\Longleftarrow & \quad \{ \text{ define } \langle \mathsf{MARK\ h :: c, s, i}\rangle \Longrightarrow \langle \mathsf{c, HAN\ h :: s, i}\rangle \} \\
 & \{\mathsf{s\ i}, \langle \mathsf{MARK\ (comp'\ e_2\ c) :: comp'\ e_1\ (UNMARK :: c)\ , s, i}\rangle \mid \mathsf{P\ s\ i}\}
\end{array}
$$

Fig. 7: Calculation for cases Val and Catch.

Since $\mathsf{i} = \mathsf{U} \wedge \mathsf{P\ s\ i}$ obviously implies $\mathsf{P\ s\ i}$, we can replace the union with its left component.

Finally, we can observe that $\mathsf{comp'\ (Val\ n)\ c}$ must be equal to $\mathsf{PUSH\ n :: c}$ and we thus add the clause $\mathsf{Val\ n \Rightarrow PUSH\ n :: c}$ to the definition of comp'.

The calculation for the case $\mathsf{e = Catch\ e_1\ e_2}$ may use the induction hypotheses for $\mathsf{e_1}$ and $\mathsf{e_2}$, which reads as follows:

$$\begin{aligned}\forall\, \mathsf{c'\ P'},\quad \{\mathsf{s\ i},\ \langle \mathsf{comp'\ e_j\ c', s, i}\rangle \mid \mathsf{P'\ s\ i}\} \Rightarrow\ & \\ \{\mathsf{s\ i\ n},\ \langle \mathsf{c', VAL\ n :: s, i}\rangle \mid \mathsf{e_j} \Downarrow_{\mathsf{i}} \mathsf{Some\ n} \wedge \mathsf{P'\ s\ i}\} & \\ \cup\, \{\mathsf{s\ i},\quad \langle\!\langle \mathsf{s, i}\rangle\!\rangle \mid \mathsf{e_j} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P'\ s\ i}\} &\end{aligned}$$

The calculation is driven by the desire to apply these induction hypotheses, which means we want to transform the configuration set such that it matches the right-hand side of one of the induction hypotheses. Achieving this for $\mathsf{e_2}$ is easy: we use the definition of $\Downarrow_{\mathsf{i}}$ to reformulate $\mathsf{Catch\ e_1\ e_2} \Downarrow_{\mathsf{i}} \ldots$ in terms of $\mathsf{e_1} \Downarrow_{\mathsf{i}} \ldots$ and $\mathsf{e_2} \Downarrow_{\mathsf{i}} \ldots$. The second set comprehension together with the third one then already have the right shape for the induction hypothesis for $\mathsf{e_2}$. We instantiate c' with c and $\mathsf{P'\ s\ i}$ with $\mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None} \wedge \mathsf{P\ s\ i}$.

We then have to transform the resulting union of two set comprehensions into the right shape for the induction hypothesis for $\mathsf{e_1}$. At first we notice that the second set comprehension has the condition $\mathsf{e_1} \Downarrow_{\mathsf{i}} \mathsf{None}$, and thus we have to transform it such that the configuration is of the shape $\langle\!\langle \mathsf{s', i'}\rangle\!\rangle$ and not $\langle \mathsf{c', s', i'}\rangle$. That means that we need to have a rule for $\Longrightarrow$ that transforms something of the form $\langle\!\langle \mathsf{s', i'}\rangle\!\rangle$ into $\langle \mathsf{comp'\ e_2\ c, s, i}\rangle$. To do so, we need to be able to draw the code component $\mathsf{comp'\ e_2\ c}$ of the target configuration from s' (or i', but that is not possible). Hence, we introduce a new stack element constructor $\mathsf{HAN : Code \to Elem}$, which is able to store code on the stack. Thus the only reasonable choice for the new rule is $\langle\!\langle \mathsf{HAN\ h :: s, i}\rangle\!\rangle \Longrightarrow \langle \mathsf{h,s,i}\rangle$.

Next, in order to be able to apply the induction hypothesis, we must manipulate the stack in the first set comprehension. It is of the form $\mathsf{VAL\ n :: s}$, but it needs to be of the form $\mathsf{VAL\ n :: HAN\ (comp'\ e_2\ c) :: s}$. To this end, we introduce an instruction UNMARK, which removes the HAN element from the stack. It is then a simple matter of moving the constraint on the shape of the stack into the predicate of the set comprehension such that we can apply the induction hypothesis. After we have applied the induction hypothesis for $\mathsf{e_1}$, we reverse this encoding.

Finally, we need to get rid of the top element of the stack such that the stack becomes just s. As in previous cases, we introduce an instruction that does this job. This completes the calculation for this case, and we can now read off the clause that we need to add to the definition of comp':

$$\mathsf{Catch\ e_1\ e_2 \Rightarrow MARK\ (comp'\ e_2\ c) :: comp'\ e_1\ (UNMARK :: c)}$$

The other cases of the calculation can be completed using the same strategies that we have used above. As the result of the calculation, we derive the following compiler definition:

```
Fixpoint comp' (e : Expr) (c : Code) : Code :=
  match e with
    | Val n       ⇒ PUSH n :: c
    | Add x y     ⇒ comp' x (comp' y (ADD :: c))
    | Throw       ⇒ [THROW]
    | Catch e1 e2 ⇒ MARK (comp' e2 c) :: comp' e1 (UNMARK :: c)
    | Seqn e1 e2  ⇒ comp' e1 (POP :: comp' e2 c)
    | Block e     ⇒ BLOCK :: comp' e (RESET :: c)
    | Unblock e   ⇒ UNBLOCK :: comp' e (RESET :: c)
  end .
```

The derived target language is the following:

```
Inductive Instr : Set := PUSH (n : ℤ) | ADD | THROW
                       | UNMARK | MARK (h : list Instr)
                       | POP | RESET | BLOCK | UNBLOCK .
```

Moreover, during the calculation we needed to extend the type of stack elements such that we can store exception handlers and interrupt status on the stack:

```
Inductive Elem : Set := VAL (n : ℤ) | HAN (c : Code) | INT (s : Status).
```

This is almost the same compiler as the one given by Hutton and Wright [7]. There are only two minor differences: Hutton and Wright's compiler compiles Throw to THROW :: c instead of [THROW]; and instead of the two instructions BLOCK and UNBLOCK they use a single instruction SET, which takes an argument of type Status such that SET B and SET U correspond to BLOCK and UNBLOCK, respectively.

However, these differences are rather superficial. More interesting differences can be found in the virtual machine that we derived from the calculation. The definition of the virtual machine is shown in Figure 8. The virtual machine used by Hutton and Wright [7] may go into a fail configuration from any unblocked configuration, no matter what the current instruction is. In our virtual machine only the instructions PUSH, THROW and BLOCK may be interrupted. It turns out that there is some room of freedom in choosing appropriate rules for $\Longrightarrow$.

We could have equally well made different choices during the calculation process, which would have resulted in a virtual machine equivalent to Hutton and Wright's.[5] For example, for the case e = Val n, we have introduced the rule

$$\langle \mathsf{PUSH\ n} :: \mathsf{c}, \mathsf{s}, \mathsf{U} \rangle \implies \langle\!\langle \mathsf{s}, \mathsf{U} \rangle\!\rangle$$

i.e. the PUSH instruction may be interrupted. However, we could have instead introduced the more general rule

$$\langle \mathsf{op} :: \mathsf{c}, \mathsf{s}, \mathsf{U} \rangle \implies \langle\!\langle \mathsf{s}, \mathsf{U} \rangle\!\rangle$$

[5] Similarly, we could have chosen to use a single instruction SET instead of BLOCK and UNBLOCK.

$$
\begin{aligned}
\langle \mathsf{PUSH\ n} :: \mathsf{c, s, i} \rangle &\Longrightarrow \langle\ \mathsf{c, VAL\ n} :: \mathsf{s, i}\ \rangle \\
\langle \mathsf{PUSH\ n} :: \mathsf{c, s, U} \rangle &\Longrightarrow \langle\!\langle \mathsf{s, U} \rangle\!\rangle \\
\langle \mathsf{ADD} :: \mathsf{c, VAL\ m} :: \mathsf{VAL\ n} :: \mathsf{s, i} \rangle &\Longrightarrow \langle \mathsf{c, VAL\ (n + m)} :: \mathsf{s, i} \rangle \\
\langle\!\langle \mathsf{VAL\ m} :: \mathsf{s, i} \rangle\!\rangle &\Longrightarrow \langle\!\langle \mathsf{s, i} \rangle\!\rangle \\
\langle \mathsf{THROW} :: \mathsf{c, s, i} \rangle &\Longrightarrow \langle\!\langle \mathsf{s, i} \rangle\!\rangle \\
\langle\!\langle \mathsf{HAN\ h} :: \mathsf{s, i} \rangle\!\rangle &\Longrightarrow \langle \mathsf{h,s,i} \rangle \\
\langle \mathsf{UNMARK} :: \mathsf{c, VAL\ n} :: \mathsf{HAN\ h} :: \mathsf{s, i} \rangle &\Longrightarrow \langle \mathsf{c, VAL\ n} :: \mathsf{s, i} \rangle \\
\langle \mathsf{MARK\ h} :: \mathsf{c, s, i} \rangle &\Longrightarrow \langle \mathsf{c, HAN\ h} :: \mathsf{s, i} \rangle \\
\langle \mathsf{POP} :: \mathsf{c, VAL\ n} :: \mathsf{s, i} \rangle &\Longrightarrow \langle \mathsf{c, s, i} \rangle \\
\langle \mathsf{RESET} :: \mathsf{c, VAL\ n} :: \mathsf{INT\ i} :: \mathsf{s, j} \rangle &\Longrightarrow \langle \mathsf{c, VAL\ n} :: \mathsf{s, i} \rangle \\
\langle \mathsf{BLOCK} :: \mathsf{c, s, i} \rangle &\Longrightarrow \langle \mathsf{c, INT\ i} :: \mathsf{s, B} \rangle \\
\langle \mathsf{BLOCK} :: \mathsf{c, s, U} \rangle &\Longrightarrow \langle\!\langle \mathsf{s, U} \rangle\!\rangle \\
\langle \mathsf{UNBLOCK} :: \mathsf{c, s, i} \rangle &\Longrightarrow \langle \mathsf{c, INT\ i} :: \mathsf{s, U} \rangle \\
\langle\!\langle \mathsf{INT\ i} :: \mathsf{s, j} \rangle\!\rangle &\Longrightarrow \langle\!\langle \mathsf{s,i} \rangle\!\rangle
\end{aligned}
$$

Fig. 8: Definition of the virtual machine.

i.e. any instruction may be interrupted, which is the semantics that Hutton and Wright chose for their virtual machine. The fact that the calculation is performed in one go makes this flexibility in the semantics of the virtual machine immediately apparent. The calculation that yields the compiler and virtual machine of Hutton and Wright can be found in the accompanying Coq code.

6 Representation in Coq

In this section, we briefly outline some of the technical setup for our calculation framework in Coq. In addition to the proof rules that we discussed in this paper, our calculation framework consists of three essential components: a representation of sets of configurations suitable for proof automation, a syntax for calculation proofs, and proof tactic that is able to prove the "barred" side condition of the Step proof rule automatically. This setup – including the proof rules – is independent of the specific definition of the virtual machine and the compiler as well as the source and the target language. In particular, the framework is defined as a functor (i.e. a parametrised module) that takes the definition of the virtual machine (given by Conf and $\Longrightarrow$) as a parameter.

6.1 Sets and Set Comprehensions

In our calculations, we need to reason over sets of configurations. The most straightforward and general representation of such sets is provided by the type Conf $\rightarrow$ Prop, i.e. the type of predicates over configurations. However, in order

to prove properties over such sets and – most importantly – to automate proofs, it is better to define an inductive type ConfSet together with an interpretation function that maps each such set to a predicate of type Conf → Prop.

At first we define a type of set comprehensions, which is indexed by the list of types that are available for existential quantification:

```
Inductive SetCom : list Type → Type :=
  | BaseSet      : Conf → Prop → SetCom []
  | ExSet {t ts} : (t → SetCom ts) → SetCom (t :: ts) .
```

The first constructor defines a singleton set that consists of a single configuration that is subject to a predicate[6]. The second constructor adds an existential quantifier. The meaning of these constructors is best explained by the interpretation function, which defines the membership predicate for each set comprehension:

```
Fixpoint SetComElem {ts} (C : Conf) (S : SetCom ts) : Prop :=
  match S with
    | BaseSet C' P   ⇒ C' = C ∧ P
    | ExSet _ _ e    ⇒ ∃ x, SetComElem C (e x)
  end .
```

For the base case we check whether the configuration equals the configuration in the set comprehension and whether the predicate is fulfilled. The second constructor is simply interpreted as existential quantification.

The type ConfSet extends set comprehensions with a union operator. The corresponding interpretation function ConfElem is straightforward:

```
Inductive ConfSet : Type :=
  | Sing {ts} : SetCom ts → ConfSet
  | Union     : ConfSet → ConfSet → ConfSet.
Fixpoint ConfElem (C : Conf) (S : ConfSet) : Prop :=
  match S with
    | Sing _ s      ⇒ SetComElem C s
    | Union S1 S2   ⇒ ConfElem C S1 ∨ ConfElem C S2
  end .
```

The equivalence of sets is then simply defined in terms of the above interpretation function, and the union and set comprehension notation is mapped to the constructors of ConfSet and SetCom:

```
Notation "S ≡ T" := (∀ x, ConfElem x S ↔ ConfElem x T)
                      (at level 80 , no associativity).
Infix "∪" := Union (at level 76 , left associativity).
Notation "{ x .. y , C | P }" :=
  (Sing (ExSet (fun x ⇒ ... (ExSet (fun y ⇒ BaseSet C P)) ...)))
  (at level 70 , x binder, y binder, no associativity) .
```

[6] Thus BaseSet may in fact represent the empty set if the predicate is false.

Using this setup, we can then formulate and prove the proof rules that we listed in Figure 3. However, some care has to be taken in formulating the proof rules in a way that they can be readily applied to a proof goal in a calculation proof. For instance the STEP proof rule is formulated as follows:

```
Theorem step : ∀ ts (S S' : SetCom ts) (T : ConfSet) ,
  (∀ x, getProp S' x → getConf S x ⟹ getConf S' x) →
  (∀ x, getProp S' x → getProp S x) →
  (∀ x, getProp S x → getConf S x ◁ Sing S' ∪ T) →
  Sing S ∪ T ⇛ Sing S' ∪ T .
```

In the above theorem, we use getProp and getConf to extract the configuration and the predicate of a given set comprehension. We define getConf as follows:

```
Fixpoint getConf {ts} (S : SetCom ts) : tuple ts → Conf :=
  match S with
    | BaseSet C P ⇒ fun xs ⇒ C
    | ExSet _ _ ex ⇒ fun xs ⇒ let (x, xs') := xs in getConf (ex x) xs'
  end .
```

The function getProp is defined analogously. The definition uses the type tuple ts, which is a nested product type of all types in the list ts defined as follows:

```
Fixpoint tuple (ts : list Type) : Type :=
  match ts with
    | []       ⇒ unit
    | t :: ts' ⇒ t * tuple ts'
  end .
```

We would not have been able to define getConf and getProp, if we had represented configuration sets using simply the type Conf → Prop. The ability to define these functions is crucial in order to formulate the STEP rule in such a way that the Coq system can readily apply it to a given proof goal.

6.2 Calculation Syntax and Proof Automation

The calculation syntax is quite easy to achieve using Coq's Tactic Notation command to define custom tactics. The implementation of our calculation syntax closely follows the work by Tesson et al. [12]. But we use a somewhat simpler setup. The details can be found in the accompanying Coq source code.

In our pen-and-paper proofs (e.g. in Figure 4) we use the notations "⇚=", "≡", and "⟸" to indicate the proof rules that we use. This notation is reflected in the Coq proofs (cf. Figure 5), where we have the corresponding tactic notations "<|= { t } s", "= { t } s", and "<== { t } s", which refer to a tactic t and a configuration set s.

Given a proof goal of the form T ⇛ U, all three tactic notations try to prove the proof goal S ⇛ U. If successful, the original proof goal T ⇛ U is

replaced by $\mathsf{T} \Rightarrow \mathsf{S}$ using the TRANS proof rule. Once we have transformed the proof goal into the form $\mathsf{S} \Rightarrow \mathsf{S}$, we can use the tactic [], which applies the REFL proof rule to complete the calculation proof.

The basic tactic <|= { t } s does little proof automation: it tries to prove the goal $\mathsf{S} \Rightarrow \mathsf{U}$ using tactic t using the UNION and REFL proof rule, which allows us to prove goals of the form

$$\mathsf{S_1} \cup \ldots \cup \mathsf{S_i} \cup \ldots \mathsf{S_n} \Rightarrow \mathsf{S_1} \cup \ldots \cup \mathsf{S'_i} \cup \ldots \mathsf{S_n}$$

using tactic t to prove $\mathsf{S_i} \Rightarrow \mathsf{S'_i}$.

The tactic = { t } s is even simpler: it tries to prove $\mathsf{S} \Rightarrow \mathsf{U}$ using the IFF proof rule, and it uses t to prove $\mathsf{S} \equiv \mathsf{U}$.

Finally, <== { t } s applies the proof rule STEP to prove $\mathsf{S} \Rightarrow \mathsf{U}$. That is, the proof goal has to be of the shape

$$\{\overline{\mathsf{x}},\ \mathsf{C} \mid \mathsf{P}\} \cup \mathsf{S'} \Rightarrow \{\overline{\mathsf{x}},\ \mathsf{C'} \mid \mathsf{P}\} \cup \mathsf{S'}.$$

The tactic t is then used to discharge the proof obligation for $\forall\, \overline{\mathsf{x}},\ \mathsf{P} \rightarrow \mathsf{C} \Longrightarrow \mathsf{C'}$, whereas it tries to prove the side condition $\forall\, \overline{\mathsf{x}},\ \mathsf{P} \rightarrow \mathsf{C} \lhd \{\overline{\mathsf{x}},\ \mathsf{C'} \mid \mathsf{P}\} \cup \mathsf{S'}$ fully generically. It does so by applying the two rules HERE-$\lhd$ and STEP-$\lhd$ that define $\lhd$ (cf. section 3). Successfully applying STEP-$\lhd$ and then HERE-$\lhd$ to all subsequent subgoals, means that all single step $\Longrightarrow$-derivations from C reach a configuration in $\{\overline{\mathsf{x}},\ \mathsf{C'} \mid \mathsf{P}\} \cup \mathsf{S'}$. This combination proves most barred side conditions and is thus tried first. If that fails, our tactic tries to prove the side condition by systematically trying all $\overset{*}{\Longrightarrow}$-derivations of a bounded length.

7 Concluding Remarks

We presented a framework for deriving correct-by-construction compilers from formal specifications by means of calculations. The distinguishing feature of our calculation framework is that it accommodates non-deterministic semantics. The key ingredient of this framework is the set of proof rules that allows us to prove the compiler correctness property in one go despite the non-deterministic semantics. The mechanisation of the framework in Coq helps to scale our approach to more intricate languages. Moreover, this mechanisation allows us to utilise proof search to discharge side conditions that are subject to the changing virtual machine semantics.

We conclude this paper with a brief discussion on related work, limitations of our approach, and possible further work.

Related work To the best of our knowledge, non-deterministic languages have not been considered in the literature on calculating compilers. Meijer [10] does consider a language with backtracking semantics, called $\mathcal{B}$, but strictly speaking it is not a non-deterministic language. The language $\mathcal{B}$ has a set-valued semantics that describes the search space, which the language is able to traverse. However, non-deterministic languages can be represented using such a set-valued semantics

as well, and we believe Meijer's approach can be used to calculate a compiler for a non-deterministic language. Unfortunately, this setup requires a detour: from the big-step semantics of the target language we need to calculate an equivalent set-valued semantics. From that semantics we calculate a compiler and a virtual machine, which is a tail recursive function on sets of machine configurations. From this representation of the virtual machine a small-step semantics representation can be calculated. Bahr and Hutton [4] describe this technique and argue that it can be used to extend their calculation approach to non-deterministic languages. In contrast, the calculation framework presented in this paper allows us to calculate a compiler and a virtual machine directly without pre-processing the input semantics or post-processing the output semantics.

We also briefly remark on the importance of the antecedent $\exists\ \mathsf{D},\ \mathsf{C} \implies \mathsf{D}$ of the Step-$\lhd$ rule, which is missing in the corresponding definition by Hutton and Wright [7]. If we removed it, we would be able to prove "correctness" of compilers that are in fact not correct. For example, assume that we extended the definition of the virtual machine from section 2 by the following rule

$$\langle \mathsf{PUSH\ n} :: \mathsf{c},\ \mathsf{s} \rangle \implies \langle [],\ [42] \rangle$$

Then the calculation in Figure 4 would still be valid even though the compiler is not correct for this virtual machine. The virtual machine admits the following single step execution, which computes the result 42 for the expression Val 0:

$$\langle \mathsf{comp}\ (\mathsf{Val}\ 0),\ [] \rangle \ = \ \langle [\mathsf{PUSH}\ 0],\ [] \rangle \implies \langle [],\ [42] \rangle$$

The underlying problem is that the definition of $\lhd$ by Hutton and Wright [7] does not satisfy Proposition 1. In particular, their definition allows us to prove that $\langle \mathsf{comp}\ (\mathsf{Val}\ 0),\ [] \rangle$ is barred by the singleton set $\{ \langle [],\ [0] \rangle \}$. However, we cannot extend the above execution of the virtual machine such that it ends in the configuration $\langle [],\ [0] \rangle$, which contradicts Proposition 1.

Our implementation of the calculation framework in Coq is derived from the work of Tesson et al. [12]. A similar framework for writing calculation proofs in Agda has been developed by Mu et al. [11]. However, since our approach relies on proof automation, we prefer the Coq system over Agda.

Future work The calculation proofs that we presented here proceed by induction on the structure of the source language. This may be a problem if the semantics is not given in a compositional manner. For instance, in their treatment of lambda calculi Bahr and Hutton [4] start with a compositional semantics, but then transform it using defunctionalisation. The resulting big-step operational semantics is not compositional anymore and the calculation proof proceeds by induction on the big-step operational semantics.

The use of induction on the source language (as opposed to induction on the semantics) appears to be unavoidable for proving the soundness property for a non-deterministic language: The very goal of proving soundness is to prove that $\mathsf{e} \Downarrow \mathsf{n}$ holds, given that n is a possible result that the compiled program yields. Hence, we cannot perform a proof by induction on $\mathsf{e} \Downarrow \mathsf{n}$.

Another computational feature worth considering is concurrency. We did consider a language with interrupts – arguably a concurrency feature, albeit a simple one. However, if we want to deal with "proper" concurrency features, the calculation framework needs to be able to deal with a semantics for the source language that is able to properly capture concurrent behaviour, e.g. small-step operational semantics. Moreover, in a setting of concurrency we need to be able to reason not only about the result of a computation but also its I/O behaviour.

References

[1] Ager, M.S., Biernacki, D., Danvy, O., Midtgaard., J.: From Interpreter to Compiler and Virtual Machine: A Functional Derivation. Technical Report RS-03-14, Department of Computer Science, University of Aarhus (2003)

[2] Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming. pp. 8–19 (2003)

[3] Backhouse, R.: Program Construction: Calculating Implementations from Specifications. John Wiley and Sons, Inc. (2003)

[4] Bahr, P., Hutton, G.: Calculating correct compilers (Jul 2014), submitted to J. Funct. Program.

[5] Danvy, O., Millikin, K.: On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. Inf. Process. Lett. 106(3), 100 – 109 (2008)

[6] Hutton, G.: Programming in Haskell, vol. 2. Cambridge University Press Cambridge (2007)

[7] Hutton, G., Wright, J.: What is the meaning of these constant interruptions? J. Funct. Program. 17(06), 777–792 (2007)

[8] Kahn, G.: Natural semantics. In: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science. pp. 22–39 (1987)

[9] Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 42–54 (2006)

[10] Meijer, E.: Calculating Compilers. Ph.D. thesis, Katholieke Universiteit Nijmegen (1992)

[11] Mu, S.C., Ko, H.S., Jansson, P.: Algebra of programming in Agda: Dependent types for relational program derivation. J. Funct. Program. 19, 545–579 (2009)

[12] Tesson, J., Hashimoto, H., Hu, Z., Loulergue, F., Takeichi, M.: Program calculation in Coq. In: Algebraic Methodology and Software Technology. LNCS, vol. 6486, pp. 163–179 (2011), Revised Selected Papers

[13] Troelstra, A.S., van Dalen, D.: Constructivism in Mathematics: An Introduction, vol. 1. Elsevier (1988)

[14] Wand, M.: Deriving Target Code as a Representation of Continuation Semantics. ACM Trans. Program. Lang. Syst. 4(3), 496–517 (1982)