Functional Big-step Semantics

Scott Owens[1], Magnus O. Myreen[2], Ramana Kumar[3], and Yong Kiam Tan[4]

[1] School of Computing, University of Kent, UK
[2] CSE Department, Chalmers University of Technology, Sweden
[3] NICTA, Australia
[4] IHPC, A*STAR, Singapore

Abstract. When doing an interactive proof about a piece of software, it is important that the underlying programming language's semantics does not make the proof unnecessarily difficult or unwieldy. Both small-step and big-step semantics are commonly used, and the latter is typically given by an inductively defined relation. In this paper, we consider an alternative: using a recursive function akin to an interpreter for the language. The advantages include a better induction theorem, less duplication, accessibility to ordinary functional programmers, and the ease of doing symbolic simulation in proofs via rewriting. We believe that this style of semantics is well suited for compiler verification, including proofs of divergence preservation. We do not claim the invention of this style of semantics: our contribution here is to clarify its value, and to explain how it supports several language features that might appear to require a relational or small-step approach. We illustrate the technique on a simple imperative language with C-like for-loops and a break statement, and compare it to a variety of other approaches. We also provide ML and lambda-calculus based examples to illustrate its generality.

1 Introduction

In the setting of mechanised proof about programming languages, it is often unclear what kind of operational semantics to use for formalising the language: common big-step and small-step approaches each have their own strengths and weaknesses. The choice depends on the size, complexity, and nature of the programming language, as well as what is being proved about it. As a rule-of-thumb, the more complex the language's features, or the more semantically intricate the desired theorem, the more likely it is that small-step semantics will be needed. This is because small-step semantics enable powerful proof techniques, including syntactic preservation/progress and step-indexed logical relations, by allowing close observation not only of the result of a program, but also how it got there. In contrast, big-step's advantages arise from following the syntactic structure of the programming language. This means that they can mesh nicely with the similarly structured compilers, type systems, etc. that one is trying to verify, and reduce the overhead of mechanised proof.

For large projects, a hybrid approach can be adopted. The CompCert [16,17] verified C compiler uses big-step for some parts of its semantics and small-step for others. In our own CakeML project [15], we have two different semantics for the source language: big-step for the compiler verification and small-step for the type soundness proof, with an additional proof connecting the two semantics.

In contrast, this paper advocates *functional big-step semantics*, which can support many of the proofs and languages that typically rely on a small-step approach, but with a structure that follows the language's syntax. A functional big-step semantics is essentially an interpreter written in a purely functional style and equipped with a clock to ensure that the function is total, even when run on diverging programs. Hence the interpreter can be used in a higher-order logic of total functions – the kind supported by Coq, HOL4, and Isabelle/HOL – as a formal definition of the semantics. In this way, it harkens back to Reynolds' idea of definitional interpreters [23] to give a readable account of a semantics.

The idea of using a clock in a semantics not new;[1] our contribution here is to analyse its advantages, especially in the context of interactive proofs:

- Functional semantics are easier to read, have a familiar feel for functional programmers, and avoid much of the duplication that occurs in big-step semantics defined with inductive relations, especially for languages with exceptions and other non-local control-flow (§2).
- Functional semantics can be used more easily in mechanised proofs based on rewriting, since functional semantics are stated in terms of equations (§3.1).
- Functional semantics also produce better induction theorems. Induction theorems for relational big-step semantics frequently force unnecessary case splits in proofs (§3.2).
- Functional semantics naturally support the kinds of proofs that push researchers to small-step semantics, including divergence preservation for compilers: the clock that is used when defining functional semantics is a convenient tool in proofs of divergence preservation (§3.3, §3.4). Other examples are supporting languages with I/O and non-determinism (§4), proving type soundness (§5), and defining (and using) step-indexed logical relations (§6).

There are a variety of advanced techniques for defining big-step semantics that solve some of these problems. For example, one can use co-induction to precisely define diverging computations [18], or the pretty-big-step approach to reduce duplication in the definition [10]. Notably, these techniques still define the semantics using inductive (and co-inductive) relations rather than recursive functions, and we are not aware of any relational approach with all of the advantages listed above. However, functional semantics, as advocated in this paper, are not without their limitations. One is that the definition of a functional semantics requires introduction of a clock which must decrease on *certain* recursive calls (§2.3). Another is that languages with non-determinism require an oracle state component to factor out the non-determinism (§4). Lastly, we have not investigated languages with unstructured non-determinism, e.g. concurrency.

[1] For example, CakeML uses a clocked, but relational, semantics for its intermediate languages, and clocked recursive evaluation functions are common in Boyer-Moore-style provers such as ACL2, where inductive relations are unavailable [8,29]. Siek has also advocated for clocks for proving type soundness [24,25]

The ideas for this work were developed in the context of the CakeML project (`https://cakeml.org`) where we use functional big-step semantics for many of our intermediate languages, and will eventually transition all of our semantics to it. However, this paper concentrates on a series of simpler examples, starting with a C-like language with `for` and `break` statements (§2). We chose this language to facilitate a comparison (§8) with Charguéraud's presentation of the pretty-big step approach to relational semantics [10]. We use it to explain in detail how the functional approach supports the verification of a simple compiler (§3). Then, we present a series of different languages and theorems to illustrate the breadth of our approach (§4, §5, and §6). Lastly, we show how to prove the equivalence of a functional big-step and small-step semantics (§7).

All of the semantics in the paper have been formalised, and all of the theorems proved, in the HOL4 proof assistant (`http://hol-theorem-prover.org`). The formalisation is available in the HOL4 examples directory (`https://github.com/HOL-Theorem-Prover/HOL/tree/master/examples/fun-op-sem`); we encourage the interested reader to consult these sources for the definitions and lemmas that we lack the space to present here.

2 Example semantics

In this section, we motivate functional big-step semantics by defining an operational semantics for a toy language in both relational and functional styles. We call our toy language FOR, as it includes `for` loops and `break` statements that are familiar from C. We first define the big-step semantics of FOR, informally, as an interpreter in Standard ML (SML); next we explain why the semantics of FOR is difficult to capture in a conventional big-step relation, but, using a functional big-step semantics, can be defined neatly as a function in logic. The functional big-step semantics for FOR is essentially an adaption of the original SML interpreter.

2.1 An interpreter in SML

The FOR language has expressions `e` and statements `t`. Like C, we allow expression evaluation to have side effects (namely, assignment).

```
datatype e = Var of string
           | Num of int
           | Add of e * e
           | Assign of string * e

datatype t = Dec of string * t
           | Exp of e
           | Break
           | Seq of t * t
           | If of e * t * t
           | For of e * e * t
```

We sketch the semantics for this language by defining functions that evaluate expressions and statements, `run_e` and `run_t` respectively. Each evaluation returns an integer wrapped in `Rval`, signals a break `Rbreak`, or fails `Rfail`.

```
datatype r = Rval of int | Rbreak | Rfail
```

Expression evaluation fails on an attempt to read the value of an uninitialised variable.

```
fun lookup y [] = NONE
  | lookup y ((x,v)::xs) = if y = x then SOME v else lookup y xs

fun run_e s (Var x) =
     (case lookup x s of
        NONE => (Rfail,s)
      | SOME v => (Rval v,s))
  | run_e s (Num i) = (Rval i,s)
  | run_e s (Add (e1, e2)) =
     (case run_e s e1 of
        (Rval n1, s1) =>
           (case run_e s1 e2 of
              (Rval n2, s2) => (Rval (n1+n2), s2)
            | r => r)
      | r => r)
  | run_e s (Assign (x, e)) =
     (case run_e s e of
        (Rval n1, s1) => (Rval n1, (x,n1)::s1)
      | r => r)
```

Below, evaluation of a `Break` statement returns `Rbreak`, which is propagated to the enclosing `For` loop. A `For` loop returns a normal `Rval` result if the body of the loop returns `Rbreak`.

```
fun run_t s (Exp e) = run_e s e
  | run_t s (Dec (x, t)) = run_t ((x,0)::s) t
  | run_t s Break = (Rbreak, s)
  | run_t s (Seq (t1, t2)) =
     (case run_t s t1 of
        (Rval _, s1) => run_t s1 t2
      | r => r)
  | run_t s (If (e, t1, t2)) =
     (case run_e s e of
        (Rval n1, s1) => run_t s1 (if n1 = 0 then t2 else t1)
      | r => r)
  | run_t s (For (e1, e2, t)) =
     (case run_e s e1 of
        (Rval n1, s1) =>
          if n1 = 0 then (Rval 0, s1) else
           (case run_t s1 t of
              (Rval _, s2) =>
                (case run_e s2 e2 of
```

```
            (Rval _, s3) => run_t s3 (For (e1, e2, t))
          | r => r)
     | (Rbreak, s2) => (Rval 0, s2)
     | r => r)
| r => r)
```

Note that these SML functions make use of catch-all patterns in case-expressions in order to conveniently propagate non-`Rval` results. We use the same approach in our functional semantics (§2.3) to keep them concise.

2.2 Relational big-step semantics

The definition above is probably a good way to describe the semantics of FOR to a programmer familiar with SML. It is, however, not directly usable as an operational semantics for interactive proofs. Next, we outline how a big-step semantics can be defined for the FOR language using conventional inductively defined relations.

Relational big-step semantics are built up from evaluation rules for an evaluation relation, typically written $\Downarrow$. Each rule states how execution of a program expression evaluates to a result. The evaluation relation for the FOR language takes as input a state and a statement; it then relates these inputs to the result pair (`r` and new state) just as the interpreter above does.

We give a flavour of the evaluation rules next. The simplest rule in the FOR language is evaluation of `Break`: evaluation always produces `Rbreak` and the state is returned unchanged. We call this rule (B).

$$\text{(B)}\quad \frac{}{(\texttt{Break}, s) \Downarrow_{\texttt{t}} (\texttt{Rbreak}, s)}$$

The semantics of `Seq` is defined by two evaluation rules. We need two rules because evaluation of t_2 only happens if evaluation of t_1 leads to `Rval`. The first rule for `Seq` (S1) states: if t_1 evaluates according to $(t_1, s) \Downarrow_{\texttt{t}} (\texttt{Rval}\ n_1, s_1)$ and t_2 evaluates as $(t_2, s_1) \Downarrow_{\texttt{t}} r$, then $(\texttt{Seq}\ t_1\ t_2, s) \Downarrow_{\texttt{t}} r$, i.e. $\texttt{Seq}\ t_1\ t_2$ evaluates state s to result r. The second rule (S2) states that a non-`Rval` result in t_1 is the result for evaluation of $\texttt{Seq}\ t_1\ t_2$.

$$\text{(S1)}\quad \frac{(t_1, s) \Downarrow_{\texttt{t}} (\texttt{Rval}\ n_1, s_1) \qquad (t_2, s_1) \Downarrow_{\texttt{t}} r}{(\texttt{Seq}\ t_1\ t_2, s) \Downarrow_{\texttt{t}} r} \qquad \text{(S2)}\quad \frac{(t_1, s) \Downarrow_{\texttt{t}} (r, s_1) \qquad \neg\texttt{is\_Rval}\ r}{(\texttt{Seq}\ t_1\ t_2, s) \Downarrow_{\texttt{t}} (r, s_1)}$$

Defining these evaluation rules is straightforward, if the language is simple enough. We include the `For` statement in our example language in order to show how this conventional approach to big-step evaluation rules becomes awkward and repetitive. The `For` statement's semantics is defined by six rules. The first rule captures the case when the loop is not executed, i.e. when the guard expression evaluates to zero. The second rule states that errors in the evaluation of the guard are propagated.

$$\text{(F1)}\quad \frac{(e_1,s)\Downarrow_{\mathsf{e}} (\texttt{Rval}\ 0,s_1)}{(\texttt{For}\ e_1\ e_2\ t,s)\Downarrow_{\mathsf{t}} (\texttt{Rval}\ 0,s_1)} \qquad \text{(F2)}\quad \frac{\begin{array}{c}(e_1,s)\Downarrow_{\mathsf{e}} (r,s_1)\\ \neg\texttt{is\_Rval}\ r\end{array}}{(\texttt{For}\ e_1\ e_2\ t,s)\Downarrow_{\mathsf{t}} (r,s_1)}$$

Execution of the body of the `For` statement is described by the following four rules. The first of the following rules (F3) specifies the behaviour of an evaluation where the guard e_1, the body t, and the increment expression e_2 each return some `Rval`. The second rule (F4) defines the semantics for the case where evaluation of the body t signals `Rbreak`. The third rule (F5) defines that errors that occur in evaluation of the body propagate. Similarly, the fourth rule (F6) states that errors in the increment expression e_2 propagate.

$$\text{(F3)}\quad \frac{\begin{array}{c}(e_1,s)\Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1,\ s_1)\\ n_1 \neq 0\\ (t,s_1)\Downarrow_{\mathsf{t}} (\texttt{Rval}\ n_2,s_2)\\ (e_2,s_2)\Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_3,s_3)\\ (\texttt{For}\ e_1\ e_2\ t,s_3)\Downarrow_{\mathsf{t}} r\end{array}}{(\texttt{For}\ e_1\ e_2\ t,s)\Downarrow_{\mathsf{t}} r} \qquad \text{(F4)}\quad \frac{\begin{array}{c}(e_1,s)\Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1,s_1)\\ n_1 \neq 0\\ (t,s_1)\Downarrow_{\mathsf{t}} (\texttt{Rbreak},s_2)\end{array}}{(\texttt{For}\ e_1\ e_2\ t,s)\Downarrow_{\mathsf{t}} (\texttt{Rval}\ 0,s_2)}$$

$$\text{(F5)}\quad \frac{\begin{array}{c}(e_1,s)\Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1,s_1)\\ n_1 \neq 0\\ (t,s_1)\Downarrow_{\mathsf{t}} (\texttt{Rval}\ n_2,s_2)\\ (e_2,s_2)\Downarrow_{\mathsf{e}} (r,s_3)\\ \neg\texttt{is\_Rval}\ r\end{array}}{(\texttt{For}\ e_1\ e_2\ t,s)\Downarrow_{\mathsf{t}} (r,\ s_3)} \qquad \text{(F6)}\quad \frac{\begin{array}{c}(e_1,s)\Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1,s_1)\\ n_1 \neq 0\\ (t,s_1)\Downarrow_{\mathsf{t}} (r,s_2)\\ \neg\texttt{is\_Rval}\ r\\ r \neq \texttt{Rbreak}\end{array}}{(\texttt{For}\ e_1\ e_2\ t,s)\Downarrow_{\mathsf{t}} (r,s_2)}$$

Clearly, the semantics can be defined in this way and, once one has become accustomed to it, these rules are quite easy to read. However, even an experienced semanticist finds it difficult to immediately see whether these rules cover all the cases. Maybe the last two rules above were surprising? Worse, these rules only provide semantics for terminating executions, i.e. if we want to reason about the behaviour of diverging evaluations, then these rules are not enough as stated (recall that this is an inductively defined relation).

Another drawback is the duplication that rules for complex languages (even for our toy FOR language) contain. In each of the four rules above, the first three lines are almost the same. This duplication might seem innocent. However, it has knock-on effects on interactive proofs: the generated induction theorem also contains duplication, and from there it leaks into proof scripts. In particular, users are forced to establish the same inductive hypothesis many times.

The rules (F2), (F5) and (F6) ensure that the `Rfail` value is always propagated to the top, preventing the big-step relation from doing the moral equivalent of getting 'stuck' in the small-step sense. Thus, we know that a program diverges iff it is not related to anything. We could omit these rules if we do not need or want to distinguish divergence from getting stuck, and this is often done with big-step semantics.[2] However, for the purposes of this paper, we are primarily interested in the (many) situations where the distinction is important – that is where the functional big-step approach has the largest benefit.

The 'not related' characterisation of divergence does not yield a useful principle for reasoning about diverging programs: the relation's induction principle only applies when a program is related to something, not when we know it is not related to anything. To define divergence with a relation [18], one adds to the existing evaluation relation $\Downarrow_{\texttt{t}}$ a co-inductively defined divergence relation $\Uparrow_{\texttt{t}}$, and gets back a useful co-induction principle.

The rules for `Seq` and `For` are given below. (S1') states that a sequence diverges if its first sub-statement does. (S2') says that the sequence diverges if the first sub-statement returns a value, using the $\Downarrow_{\texttt{t}}$ relation, and the second sub-statement diverges. Notice the duplication between the definitions of $\Downarrow_{\texttt{t}}$ and $\Uparrow_{\texttt{t}}$: both must allow the evaluation to progress normally up to a particular sub-statement, and then $\Downarrow_{\texttt{t}}$ requires it to terminate, and $\Uparrow_{\texttt{t}}$ requires it to diverge. This corresponds to the duplication internal to $\Downarrow_{\texttt{t}}$ for propagating `Rbreak` and other exceptional results.

$$\text{(S1')}\ \frac{(t_1, s)\ \Uparrow_{\texttt{t}}}{(\texttt{Seq}\ t_1\ t_2, s)\ \Uparrow_{\texttt{t}}} \qquad \text{(S2')}\ \frac{\begin{array}{c}(t_1, s)\ \Downarrow_{\texttt{t}}\ (\texttt{Rval}\ n_1, s_1)\\ (t_2, s_1)\ \Uparrow_{\texttt{t}}\end{array}}{(\texttt{Seq}\ t_1\ t_2, s)\ \Uparrow_{\texttt{t}}}$$

$$\text{(F1')}\ \frac{\begin{array}{c}(e_1, s)\ \Downarrow_{\texttt{e}}\ (\texttt{Rval}\ n_1, s_1)\\ n_1 \neq 0\\ (t, s_1)\ \Uparrow_{\texttt{t}}\end{array}}{(\texttt{For}\ e_1\ e_2\ t, s)\ \Uparrow_{\texttt{t}}} \qquad \text{(F2')}\ \frac{\begin{array}{c}(e_1,\ s)\ \Downarrow_{\texttt{e}}\ (\texttt{Rval}\ n_1,\ s_1)\\ n_1 \neq 0\\ (t,\ s_1)\ \Downarrow_{\texttt{t}}\ (\texttt{Rval}\ n_2,\ s_2)\\ (e_2,\ s_2)\ \Downarrow_{\texttt{e}}\ (\texttt{Rval}\ n_3,\ s_3)\\ (\texttt{For}\ e_1\ e_2\ t, s_3)\ \Uparrow_{\texttt{t}}\end{array}}{(\texttt{For}\ e_1\ e_2\ t,\ s)\ \Uparrow_{\texttt{t}}}$$

2.3 Functional big-step semantics

The interpreter written in SML, given in §2.1, avoids the irritating duplication of the conventional big-step semantics. It is also arguably easier to read and clearly gives some semantics to all cases, since the function is unconditional.

The obvious question now is: why not just take the SML code and define it as a function in logic? The answer is that the SML code does not terminate for all inputs, e.g., `run_t [] (For (Num 1, Num 1, Exp (Num 1)))`.

In order to define `run_t` as a function in logic, we need to make it total somehow. A technique for doing this is to add a clock to the function: on each recursive call, for which termination is non-obvious, one adds a clock decrement. The clock is a natural number, so when it hits zero, execution is aborted with a special time out signal.

[2] If we had another mode of failure, e.g., from a `raise` expression, then these rules would be needed to propagate that anyway.

A very simple implementation of the clocked-function solution is to add a check-and-decrement on every recursive call. The termination proof becomes trivial, but the function is cluttered with the clock mechanism.

Instead of inserting the clock on every recursive call, we suggest that the clock should only be decremented on recursive function calls for which the currently evaluated expressions does not decrease in size. For the FOR language, this means adding a clock-check-and-decrement only on the looping call in the `For` case. In the SML code, this recursive call is performed here:

```
| run_t s (For (e1, e2, t)) =
    ...
      (Rval _, s3) => run_t s3 (For (e1, e2, t))
```

In our functional big-step semantics for the FOR language, called `sem_t`, we write the line above as follows. Here `dec_clock` decrements the clock that is stored in the state.

```
sem_t s (For e1 e2 t) =
    ...
      (Rval n1,s3) ⇒
         if s3.clock ≠ 0 then
           sem_t (dec_clock s3) (For e1 e2 t)
         else (Rtimeout,s3)
```

All other parts of the SML code are directly translated from SML into logic. The complete definition of `sem_t` is given below. Here `sem_e` is the logic function equivalent of `run_e`, and `store_var` x 0 s is state s updated to have value 0 in variable x.

```
sem_t s (Exp e) = sem_e s e
sem_t s (Dec x t) = sem_t (store_var x 0 s) t
sem_t s Break = (Rbreak,s)
sem_t s (Seq t1 t2) =
case sem_t s t1 of
  (Rval _,s1) ⇒ sem_t s1 t2
| r ⇒ r)
sem_t s (If e t1 t2) =
case sem_e s e of
  (Rval n1,s1) ⇒ sem_t s1 (if n1 = 0 then t2 else t1)
| r ⇒ r)
sem_t s (For e1 e2 t) =
case sem_e s e1 of
  (Rval 0,s1) ⇒ (Rval 0,s1)
| (Rval _,s1) ⇒
    (case sem_t s1 t of
       (Rval _,s2) ⇒
         (case sem_e s2 e2 of
            (Rval _,s3) ⇒
              if s3.clock ≠ 0 then
                sem_t (dec_clock s3) (For e1 e2 t)
```

```
              else (Rtimeout,s₃)
           | r ⇒ r)
       | (Rbreak,s₂) ⇒ (Rval 0,s₂)
       | r ⇒ r)
   | r ⇒ r)
```

Note that, in our logic version of the semantics, we have introduced a new kind of return value called `Rtimeout`. This return value is used only to signal that the clock has aborted evaluation. It always propagates to the top, and can be used in reasoning about divergence preservation (§3.3).

Termination proof We prove termination for `sem_t` by providing a well-founded measure: the lexicographic ordering on the clock value and the size of the statement that is being evaluated. This measure works because the value of the clock is never increased, and, on every recursive call where the clock is not decremented, the size of the statement that is being evaluated decreases.[3]

No termination proof is required for relational big-step semantics. This requirement is, therefore, a drawback for the functional version. However, the functional representation brings some immediate benefits that are not immediate for relational definitions. The functional representation means that the semantics is total (by definition) and that the semantics is deterministic (see §4 for an account of non-deterministic languages). These are properties that can require tedious proof for relational definitions.

Semantics of terminating and non-terminating evaluations The `sem_t` function terminates for all inputs. However, at the same time, it gives semantics to both terminating and non-terminating (diverging) evaluations. We say that evaluation terminates, if *there exists* some initial value for the clock for which the `sem_t` returns `Rval`. An evaluation is non-terminating if `sem_t` returns `Rtimeout` *for all* initial values for the clock. In all other cases, the semantics fails. The top-level semantics is defined formally as follows. There are three observable outcomes: `Terminate`, `Diverge`, and `Crash`.

```
semantics t =
if ∃ c v s. sem_t (s_with_clock c) t = (Rval v,s) then Terminate
else if ∀ c. ∃ s. sem_t (s_with_clock c) t = (Rtimeout,s) then Diverge
else Crash
```

§3.3 verifies a compiler that preserves this semantics, and §4 extends the FOR language with input, output, and internal non-determinism.

3 Using functional semantics

The previous section showed how big-step semantics can be defined as functions in logic, and how they avoid the duplication that occurs in conventional big-step semantics. In this section, we highlight how the change in style of definition

[3] HOL4's current definition package requires some help to prove and use the fact that the clock never increases.

affects proofs that use the semantics. We compare proofs based on the functional semantics with equivalent proofs based on the relational semantics.

3.1 Rewriting with the semantics

Since the functional semantics is defined as a function, it can be used for evaluation in the logic and used directly for proofs by rewriting. As a simple example, we can easily show that the `Dec` statement is an abbreviation for a longer program. This proof is just simple a call to the automatic rewriter in HOL4.

$$\vdash \texttt{sem\_t}\ s\ (\texttt{Dec}\ v\ t) = \texttt{sem\_t}\ s\ (\texttt{Seq}\ (\texttt{Exp}\ (\texttt{Assign}\ v\ (\texttt{Num}\ 0)))\ t)$$

This ability to perform symbolic evaluation within the logic is a handy tool, as any ACL2 expert will attest [19].

Sometimes rewriting with a functional semantics can get stuck in an infinite loop. This happens when the left-hand side of the definition, e.g. in our example $\texttt{sem\_t}\ s\ (\texttt{For}\ e_1\ e_2\ t)$, matches a subexpression on the right-hand side of the equation, e.g. $\texttt{sem\_t}\ (\texttt{dec\_clock}\ s_3)\ (\texttt{For}\ e_1\ e_2\ t)$, and the rewriting engine does not detect that it is about to diverge (e.g., HOL4's simplifier does not). We use a simple work-around for this: we define $\texttt{STOP}\ x = x$ and prove an equation where the right-hand side is $\texttt{sem\_t}\ (\texttt{dec\_clock}\ s_3)\ (\texttt{STOP}\ (\texttt{For}\ e_1\ e_2\ t))$. We ensure that the automatic simplifier cannot remove `STOP` and thus cannot apply the rewrite beyond the potentially diverging recursive call.

Rewriting is possible but often more cumbersome with relational big-step semantics. In HOL4, every definition of an inductive relation produces a rewrite theorem of the following form. We only show the cases relating to `Seq`, eliding others with ellipses.

$$\begin{array}{l}
\vdash (t,s) \Downarrow_{\texttt{t}} res \iff \\
\quad \ldots \lor \ldots \lor \ldots \lor \\
\quad (\exists s_1\ t_1\ t_2\ n_1. \\
\qquad (t = \texttt{Seq}\ t_1\ t_2) \land (t_1,s) \Downarrow_{\texttt{t}} (\texttt{Rval}\ n_1,s_1) \land \\
\qquad (t_2,s_1) \Downarrow_{\texttt{t}} res) \lor \\
\quad (\exists s_1\ t_1\ t_2\ r. \\
\qquad (t = \texttt{Seq}\ t_1\ t_2) \land (res = (r,s_1)) \land (t_1,s) \Downarrow_{\texttt{t}} (r,s_1) \land \\
\qquad \neg\texttt{is_Rval}\ r) \lor \ldots \lor \ldots \lor \ldots \lor \ldots \lor \ldots \lor \ldots \lor \ldots \lor \ldots \lor \ldots
\end{array}$$

Such theorems have unrestricted left-hand sides, which easily cause HOL4's rewriter to diverge, and right-hand sides that introduce a large number of disjunctions. One can often avoid divergence by providing the rewriter with manually proved theorems with specialised left-hand sides, e.g. $(\texttt{Seq}\ t_1\ t_2,s) \Downarrow_{\texttt{t}} res$. Functional semantics require less work for use in proofs by rewriting.

3.2 Induction theorem

The ability to rewrite with the functional semantics helps improve the details of interactive proofs. Surprisingly, the use of functional semantics also improves the overall structure of many proofs. The reason for this is that the induction theorems produced by functional semantics avoid the duplication that comes from the relational semantics.

The induction theorems for the FOR language are shown in Figures 1 and 2. The induction theorem for `sem_t` only has one case for the `For` loop. In contrast, the induction theorem for the relational semantics has six cases for the `For` loop. The duplication in the relation semantics carries over to duplication in the induction theorem and, hence, to the structure of interactive proofs, making them longer and more repetitive. This difference is significant for languages with complex program constructs.

Avoiding duplication in relations The duplication problem can be avoided in relational big-step semantics. The trick is to define the evaluation rules such that program constructs are described by only one giant rule each. Below is an example of how one can package up all of the rules about `For` into one giant rule.

$$\frac{\begin{array}{l} (e_1, s) \Downarrow_{\mathtt{e}} (r_1, s_1) \wedge \\ (\textbf{if } (r_1 = \mathtt{Rval}\ n_1) \wedge n_1 \neq 0 \textbf{ then} \\ \quad (t, s_1) \Downarrow_{\mathtt{t}} (r_2, s_2) \wedge \\ \quad \textbf{if } r_2 = \mathtt{Rval}\ n_2 \textbf{ then} \\ \quad\quad (e_2, s_2) \Downarrow_{\mathtt{e}} (r_3, s_3) \wedge \\ \quad\quad \textbf{if } r_3 = \mathtt{Rval}\ n_3 \textbf{ then } (\mathtt{For}\ e_1\ e_2\ t, s_3) \Downarrow_{\mathtt{t}} \mathit{result} \\ \quad\quad \textbf{else } \mathit{result} = (r_3, s_3) \\ \quad \textbf{else } \mathit{result} = (r_2, s_2) \\ \ \textbf{else } (\mathit{result} = (r_1, s_1))) \end{array}}{(\mathtt{For}\ e_1\ e_2\ t, s) \Downarrow_{\mathtt{t}} \mathit{result}}$$

By avoiding the duplication in the rules, the induction theorem also avoids the duplication. Writing packaged rules, as shown above, is unusual and certainly not aesthetically pleasing. However, if relational definitions are to be used, we suspect that packaging evaluation rules in this way is potentially less intrusive to proofs than use of the pretty-big-step approach.[4]

3.3 Example compiler verification

Next, we outline how functional big-step semantics support compiler verification, proving that a simple compiler preserves the observable behaviour. Our compiler targets a simple assembly-like language, where the code is a list of instructions (`instr`).

```
instr = Add reg reg reg | Int reg int | Jmp num | JmpIf reg num
```

The compiler, `compile`, is a composition of three separate phases. The first phase, `phase1`, simplifies `For` and removes `Dec`; `phase2` splits assignments into

[4] Note that such packaged big-step rules are easy to define in HOL4. However, they do not fit well with Coq's default mechanism for defining inductive relations. Charguéraud's pretty-big-step approach was developed in the context of Coq.

```
⊢ (∀s e. P s (Exp e)) ∧
  (∀s x t. P (store_var x 0 s) t ⇒ P s (Dec x t)) ∧
  (∀s. P s Break) ∧
  (∀s t1 t2.
     (∀v2 s1 v5.
        (sem_t s t1 = (v2,s1)) ∧ (v2 = Rval v5) ⇒ P s1 t2) ∧
     P s t1 ⇒
     P s (Seq t1 t2)) ∧
  (∀s e t1 t2.
     (∀v2 s1 n1.
        (sem_e s e = (v2,s1)) ∧ (v2 = Rval n1) ⇒
        P s1 (if n1 = 0 then t2 else t1)) ⇒
     P s (If e t1 t2)) ∧
  (∀s e1 e2 t.
     (∀v2 s1 n1 v2' s2 n1' v2'' s3 n1''.
        (sem_e s e1 = (v2,s1)) ∧ (v2 = Rval n1) ∧ n1 ≠ 0 ∧
        (sem_t s1 t = (v2',s2)) ∧ (v2' = Rval n1') ∧
        (sem_e s2 e2 = (v2'',s3)) ∧ (v2'' = Rval n1'') ∧
        s3.clock ≠ 0 ⇒
        P (dec_clock s3) (For e1 e2 t)) ∧
     (∀v2 s1 n1.
        (sem_e s e1 = (v2,s1)) ∧ (v2 = Rval n1) ∧ n1 ≠ 0 ⇒
        P s1 t) ⇒
     P s (For e1 e2 t)) ⇒
  ∀v v1. P v v1
```

Fig. 1. Induction theorem for functional big-step semantics.

```
⊢ ... ∧ ... ∧ ... ∧ ... ∧ ... ∧ ... ∧ ... ∧ ... ∧
  (∀s s1 e1 e2 t.
     (e1,s) ⇓e (Rval 0,s1) ⇒ P (For e1 e2 t,s) (Rval 0,s1)) ∧
  (∀s s1 e1 e2 t r.
     (e1,s) ⇓e (r,s1) ∧ ¬is_Rval r ⇒ P (For e1 e2 t,s) (r,s1)) ∧
  (∀s s1 s2 s3 e1 e2 t n1 n2 n3 r.
     (e1,s) ⇓e (Rval n1,s1) ∧ n1 ≠ 0 ∧ P (t,s1) (Rval n2,s2) ∧
     (e2,s2) ⇓e (Rval n3,s3) ∧ P (For e1 e2 t,s3) r ⇒
     P (For e1 e2 t,s) r) ∧
  (∀s s1 s2 e1 e2 t n1.
     (e1,s) ⇓e (Rval n1,s1) ∧ n1 ≠ 0 ∧ P (t,s1) (Rbreak,s2) ⇒
     P (For e1 e2 t,s) (Rval 0,s2)) ∧
  (∀s s1 s2 s3 e1 e2 t n1 n2 r.
     (e1,s) ⇓e (Rval n1,s1) ∧ n1 ≠ 0 ∧ P (t,s1) (Rval n2,s2) ∧
     (e2,s2) ⇓e (r,s3) ∧ ¬is_Rval r ⇒
     P (For e1 e2 t,s) (r,s3)) ∧
  (∀s s1 s2 e1 e2 t n1 r.
     (e1,s) ⇓e (Rval n1,s1) ∧ n1 ≠ 0 ∧ P (t,s1) (r,s2) ∧ ¬is_Rval r ∧
     r ≠ Rbreak ⇒
     P (For e1 e2 t,s) (r,s2)) ⇒
  ∀ts rs. ts ⇓t rs ⇒ P ts rs
```

Fig. 2. Induction theorem for relational big-step semantics. Parts omitted with '…'.

simple instruction-like assignments to un-nest expressions, but stays within the source language; and phase3 reduces the remaining subset of the source language into a list of target instructions.

```
compile t = phase3 0 0 (phase2 (phase1 t))
```

The first phase is a source-to-source transformation. The following (easy to prove) theorem explains how For loops are simplified by this phase. Here Loop t = For (Num 1) (Num 1) t.

```
⊢ ∀ s b₁ b₂ t.
    sem_t s (For b₁ b₂ t) = sem_t s (Loop (If b₁ (Seq t (Exp b₂)) Break))
```

The compilation function phase1 traverses the code and replaces For and Dec. This compilation function has a very simple correctness theorem:

```
⊢ ∀ t s. sem_t s (phase1 t) = sem_t s t
```

We also prove that phase1 preserves the observable semantics:

```
⊢ ∀ t. semantics (phase1 t) = semantics t
```

Subsequent phases assume that For statements have been simplified to Loop. The verification of the second phase, phase2, is almost as simple but a little longer because phase2 invents variable names to hold temporary results.

The third phase compiles the resulting subset of the FOR language into a list of instructions in the assembly-like target language. The crucial lemma, stated below, was proved by induction using the theorem shown in Figure 1. This lemma's statement can informally be read as: if the source semantics sem_t dictates that program t successfully evaluates state s_1 to state s_2, the source program t is within the allowed syntactic subset, and the compiled code for t is installed in a store-related target state x; then the target semantics sem_a evaluates x to a new target state x' that is store-related to s_2. Below, sem_a is the functional big-step semantics for the target assembly language. The sem_a function executes one instruction at a time and is tail-recursive. Its lengthy definition is omitted. phase3_subset defines the syntactic restrictions that programs must follow after phases 1 and 2. The ellipses elide several detailed parts of the conclusion that are only necessary to make the induction go through: in particular, where the program counter will point at exit based on the result *res*.

```
⊢ ∀ s₁ t res s₂ x xs ys b.
    (sem_t s₁ t = (res,s₂)) ∧ phase3_subset t ∧ (x.store = s₁) ∧
    (x.pc = LENGTH xs) ∧
    (x.instrs = xs ++ phase3 (LENGTH xs) b t ++ ys) ∧ res ≠ Rfail ∧
    ((res = Rbreak) ⇒ LENGTH (xs ++ phase3 (LENGTH xs) b t) ≤ b) ⇒
    ∃ x'. (sem_a x = sem_a x') ∧ (x'.store = s₂) ∧ ...
```

From the lemma above, it is easy to prove that phase3 0 0 t preserves the observable semantics, if t is in the subset expected by the third phase and t does not Crash in the source semantics.

```
⊢ ∀t.
    semantics t ≠ Crash ∧ phase3_subset t ⇒
    (asm_semantics (phase3 0 0 t) = semantics t)
```

Here asm_semantics is the observable semantics of the target assembly language.

```
asm_semantics code =
if ∃c s. sem_a (a_state code c) = (Rval 0,s) then Terminate
else if ∀c. ∃s. sem_a (a_state code c) = (Rtimeout,s) then Diverge
else Crash
```

The following top-level compiler correctness theorem is produced by combining the semantics preservation theorems from all three phases. The assumption that the source semantics does not Crash is implied by a simple syntactic check syntax_ok, which checks that all variables been declared (Dec) and that all Break statements are contained within For loops.

```
⊢ ∀t. syntax_ok t ⇒ (asm_semantics (compile t) = semantics t)
```

3.4 Comparison with proof in relational semantics

We provide a corresponding proof of correctness for phase1 in the relational semantics. As a rough point of comparison, our relational proof required 192 lines while the functional big-step proof required 68. The proof is split into two parts, corresponding to the relations defining our big-step semantics:

```
⊢ ∀s t res. (t,s) ⇓t res ⇒ (phase1 t,s) ⇓t res
⊢ ∀s t. (t,s) ⇑t ⇒ (phase1 t,s) ⇑t
```

The advantage of (non-looping) functional rewriting is apparent in our proofs: we often had to manually control where rewrites were applied in the relational proof. Additionally, we had to deal with significantly more cases in the relational proofs; these extra cases came from two sources, namely, the ones arising from an additional co-inductive proof for diverging programs, and extra (similar) cases in the induction theorems.

The additional co-inductive proof is a good point of comparison, since our technique of decrementing the clock only on recursive calls in the functional big-step semantics gives us divergence preservation for free in compilation steps that do not cause additional clock ticks. The cases arising in our co-inductive proof also required a different form of reasoning from the inductive proof; this naturally arises from the difference between induction and co-induction but it meant that we could not directly adapt similar cases across both proofs.

The top-level observable semantics can be similarly defined for relational semantics:

```
rel_semantics t =
if ∃v s. (t,init_store) ⇓t (Rval v,s) then Terminate
else if (t,init_store) ⇑t then Diverge
else Crash
```

So we can prove the correctness of `phase1` with respect to `rel_semantics`:

$$\vdash \forall t.\ \texttt{rel\_semantics}\ t \neq \texttt{Crash} \Rightarrow (\texttt{rel\_semantics}\ (\texttt{phase1}\ t) = \texttt{rel\_semantics}\ t)$$

This proof requires that the relations ($\Downarrow_{\texttt{t}}$, $\Uparrow_{\texttt{t}}$) are disjoint:

$$\vdash \forall s\ t\ res.\ (t,s) \Downarrow_{\texttt{t}} res \Rightarrow \neg(t,s) \Uparrow_{\texttt{t}}$$

To further validate the functional big-step approach, we prove the equivalence of the functional big-step semantics (`sem_t`) and the relational semantics ($\Downarrow_{\texttt{t}}$, $\Uparrow_{\texttt{t}}$). (We also prove the equivalence with a small-step semantics in §7). The equivalences are described by the following theorems; the first two show equivalence for terminating programs while the final theorem shows equivalence on diverging programs.

$$\vdash \forall s\ t\ r\ s'.\ (\texttt{sem\_t}\ s\ t = (r,s')) \land r \neq \texttt{Rtimeout} \Rightarrow (t,s) \Downarrow_{\texttt{t}} (r,s'\ \texttt{with clock := } s\texttt{.clock})$$

$$\vdash \forall s\ t\ r\ s'.\ (t,s) \Downarrow_{\texttt{t}} (r,s') \Rightarrow \forall c'.\ \exists c.\ \texttt{sem\_t}\ (s\ \texttt{with clock := } c)\ t = (r,s'\ \texttt{with clock := } c')$$

$$\vdash \forall s\ t.\ (\forall c.\ \texttt{FST}\ (\texttt{sem\_t}\ (s\ \texttt{with clock := } c)\ t) = \texttt{Rtimeout}) \iff (t,s) \Uparrow_{\texttt{t}}$$

The proofs rely on the disjointness lemma above and a determinism lemma for the relational semantics:

$$\vdash \forall s\ t\ res.\ (t,s) \Downarrow_{\texttt{t}} res \Rightarrow \forall res'.\ (t,s) \Downarrow_{\texttt{t}} res' \Rightarrow (res = res')$$

They also rely on an analogue of determinism for the functional big-step semantics: if a program does not time out for a given clock, then every increment to the clock gives the same result[5].

$$\vdash \forall s\ t\ r\ s'.\ (\texttt{sem\_t}\ s\ t = (r,s')) \land r \neq \texttt{Rtimeout} \Rightarrow \forall k.\ \texttt{sem\_t}\ (s\ \texttt{with clock := } s\texttt{.clock + } k)\ t = (r,s'\ \texttt{with clock := } s'\texttt{.clock + } k)$$

These lemmas are easy to prove compared to the main body of the equivalence proof, and our examples above demonstrate that the number of such lemmas required is comparable between the two semantics.

[5] This lemma also implies that if a program times out for a given clock, then it times out for all smaller clocks.

4 Non-determinism

We now add non-deterministic evaluation order and input/output expressions to the FOR language. The only syntactic change is the addition of two expressions: `Getchar` and `Putchar` e. However, the observable behaviours of programs have changed significantly. Instead of doing exactly one of terminating, diverging, or crashing, a program can now exhibit a set of those behaviours. Furthermore, both termination and divergence results now include the I/O stream that the program consumed/produced. For technical reasons, it also contains the choices made by the non-deterministic evaluation order (see §7). In the type of `observation`, the `llist` type is the lazy list type that contains both finite and infinite lists, and `+` is the type constructor for disjoint unions.

```
observation =
    Terminate ((io_tag + bool) list)
  | Diverge ((io_tag + bool) llist)
  | Crash
```

As a function, `sem_t` seems to be inherently deterministic: we cannot simply have it internally know what the next input character is, or choose which subexpression to evaluate first. We are left with two options: we can factor out the input stream and all choices into the state argument of `sem_t` and then existentially quantify them in the top-level semantic function to build a set of results; or alternatively, we can change the type of `sem_t` to return sets of results (alongside partial I/O traces). Here we take the first approach which leads to only minor changes in the definition of `sem_t`.

First, the state argument of `sem_t` gets three new fields: `io_trace` to record the characters read and written; `input` to represent the (possibly infinite) input stream; and `non_det_o` which represents an infinite stream of decisions that determine the subexpression evaluation ordering. We include the inputs in the `io_trace` to accurately model the order in which the I/O operations happened.

```
io_tag = Itag int | Otag int
state =
  <| store : (string ↦ int);
     clock : num;
     io_trace : ((io_tag + bool) list);
     input : (char llist);
     non_det_o : (num -> bool) |>
```

Because all of our changes are limited to the expression language, and encapsulated in the extended state argument, which `sem_t` does not access, the definition of `sem_t` looks identical to the previous one. The changes to `sem_e` are limited to the `Add` case (where a non-deterministic choice is made), and two new cases for the new expressions.

```
sem_e s (Add e1 e2) =
(let ((fst_e,snd_e),nd_o,switch) = permute_pair s.non_det_o (e1,e2) in
   case
```

```
      sem_e
        (s with
         <|non_det_o := nd_o; io_trace := s.io_trace ++ [INR switch]|>)
        fst_e
    of
      (Rval fst_n,s1) ⇒
        (case sem_e s1 snd_e of
           (Rval snd_n,s2) ⇒
             (let (n1,n2) = unpermute_pair (fst_n,snd_n) switch in
                (Rval (n1 + n2),s2))
         | r ⇒ r
    | r ⇒ r

sem_e s Getchar =
(let (v,rest) = getchar s.input in
   (Rval v,
    s with <|input := rest; io_trace := s.io_trace ++ [INL (Itag v)]|>))

sem_e s (Putchar e) =
case sem_e s e of
  (Rval n1,s1) ⇒
    (Rval n1,s1 with io_trace := s1.io_trace ++ [INL (Otag n1)])
| r ⇒ r
```

The Add case is similar to before, but uses the permute_pair function to swap the sub-expressions or not, depending on the oracle. It also returns a new oracle ready to get the next choice, and whether or not it switched the sub-expressions. The latter is used to un-permute the values to apply the primitive + in the right order (which would matter for a non-commutative operator). Getchar similarly consumes one input and updates the state. Putchar adds to the I/O trace.

Critically, the above modifications are orthogonal to the clock, and do not affect the termination proof, or the usefulness of the induction theorems and rewriting equations. More substantial changes occur in the semantic function.

```
semantics t input (Terminate io_trace) ⇔
∃c nd i s.
  (sem_t (init_st c nd input) t = (Rval i,s)) ∧
  (FILTER ISL s.io_trace = io_trace)
semantics t input Crash ⇔
∃c nd r s.
  (sem_t (init_st c nd input) t = (r,s)) ∧ ((r = Rbreak) ∨ (r = Rfail))
semantics t input (Diverge io_trace) ⇔
∃nd.
  (∀c. ∃s. sem_t (init_st c nd input) t = (Rtimeout,s)) ∧
  (io_trace =
   ⋁c.
     fromList
       (FILTER ISL (SND (sem_t (init_st c nd input) t)).io_trace))
```

Firstly, `semantics` is now a predicate[6] over programs, inputs, and observation. Termination and crashing are still straightforward: the non-determinism oracle and input are quantified along with the clock, and the resulting I/O trace is read out of the result state. We filter the trace so it only contains the I/O actions and not the record of the non-determinism oracle. Some choices of oracles might lead to a crash whereas others might lead to different terminating results.

Divergence is more subtle. First, note that a program can both terminate and diverge. For example, in the following `x` can be assigned either `1` or `0`, depending on which sub-expression is evaluated first.

```
Seq (Exp (Add (Assign "x" 1) (Assign "x" 0)))
    (For (Var "x") (Num 1) (Exp (Num 1)))
```

Thus, in the definition of `semantics`, we first existentially quantify the non-determinism, then check that it results in a timeout for all clock values given that particular oracle. To ensure that the resulting I/O trace is correct, we consider the set of all I/O traces for every possible clock in the complete partial order of lazy lists ordered by the prefix relation. This set forms a chain, because we prove that increasing the input clock does not alter the I/O already performed. Hence, the resulting I/O behaviour is the least upper bound, which can be either a finite or infinite lazy list. Operationally, as we increase the clock, we potentially see more I/O behaviour, and the least upper bound defines the lazy list that incorporates all of these. (Notation: the $\bigvee$ binder takes lubs in this PO.)

Adapting the compiler verification Adapting the compiler verification to the I/O and non-determinism extension is an almost trivial exercise. The I/O streams were modelled in the same way in the assembly language, which we kept deterministic. The new proof engineering work stems mostly from the substantial change to the definition of the top-level semantics function `semantics`. Due to the internal non-determinism, which the compiler removes, the correctness theorem is now stated as a subset relation: every behaviour of the generated (deterministic) assembly code is also a behaviour of the (non-deterministic) source program.

$\vdash \forall t\ inp.$ `syntax_ok` $t \Rightarrow$ `asm_semantics (compile` t`)` $inp \subseteq$ `semantics` $t\ inp$

Relational big-step Non-determinism can be handled naturally with two big-step rules for `Add` (although that does introduce duplication), but a mixed co-inductive/inductive relation is required to define the I/O stream of a non-terminating program. We no longer have the option to equate divergence with a failure to relate the program to anything.

Concurrency The techniques described in this section can support functional big-step semantics for a large variety of practical languages, but they do share a significant limitation with other big-step approaches: concurrency. Concurrent execution would require interleaving the evaluation of multiple expressions,

[6] Note that HOL4 identifies the types α `-> bool` and α `set`.

whereas the main principle of a big-step semantics (ours included) is to evaluate an expression to a value in one step. Our non-determinism merely selects which to do first. Work-arounds, such as having `sem_t` return sets of traces of inter-thread communications, might sometimes be possible, but would have large effects on the shape of the definition of the semantics.

5 Type soundness

Whereas big-step semantics are common in compiler verification, small-step semantics enable the standard approach to type soundness by preservation and progress lemmas [28]. A type soundness theorem says that well-typed programs do not crash; they either terminate normally or diverge. As Siek notes [24], a critical thing a semantics must provide is a good separation between divergence and crashing, and a clocked big-step semantics does this naturally. We have experimented with two type systems and found that functional big-step semantics works very well for proving type soundness.

Our first type soundness example is for the FOR language. We prove that `syntax_ok` programs do not evaluate to `Rfail`. The key is to use the induction theorem associated with the functional semantics, rather than rule induction derived from the type system.

We carry the same approach to languages with more interesting type systems: our second type soundness example is for the Core ML language from Wright and Felleisen [28] with a functional big-step semantics that closely resembles an interpreter for ML. The type system is more complex than the FOR language's, supporting references, exceptions, higher-order functions and Hindley-Milner polymorphism. However, it turns out that this extra complexity in the type system factors out neatly, and does not disrupt the proof outline.

Our approach is similar to the one described by Siek [25], who uses a clocked functional big-step semantics and demonstrates the utility of the induction theorem arising from the clocked semantics. As a result, our main type soundness proof, which interacts with the big-step semantics, is easy. Siek's example type system is simpler than Core ML's: it has no references or polymorphism; but these difficult aspects can be isolated. The most difficult lemmas in our proof are about the type system, and rely on α-equivalence reasoning over type schemes. We follow Tofte [26] for these lemmas, but whereas Tofte goes on to use co-induction over an un-clocked big-step semantics, we can get by with induction on the clocked semantics.

The reason Tofte needed to use co-induction is that un-clocked big-step semantics do not distinguish between divergence and failure. Our statement of type soundness for Core ML is: if a program is well-typed, then for all clocks, the semantics of the program is either `Rtimeout` or an exception or value of the correct type, but never `Rfail`. The universal quantification of clocks makes this a strong statement, since it implies diverging well-typed programs also cannot fail. For contrast, we have also written un-clocked big-step semantics for Core ML and proved a similar theorem: if a program is well-typed and converges to r, then r is an exception or value of the correct type, but never Rfail. The proof by induction is essentially the same as for the clocked semantics, and all the type-system lemmas can be re-used exactly, but the conclusion is much weaker because diverging programs do not satisfy the assumption. Furthermore, the proof is longer (330 lines vs. 200) because of the duplication in the relational semantics.

6 Logical relations

The technique of step-indexed logical relations [2] supports reasoning about programs that have recursive types, higher-order state, or other features that introduce aspects of circularity into a language's semantics [1,12]. The soundness of these relations is usually proved with respect to a small-step semantics, because the length of a small-step trace can be used to make the relation well-founded when following the structure of the language's cyclic constructs (e.g., when following a pointer cycle in the heap or unfolding a recursive type). Here we show that the clock in a functional big-step semantics can serve the same purpose.

Because our main purpose here is to illustrate functional big-step semantics, we first present the relation and defer its motivation to the end of this section. For now, it suffices to say that it has some significant differences from the existing literature, because it is designed to validate compiler optimisations in an untyped setting.

We start with an untyped lambda calculus with literals, variables (using de Bruijn indices), functions, and a tick expression that decrements the clock. The semantics will also use closure values, and a state with a clock.

```
exp = Lit lit | Var num | App exp exp | Fun exp | Tick exp
v = Litv lit | Clos env exp
env = v list
state = <| clock : num; store : env |>
```

We can then define the function sem, which implements call-by-value and decrements the clock on every function call. EL gets the nth element of a list.

```
⊢ (sem env s (Lit i) = (Rval (Litv i),s)) ∧
  (sem env s (Var n) =
   if n < LENGTH env then (Rval (EL n env),s) else (Rfail,s)) ∧
  (sem env s (App e1 e2) =
   case sem env s e1 of
     (Rval v1,s) ⇒
       (case sem env s e2 of
          (Rval v2,s) ⇒
            if s.clock ≠ 0 then
              case v1 of
                Litv v4 ⇒ (Rfail,s)
              | Clos env' e ⇒ sem (v2::env') (dec_clock s) e
            else (Rtimeout,s)
        | r ⇒ r)
```

```
  | r ⇒ r)
 (sem env s (Fun e) = (Rval (Clos env e),s)) ∧
 (sem env s (Tick e) =
  if s.clock ≠ 0 then sem env (dec_clock s) e else (Rtimeout,s))
```

The top-level semantic function is defined in the same way as for the FOR language of §2.

We then define the relations `val_rel`, which relates two values; `exec_rel`, which relates two environment/store/expression triples (i.e., the inputs to `sem`); and `state_rel`, which relates two stores; all at a given index.

```
val_rel i (Litv l) (Litv l′) ⟺ (l = l′)
val_rel i (Clos env e) (Clos env′ e′) ⟺
  ∀ i′ a a′ s s′.
    i′ < i ⇒
    state_rel i′ s s′ ∧ val_rel i′ a a′ ⇒
    exec_rel i′ (a::env,s,e) (a′::env′,s′,e′)
val_rel i (Litv l) (Clos env e) ⟺ F
val_rel i (Clos env e) (Litv l) ⟺ F
exec_rel i (env,s,e) (env′,s′,e′) ⟺
  ∀ i′. i′ ≤ i ⇒
    (let (res₁,s₁) = sem env (s with clock := i′) e in
     let (res₂,s₂) = sem env′ (s′ with clock := i′) e′ in
       case (res₁,res₂) of
         (Rval v₁,Rval v₂) ⇒
           (s₁.clock = s₂.clock) ∧ state_rel s₁.clock s₁ s₂ ∧
           val_rel s₁.clock v₁ v₂
       | (Rtimeout,Rtimeout) ⇒ state_rel s₁.clock s₁ s₂)
       | (Rfail,_)) ⇒ T
       | r ⇒ F
state_rel i s s′ ⟺
  LIST_REL (λ a′ a. val_rel i a′ a) s.store s′.store
```

The definitions of `val_rel` and `state_rel` are typical of a logical relation; `exec_rel` is where the relation interacts with the functional big-step semantics. In the small-step setting, `exec_rel` would say that the two triples are related if they remain related for i steps of the small-step semantics. With the functional big-step semantics, we instead check that the results of the `sem` function are related when we set the clock to a value less than i.

From here we prove that the relation is compatible with the language's syntax, that it is reflexive and transitive, that it is sound with respect to contextual approximation, and finally that β-value conversion is in the relation, and hence a sound optimisation for the language at any subexpression. Most of the proof is related to the semantic work at hand, rather than the details of the semantics, but we do need to rely on several easy-to-prove lemmas about the clock that capture intuitive aspects of what it means to be a clocked evaluation function. They correspond to the last lemma from §3.4.

Motivation The language and relation are designed as a mock-up of an under-development intermediate language for CakeML that is similar to the *clambda* intermediate language in the OCaml compiler [9]. Because this is an untyped intermediate language for a typed source language, the compiler should be able to change a failing expression into anything at all. We know that we will never try to compile an expression that fails, and this design allows us to omit run-time checks that would otherwise be needed to signal failure. This is why `exec_rel` relates `Rfail` to anything, and why our relation is not an equivalence, but an approximation: the compiler must never convert a good expression into one that fails.

Furthermore, the compiler must not convert a diverging program into one that converges (or vice-versa). This is why `Rtimeout` is only related to itself, and why the clocks are both set to the same i' when running the expressions. In a typed setting, the clock for the right-hand argument is existentially quantified, thereby allowing a diverging expression to be related to a converging one, and if one wants to show equivalence, one proves the approximation both ways. Because of our treatment of failure, that is not an option here. The drawback is that we cannot support transformations that increase the number of clock ticks needed. For transformations that might reduce the number of ticks, including our β-value conversion, the transformation just needs to introduce extra `Tick` instructions.

All of the above applies in a small-step setting too. However, the functional big-step approach does have some flexibility for changing the amount of computation done. For example, both $1 + 2$ and 3 evaluate with the same clock, and so this type of logical relation could be used to show that constant folding is a sound optimisation. However, they do not have the same small-step trace length, and so a constant folding optimisation would have to introduce extra `Tick`-like instructions to introduce dummy steps in the small-step setting.

7 Equivalence with small-step semantics

We build a straightforward small-step semantics for the FOR language by adding a `Handle` statement to the language, to stop the propagation of `Break` statements upward, and implement `For` as follows (we write `Seq` as an infix `;`):

$$\texttt{(For } e_1\ e_2\ t\texttt{, } s\texttt{)} \to_{\texttt{t}} \texttt{(Handle (If } e_1\ \texttt{(}t\texttt{;Exp } e_2\texttt{;For } e_1\ e_2\ t\texttt{) (Exp (Num 0))),}s\texttt{)}$$

To prove the equivalence of the functional big-step and small-step, we need two lemmas. First, that the functional semantics only gives `Rtimeout` with a clock of 0 (which is trivial to prove). Second, that any result of the functional semantics has a corresponding trace through the small-step semantics that is long enough. In the theorem below, we represent the small-step trace with a list so that we can check its length. The `check_trace` predicate checks that it is indeed a trace of $\to_{\texttt{t}}$ steps. The length check ensures that if the functional big-step diverges, then we will be able to build a small-step trace of arbitrary length, and so it diverges too. The subtraction calculates how many clock ticks the evaluation actually used.

$$
\begin{array}{l}
\vdash (\texttt{sem\_t}\ s\ t = r) \Rightarrow \\
\quad \exists\, tr. \\
\qquad tr \neq [\,] \wedge s.\texttt{clock} - (\texttt{SND}\ r).\texttt{clock} \leq \texttt{LENGTH}\ tr \wedge \\
\qquad \texttt{check\_trace}\ (\lambda\, st.\ \texttt{some}\ st'.\ st \rightarrow_{\mathtt{t}} st')\ tr \wedge \\
\qquad (\texttt{HD}\ tr = (s.\texttt{store},\texttt{t\_to\_small\_t}\ t)) \wedge \texttt{res\_rel\_t}\ r\ (\texttt{LAST}\ tr)
\end{array}
$$

One would expect a theorem like this to show up in any big-step/small-step equivalence proof. The extra length check adds very little difficulty to the proof, but ensures that we do not need to explicitly prove anything about divergence, or additionally reason going from small-step traces to big-step results. Similar to type soundness (§5), we prove this using the induction principle of `sem_t`.

In the non-deterministic case, we extend the state of the small-step semantics with the same oracle that the functional big-step semantics uses, and we use the oracle to choose which sub-expression of an `Add` to start evaluating. `AddL` and `AddR` expressions are included to mark which argument is being evaluated, so that we do not consult the oracle in subsequent steps for the same decision or switch back-and-forth between subexpressions. For example, if the oracle returns false, we start evaluating the left sub-expression on the updated oracle state. The `oracle_upd` function puts the new oracle into s and adds `F` to its *io_trace*.

$$
\frac{\texttt{oracle\_get}\ s.\texttt{non\_det\_o} = (\texttt{F},o')}{(\texttt{Add}\ e_1\ e_2,\ s) \rightarrow_{\mathtt{e}} (\texttt{AddL}\ e_1\ e_2,\texttt{oracle\_upd}\ s\ (\texttt{F},o'))}
$$

Thus, the small-step semantics remains non-deterministic, and we can use the same approach as above. There are three significant differences. One, we look at the list of all I/O actions and non-determinism oracle results stored in *io_trace* instead of the return value. This is why we need to record the oracle results there. Two, our trace-building must account for the `AddL` and `AddR` expressions. Three, we must know that the *io_trace* is monotone with respect to stepping in the small-step semantics, and with respect to the clock in the functional big-step semantics. The only difficulty in this proof, over the deterministic one, was in handling the `AddL` and `AddR` forms, not in dealing with the oracle or trace.

To get an equivalent non-deterministic labelled transition system (LTS) with I/O actions as labels, one would prove the equivalence entirely in the small-step world with a simulation between the oracle small-step and the LTS semantics.

In the above, there was nothing special about the FOR language itself, and the same connection to small-step semantics could be proved for any situation where the big-step to small-step lemma above holds, along with other basic properties of the semantics. In fact, our proof for the FOR language is based on a general theorem that distills the essence of the approach. (We omit the details, which are obscured by the need to treat the two kinds of semantics abstractly).

8 Related work and discussion

The essence of the functional big-step approach is that the semantics are just an interpreter for the language, modified with a clock to make it admissible in higher-order logic. In this sense, we are just following Reynolds' idea of definitional interpreters [23], but using higher-order logic, rather than a programming language, as the meta-language. Using a clock to handle potential non-termination keeps the mathematics basic, and fits in well with the automation available in HOL4. Other approaches are possible, such as Danielsson's use of a co-inductive partiality monad [11] to define functional big-step semantics. He defines a compiler from a lambda calculus with non-determinism to a stack-based virtual machine, and verifies it, including divergence preservation, in Agda. The compiler that we verify here is more complicated than his, because our target language is lower level, but otherwise a direct comparison is impossible: the source languages are different, and Agda and HOL4 are very different provers.

Nakata and Uustalu [20] give a functional big-step semantics whose co-domain is (possibly infinite) traces of all states the program has passed through, rather than final results. Although their function is recursive, it relies on co-recursive helpers for sequencing and looping: in this way it looks less like a definitional interpreter. They prove equivalence between a variety of trace-based semantics, but do not use the semantics for compiler verification or type soundness. Our FOR language with I/O also keeps traces – although not of all of the program states passed through – but they are kept in the state, rather than in the function's result. Instead of using co-recursion, we take a least upper bound to build possibly infinite traces of I/O actions.

Several improvements have been made to basic, inductive, relational big-step semantics. Leroy and Grall show how to use co-inductive definitions to give a semantics to a lambda-calculus and verify type soundness, and compiler correctness (for a compiler to a virtual machine) while properly handling divergence [18].

Charguéraud's pretty-big-step semantics keeps the co-induction and removes some of the duplication by representing partial computations with new syntax and providing rules for completing the evaluation of the partially evaluated syntax [10]. For the FOR language, he introduces new syntax, `For1`, `For2`, and `For3`, that contain semantic contexts for partial evaluations. The evaluation rule for `For` has a hypothesis about evaluation of `For1`, which represents the state of evaluation after the first expression in `For` has been evaluated. Similarly, the semantics of `For1` is given semantics in terms of `For2`, and so forth. The pretty-big-step approach leads to many rules, but there are fewer than in a conventional big-step definitions, and the duplication is removed by factoring it out into rules that introduce `For1`, `For2`, and `For3`.

Bach Poulsen and Mosses show how to derive a (co-inductive) pretty-big-step semantics from a certain kind of small-step semantics (MSOS). This allows one to get the conciseness of a small-step definition and some of the reasoning benefits of a big-step style [3]. They further show that the duplication between the inductive and co-inductive rules can be reduced by encoding in the state whether the computation is trying to diverge or converge, under certain restrictions [4]. Their approach to encoding control-flow effects in the state could be applied in the functional big-step setting. From the point of view of writing an interpreter, this would correspond to using a state monad to encode an exception monad.

In their book on semantics in Isabelle, Nipkow and Klein use an inductive big-step semantics for a simple imperative language, along with a small-step semantics proved equivalent, and show how to verify a compiler for it [21]. The language cannot have run-time errors, so they do not have to use co-induction. (When they add a type system and possible runtime errors, they switch to small-step). However, their compiler correctness proof and big-step/small-step equivalence proofs each rely on two lemmas. The first assumes a converging big-step execution and builds a small-step trace (their target language has a small-step semantics), just like our corresponding proofs in §3.3 and §7. Their second assumes a small-step trace and shows that the big-step semantics converges to the right thing. With functional big-step semantics, we do not need this direction because we are in a deterministic setting and we correlate the trace length with clock in the first lemma. This is significant because the second lemma has the more difficult proof: any machine state encountered when running the compiled program must be related back to a source program.

Testing semantics To test a semantics, one must actually use it to evaluate programs. Functional big-step semantics can do this out-of-the-box, as can many small-step approaches [13,14]. Where semantics are defined in a relational big-step style, one needs to build an interpreter that corresponds to the relation and verify that they are equivalent – essentially, building a functional big-step semantics anyway. This construction and proof has been done by hand in several projects [6,7,22], and both Coq and Isabelle have mechanisms for automatically deriving functions from inductive relations, although under certain restrictions, and not for co-inductive relations [5,27].

9 Conclusion

We have shown how to take an easy to understand interpreter and use it as a formal semantics suitable for use in an interactive theorem prover. To make this possible we added clocks and oracles to the interpreter. Although our example FOR language is simple, it exhibits a wide range of programming language features including divergence, I/O, exceptions (`Break`), and stores. We have also shown how the functional big-step style can support functional language semantics with Core ML and call-by-value lambda calculus examples.

Although our work was carried out in HOL4, we expect the same lessons to apply in Isabelle/HOL, which has a similar logic, and broadly similar facilities for defining relations and functions. We also expect the functional big-step approach to be applicable in other provers, since it only relies on basic mathematics.

Acknowledgements. We thank Arthur Charguéraud for advice on Coq and pretty-big-step. The first author was supported by the Engineering and Physical Sciences Research Council [EP/K040561/1]. The second author was partially supported by the Swedish Research Council. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Proceedings*, pages 69–83, 2006. doi:10.1007/11693024_6.
2. A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. doi:10.1145/504709.504712.
3. C. Bach Poulsen and P. D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Proceedings*, pages 270–289, 2014. doi:10.1007/978-3-642-54833-8_15.
4. C. Bach Poulsen and P. D. Mosses. Divergence as state in coinductive big-step semantics (extended abstract). In *26th Nordic Workshop on Programming Theory, NWPT '14*, 2014. URL: http://www.plancomps.org/nwpt2014/.
5. S. Berghofer, L. Bulwahn, and F. Haftmann. Turning inductive into equational specifications. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009. Proceedings*, pages 131–146, 2009. doi:10.1007/978-3-642-03359-9_11.
6. S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009. doi:10.1007/s10817-009-9148-3.
7. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 87–100, 2014. doi:10.1145/2535838.2535876.
8. R. Boyer and J. S. Moore. Mechanized formal reasoning about programs and computing machines. In *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.
9. P. Chambart. High level OCaml optimisations. 2013. URL: https://ocaml.org/meetings/ocaml/2013/slides/chambart.pdf.
10. A. Charguéraud. Pretty-big-step semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013. Proceedings*, pages 41–60, 2013. doi:10.1007/978-3-642-37036-6_3.
11. N. A. Danielsson. Operational semantics using the partiality monad. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 127–138, 2012. doi:10.1145/2364527.2364546.
12. D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012. doi:10.1017/S095679681200024X.
13. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 533–544, 2012. doi:10.1145/2103656.2103719.
14. C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 285–296, 2012. doi:10.1145/2103656.2103691.

15. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, 2014. doi:10.1145/2535838.2535841.
16. X. Leroy. 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, POPL 2006*, pages 42–54, 2006. doi:10.1145/1111037.1111042.
17. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.
18. X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.
19. J. S. Moore. Symbolic simulation: An ACL2 approach. In *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Proceedings*, pages 334–350, 1998. doi:10.1007/3-540-49519-3_22.
20. K. Nakata and T. Uustalu. Trace-based coinductive operational semantics for While. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009. Proceedings*, pages 375–390, 2009. doi:10.1007/978-3-642-03359-9_26.
21. T. Nipkow and G. Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. doi:10.1007/978-3-319-10542-0.
22. S. Owens. A sound semantics for OCaml light. In *Programming Languages and Systems: 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *LNCS*, pages 1–15. Springer, 2008. doi:10.1007/978-3-540-78739-6_1.
23. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-order and Symbolic Computation*, 11(4):363–397, 1998.
24. J. Siek. Big-step, diverging or stuck? http://siek.blogspot.com/2012/07/big-step-diverging-or-stuck.html, 2012.
25. J. Siek. Type safety in three easy lemmas. http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html, 2013.
26. M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990. doi:10.1016/0890-5401(90)90018-D.
27. P. Tollitte, D. Delahaye, and C. Dubois. Producing certified functional code from inductive specifications. In *Certified Programs and Proofs - Second International Conference, CPP 2012. Proceedings*, pages 76–91, 2012. doi:10.1007/978-3-642-35308-6_9.
28. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.
29. W. D. Young. A mechanically verified code generator. *J. Autom. Reasoning*, 5(4):493–518, 1989. doi:10.1007/BF00243134.