CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Section 17.1

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

A Correctness Proof for a Toy Compiler

Goal: to study techniques to represent and reason about programming languages.

- interpreting and compiling numeric expressions
  - representation and notation
  - defining the interpreter
  - an instruction set and a virtual machine
  - compiling numeric expressions
  - correctness

Representation and notation

The language will only contain simple ground numeric expressions. So an expression will be either:

- a constant natural number, such as (the unary Peano representations of) 4 or 58, or else
- it will be the sum, difference, product, or quotient of two expressions.

Accordingly, the abstract grammar of such expressions can be informally described as follows:

$$e ::= n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \cdot e_2 \mid e_1/e_2$$

where n ranges over the natural numbers.

Representation and notation

The inductive datatype `Exp` can be used to represent expressions in the language:

```
module TC {

  datatype Exp := (const N)
                | (sum Exp Exp)
                | (diff Exp Exp)
                | (prod Exp Exp)
                | (quot Exp Exp)
 ...
}
```

`TC` stands for "Toy Compiler", the name of the Athena module that will contain the correctness proof.

Representation and notation

The constructor `const` builds simple (constant) natural-number expressions; the constructor `sum` builds sums, and so on.
Thus, for instance, the expression $1 + 2$ is captured by the term

```
(sum (const (S zero))
     (const (S (S zero))))
```

while $1 + (2 \cdot 3)$ is represented by

```
(sum (const (S zero))
     (prod (const (S (S zero)))
           (const (S (S (S zero))))))
```

With infix notation for the binary operators and no parentheses for unary operators, we can write:

```
((const S zero) sum (const S S zero))
((const S zero) sum (const S S zero) prod (const S S S zero))
```

Representation and notation

We specify a high precedence level for `const`, to ensure that it binds tighter than the other constructors (whose default precedence level is 110):

```
set-precedence const 350
```

Thus, the preceding terms representing $1 + 2$ and $1 + (2 \cdot 3)$ can now be written as:

```
(const S zero sum const S S zero)
```

and

```
(const S zero sum const S S zero prod const S S S zero).
```

Representation and notation

It is also convenient to make the expression constructors accept integer numeral arguments as well as natural numbers:

```
define (int->const x) :=
  check {(integer-numeral? x) => (const int->nat x)
         | else => x}

expand-input const [int->nat]
expand-input sum, diff, prod, quot [int->const int->const]
```

- The first `expand-input` directive forces `const` to accept integer numerals as inputs, which will be converted to natural numbers by the procedure `int->nat`.
- The second directive does the same for the four binary constructors of `Exp`, and for both argument positions of each, but using `int->const` rather than `int->nat`.

Representation and notation

Thus, we can now write:

```
> (const 4)

Term: (const (S (S (S (S zero)))))

> (1 sum 2)

Term: (sum (const (S zero))
           (const (S (S zero))))

> (1 sum 2 prod 3)

Term: (sum (const (S zero))
           (prod (const (S (S zero)))
                 (const (S (S (S zero))))))
```

Representation and notation

Going further still, let us overload the usual numeric function symbols (+, etc.) to do double duty as constructors of Exp, and then also as the corresponding operators on natural numbers:

```
overload (+ sum) (- diff) (* prod) (/ quot)
overload (+ N.+) (- N.-) (* N.*) (/ N./)
```

and let us define a few variables ranging over Exp and N:

```
define [e e' e1 e2 e3 e4] := [?e:Exp ?e':Exp ...]
define [n n' n1 n2 n3 n4] := [?n:N ?n':N ?n1:N ...]
```

The sorts of these variables can now serve to disambiguate uses of the overloaded symbols:

```
> (e1 + e2)
Term: (sum ?e1:Exp ?e2:Exp)

> (n1 * n2)
Term: (N.* ?n1:N ?n2:N)
```

Defining the interpreter

Next, we introduce an interpreter that takes as input an expression e and produces a natural number denoting the value of e:

```
declare I: [Exp] -> N [350]
```

For example, the application of `I` to the expression

`(1 sum 2)`

should represent the result 3, or more precisely, the natural number

`(S (S (S zero)))`.

Note that we have given `I` a precedence level higher than the precedence levels of the numeric operators `N.+`, `N.-`, etc. Thus, for instance,

`(I e1 + I e2)`

is parsed as the application of `N.+` to `(I e1)` and `(I e2)`.

Defining the interpreter

We can now specify the behavior of the interpreter `I` as follows:

```
assert* I-def :=
  [(I const n = n)
   (I (e1 + e2) = I e1 + I e2)
   (I (e1 - e2) = I e1 - I e2)
   (I (e1 * e2) = I e1 * I e2)
   (I (e1 / e2) = I e1 / I e2)]
```

If we view `I` as a specification of the meaning of these expressions, then

- the first equation can be understood as saying that the meaning of a constant expression `(const` n`)` is n.
- the remaining equations are all similar in their structure, and are paradigms of recursive compositionality.

Notice symbol overloading, e.g., the occurrence of `+` on the left resolves to `sum`, whereas the occurrence of `+` on the right resolves to `N.+`.

Defining the interpreter

We can turn to `eval` to test the definition. It will be clearer to see evaluation results in integer notation, so we first issue a `transform-output` directive.

```
transform-output eval [nat->int (clist->alist nat->int)]

> let {2+3  := (2 sum 3);
       10-6 := (10 diff 6);
       10/2 := (10 quot 2);
       9*9  := (9 prod 9)}
   (print "\nValue of 2+3:\t" (eval I 2+3) "\nand 10-6:\t" (eval I 10-6)
          "\nand 10/2:\t" (eval I 10/2) "\nand 9*9:\t" (eval I 9*9) "\n")

Value of 2+3:    5
and 10-6:        4
and 10/2:        5
and 9*9:         81

Unit: ()
```

An instruction set and a virtual machine

We are now ready to introduce a lower-level "machine language" into which we will be compiling the numeric expressions (ground `Exp` terms).

This machine language will consist of a few simple commands for manipulating a stack, namely:

- pushing a natural number on top of the stack; and
- adding, subtracting, multiplying, or dividing the top two elements of the stack, and replacing them with the result of the operation.

The datatype `Command` represents these possibilities:

```
datatype Command := (push N) | add | sub | mult | div
```

An instruction set and a virtual machine

As with `const`, it will be convenient to use integer numerals with `push`, in addition to natural numbers. We also set its precedence level to 350:

```
expand-input push [int->nat]
set-precedence push 350
```

A program in this machine language is simply a list of commands. We define an abbreviation for this sort:

```
define-sort Program := (List Command)
```

We also introduce a sort `Stack` as an abbreviation for lists of natural numbers and some appropriately named variables for these two sorts:

```
define-sort Stack := (List N)

define [cmd cmd' prog prog1 prog2 stack stack1 stack2] :=
       [?cmd:Command ... ?prog:Program ... ?stack:Stack ...]
```

An instruction set and a virtual machine

We can now define the operation of the abstract machine by means of a binary function `exec` that takes as inputs

- a program (i.e., a list of commands) and
- a stack of natural numbers;

and executes the program with respect to the given stack, eventually producing a natural number as the result:

```
declare exec: [Program Stack] -> N [wrt 101 [(alist->clist id)
                                             (alist->clist int->nat)]]
```

We have expanded the inputs of `exec` so that it can work directly with Athena (square-bracket) lists on both argument positions, and so that any elements of the second list argument that happen to be integer numerals are automatically converted to natural numbers.

An instruction set and a virtual machine

We have also set `exec`'s precedence level to 101, and we have introduced the name `wrt` as a more infix-friendly alias for it, so we can write, for instance, `(prog wrt stack)` interchangeably with `(exec prog stack)`.

Because we will be dealing with lists quite a bit, we also expand the input range of the "consing" constructor `::` (the reflexive constructor for the `List` datatype), so that it can accept Athena lists in the second argument position, and we define `++` as `List.join`.

```
expand-input :: [id (alist->clist int->nat)]
define ++ := List.join
```

An instruction set and a virtual machine

The definition of exec will constitute a sort of (small-step) operational semantics for our machine language.

Again, we define it with a number of axioms using structural recursion on the first argument—the input list of commands:

```
assert* exec-def :=
    [([] wrt n::_ = n)
     ((push n)::rest wrt stack = rest wrt n::stack)
     (add::rest  wrt n1::n2::stack = rest wrt (n1 + n2)::stack)
     (sub::rest  wrt n1::n2::stack = rest wrt (n1 - n2)::stack)
     (mult::rest wrt n1::n2::stack = rest wrt (n1 * n2)::stack)
     (div::rest  wrt n1::n2::stack = rest wrt (n1 / n2)::stack)]
```

An instruction set and a virtual machine

We again use `eval` to test the definition of `exec`:

- Execute an empty program on a nonempty stack:

```
> (eval [] wrt [99])

Term: 99
```

- Execute an addition instruction on a stack with at least two numbers:

```
> (eval [add] wrt [2 3])

Term: 5
```

- First add 2 and 3, then multiply the result by 6:

```
> (eval [add mult] wrt [2 3 6])

Term: 30
```

An instruction set and a virtual machine

- Add 2 and 3, multiply by 6, then divide by 10:

```
> (eval [add mult div] wrt [2 3 6 10])

Term: 3
```

- Execute a binary instruction with too few stack operands:

```
> (eval [add] wrt [2])

Unable to reduce the term:

(TC.exec (:: TC.add
             nil)
         (:: (S (S zero))
             nil))

to a normal form.

Unit: ()
```

An instruction set and a virtual machine

- And a longer example:

```
> define input-program := [(push 1) (push 2) add (push 2) mult mult]

List input-program defined.

> define input-stack := [4]

List input-stack defined.

> (eval input-program wrt input-stack)

Term: 24
```

An instruction set and a virtual machine

Let us now define a virtual machine that executes a given program on the empty stack:

```
declare run-vm: [Program] -> N [[(alist->clist id)]]

assert* vm-def := [(run-vm prog = prog wrt [])]
```

And again using `eval`:

```
> (eval run-vm [(push 2) (push 3) add])

Term: 5
```

Compiling numeric expressions

We are finally ready to define the compiler.

- It takes an arithmetic expression and
- returns a program in the machine language that will (hopefully) compute the value of the expression:

```
declare compile: [Exp] -> Program [380]
```

Compiling numeric expressions

The definition of the compiler is given by structural recursion:

```
assert* compiler-def :=
   [(compile const n = (push n)::[])
    (compile (e1 + e2) = compile e2 ++ compile e1 ++ [add])
    (compile (e1 - e2) = compile e2 ++ compile e1 ++ [sub])
    (compile (e1 * e2) = compile e2 ++ compile e1 ++ [mult])
    (compile (e1 / e2) = compile e2 ++ compile e1 ++ [div])]
```

- For an expression of the form `(const` n`)`, the compiled program is just the one-element list `[(push` n`)]`.
- For a compound expression `(`e_1 *op* e_2`)`, the idea is
  - to recursively compile e_1 and e_2,
  - join the obtained instruction lists in the right order, and then
  - append the proper machine instruction for *op*.

Compiling numeric expressions

We can test the compiler on a number of different inputs:

```
> (eval compile const 5)

List: [(TC.push (S (        (S zero))))))]

> (eval compile (2 sum 3))

List: [(TC.push (S (S (S zero))))          # push 3
       (TC.push (S (S zero)))              # push 2
       TC.add]                             # add

> (eval compile (2 sum 3 prod 5))

List: [(TC.push (S (S (S (S (S zero))))))  # push 5
       (TC.push (S (S (S zero))))          # push 3
       TC.mult                             # multiply
       (TC.push (S (S zero)))              # push 2
       TC.add]                             # add
```

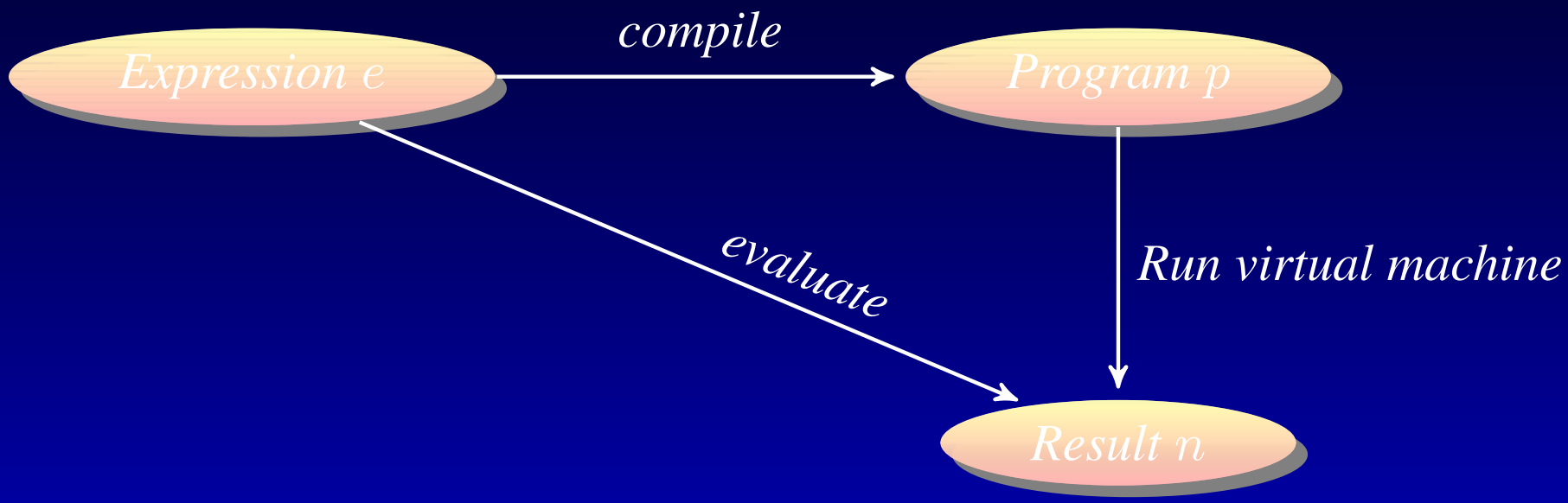
Correctness

- Is this simple compiler correct? What does correctness mean in this context?
- As with all compilers, correctness here means that the semantics of the target program are the same, in some appropriate sense, as the semantics of the source.
- In this case, the semantics of the source expression are given by the interpreter `I`, while the semantics of the compiled program are given by `run-vm`.
- To say that the two semantics are the same means that if we apply the virtual machine to the compiled program we will obtain the same result that we get by applying the interpreter `I` to the source expression:

```
define compiler-correctness := (forall e . run-vm compile e = I e)
```

Correctness—commuting diagram

We can express this condition graphically as follows:



A diagram of this form is known as a *commuting diagram*. In such a diagram, every directed arc represents a function (or a *morphism*).

- The idea is that the function compositions represented by any two directed paths from the same starting point to the same destination are identical functions: They produce the same outputs when applied to the same inputs.

Correctness—generalizing correctness property

So `compiler-correctness` is the property that we need to prove.

- Since this property is universally quantified over all expressions, which are inductively generated, we decide to go about it by way of structural induction.
- But a bit of experimentation will reveal that, as stated, the property is not quite suitable for induction.
- The semantics of our machine language are given in terms of `exec`, which takes a stack as an explicit second argument, and we need to take that into account.
- We thus define the property to be proven inductively as follows:

```
define (correctness e) := (forall stack . compile e wrt stack = I e)
```

Correctness—and specializing property

- If we manage to prove

```
(forall e . correctness e)
```

  by induction, we can then easily derive the desired result, `compiler-correctness`, by specializing the stack argument with the empty list.

Correctness—proving by structural induction

Unfortunately, when we try an inductive proof, we find a problem on the inductive steps. Suppose, for example, that we are inducting on an expression of the form `(`e_1 `sum` e_2`)`, so we need to derive

`(compile (`e_1 `sum` e_2`) wrt stack = I (`e_1 `sum` e_2`))`

(where `stack` is a variable introduced by some preceding `pick-any`). The only option here is to reduce the term on the left-hand side to the term on the right-hand side. No axioms for `exec` are applicable, so our only course of action is to expand the subterm

`(compile (`e_1 `sum` e_2`))`,

which will transform the left-hand side of the equation into:

`(compile` e_1 `++ compile` e_2 `++ [add] wrt stack).`

Correctness—getting stuck

$$\texttt{(compile } e_1 \texttt{ ++ compile } e_2 \texttt{ ++ [add] wrt stack).} \quad (1)$$

At this point, however, we are not able to exploit our inductive hypotheses about e_1 and e_2, e.g.:

$$\texttt{(compile } e_1 \texttt{ wrt stack) = I e1.}$$

The problem is that the inductive subterms `(compile` e_1`)` and `(compile` e_2`)` are directly embedded inside `List.join` (`++`) superterms, whereas our inductive property, `correctness`, requires them to be embedded directly inside `exec` (`wrt`). Thus, (1) is irreducible and we cannot get any further in deriving:

$$\texttt{(compile (}e_1 \texttt{ sum } e_2\texttt{) wrt stack = I (}e_1 \texttt{ sum } e_2\texttt{))}$$

Correctness—strenghtening inductive hypothesis

When we *generalize* the property we are trying to prove—a technique known as *strengthening* the inductive property—then the proof succeeds, and we can obtain the initially desired property by simple specialization.

In this case, we will generalize the inductive property so that `List.join` (++) appears directly in it:

```
define (correctness e) :=
  (forall prog stack . compile e ++ prog wrt stack = prog wrt I e::stack)
```

Crucially, this is a generalization of our initial correctness property because the latter can be readily obtained from this by substituting the empty list of commands for the universally quantified `prog`.

Correctness—basis step

Let us now go ahead and prove that every expression has this property:

```
(forall e . correctness e).
```

As usual, we proceed by structural induction. The basis step occurs when the expression is of the form `(const` n`)`. We express the step as a method parameterized over n:

```
define (basis-step n) :=
  pick-any p:Program stack:Stack
    (!chain [((compile const n) ++ p wrt stack)
             = ([(push n)] ++ p wrt stack)        [compiler-def]
             = ((push n)::p wrt stack)          [List.join.left-singleton]
             = (p wrt n::stack)                  [exec-def]
             = (p wrt I const n :: stack)        [I-def]])
```

`List.join.left-singleton` is the following property:

```
(forall x L . [x] ++ L = x::L).
```

Correctness—basis step testing

We can test the basis step by itself:

```
> (!basis-step zero)

Theorem: (forall ?p:(List TC.Command)
           (forall ?stack:(List N)
             (= (TC.exec (List.join (TC.compile (TC.const zero))
                                    ?p:(List TC.Command))
                         ?stack:(List N))
                (TC.exec ?p:(List TC.Command)
                         (:: (TC.I (TC.const zero))
                             ?stack:(List N))))))
```

Correctness—inductive steps

We continue with the four inductive cases, when the expression is either a sum, a difference, a product, or a quotient.

- Instead of writing four separate inductive subproofs, one for each case, we will write a single method that takes the overall expression as input, determines its form (whether it is a sum, product, etc.), and then acts accordingly.
- This is possible because the reasoning is essentially the same in all four cases, as we might have anticipated in advance (e.g., by looking at the `exec` axioms for the four commands, which shows that they all have the exact same structure).

We will package this reusable bit of reasoning into a unary method called `inductive-step`.

Correctness—inductive steps

We first need a simple procedure that takes a binary expression constructor *exp-op* and produces the corresponding machine-language command and the corresponding operator on natural numbers:

```
define (exp-op->cmd-and-num-op exp-op) :=
  match exp-op {
     sum =>  [add N.+]
  | diff => [sub N.-]
  | prod => [mult N.*]
  | quot => [div N./]
  }
```

Correctness—inductive steps method

We can now define the `inductive-step` method as follows:

```
define (inductive-step e) :=
 match e {
  ((some-symbol exp-op:(OP 2)) e1:Exp e2:Exp) =>
    let {[cmd num-op:(OP 2)] := (exp-op->cmd-and-num-op exp-op);
         # Two inductive hypotheses, one for each sub-expression,
         # can be assumed to be in the a.b. when this method is called.
         [ih1 ih2] := [(correctness e1) (correctness e2)]}
      pick-any p:Program stack:Stack
       (!chain
         [(compile (e1 exp-op e2) ++ p wrt stack)
        = (compile e2 ++ compile e1 ++ [cmd] ++ p wrt stack)
                                        [compiler-def List.join.Associative]
        = ([cmd] ++ p wrt (I e1)::(I e2)::stack)  [ih1 ih2]
        = (cmd::p wrt (I e1)::(I e2)::stack)  [List.join.left-singleton]
        = (p wrt (I e1 num-op I e2)::stack)       [exec-def]
        = (p wrt (I e)::stack)                    [I-def]])
 }
```

Correctness—inductive steps method

The `OP` annotations ensure that the corresponding names can be used in infix with the arities specified in the annotations.

- For example, the annotation `num-op:(OP 2)` allows `num-op` to be subsequently used as a binary function symbol.

`List.join.Associative` is simply the associativity property for `List.join`:

```
(forall L1 L2 L3 . (L1 ++ L2) ++ L3 = L1 ++ (L2 ++ L3)).
```

Correctness—inductive steps method

Let's go through the reasoning line by line.

- The first order of business is to decompose the input expression, which must be of the form `(`e_1 *exp-op* e_2`)`.
  - We then use

    `exp-op->cmd-and-num-op`

    to obtain the machine-language command *cmd* and natural-number operator *num-op* that correspond to the constructor *exp-op*; and
  - we give the names `ih1` and `ih2` to the inductive hypotheses for the subexpressions e_1 and e_2, respectively.
- Then, on the `pick-any` line, we consider an arbitrary program (i.e., command list) `p` and an arbitrary `stack` and proceed to derive the desired identity by chaining.

Correctness—inductive steps method

- The first chaining step goes through by the definition of `compile` and the associativity of list concatenation (`++`). Observe that this step will go through only if `cmd` is the correct analogue of `exp-op`.
- The second step goes through owing to the inductive hypotheses. (We are assuming that the method will only be invoked in assumption bases that contain these inductive hypotheses.)
- The third step simply replaces the subterm `[cmd] ++ p` with `cmd::p`, which follows from `List.join.left-singleton`.
- The fourth step follows from the definition of `exec`. Note here, too, that this will go through only if `num-op` correctly corresponds to `exp-op`: `N.+` to `sum`, and so forth.
- Finally, the fifth step goes through by the definition of the interpreter.

Correctness—inductive steps method testing

A benefit of factoring out all the inductive work into a separate method is that we can test its reasoning independently:

```
set-flag print-var-sorts "off"
set-flag print-qvar-sorts "off"

> pick-any e1:Exp e2:Exp
    assume (correctness e1)
      assume (correctness e2)
        (!inductive-step (e1 + e2))
```

Correctness—inductive steps method testing

```
Theorem: (forall ?e1
           (forall ?e2
             (if (forall ?prog
                   (forall ?stack
                     (= (exec (List.join (compile ?e1)
                                         ?prog)
                              ?stack)
                        (exec ?prog
                              (:: (I ?e1)
                                  ?stack)))))
                 (if (forall ?prog
                       (forall ?stack
                         (= (exec (List.join (compile ?e2)
                                             ?prog)
                                  ?stack)
                            (exec ?prog
                                  (:: (I ?e2)
                                      ?stack)))))
```

Correctness—inductive steps method testing

```
(forall ?p
  (forall ?stack
    (= (exec (List.join (compile (sum ?e1 ?e2))
                        ?p)
               ?stack)
       (exec ?p
             (:: (I (sum ?e1 ?e2))
                 ?stack)))))))))
```

Correctness—inductive proof

The overall inductive proof now takes the following form:

```
> define main-correctness-theorem :=
    by-induction (forall e . correctness e) {
      (const n) =>          (!basis-step n)
    | (e as (sum _ _)) =>   (!inductive-step e)
    | (e as (diff _ _)) =>  (!inductive-step e)
    | (e as (prod _ _)) =>  (!inductive-step e)
    | (e as (quot _ _)) =>  (!inductive-step e)
    }
Theorem: (forall ?e:TC.Exp
           (forall ?prog:(List TC.Command)
             (forall ?stack:(List N)
               (= (exec (List.join (compile ?e)
                                   ?prog)
                        ?stack)
                  (exec ?prog
                        (:: (I ?e)
                            ?stack))))))
```

Correctness—the final proof

The top-level result that we have been after can now be derived in a few lines:

```
conclude compiler-correctness
  pick-any e
    (!chain
      [(run-vm compile e)
     = (compile e wrt [])              [vm-def]
     = (compile e ++ [] wrt [])        [List.join.right-empty]
     = ([] wrt (I e)::[])              [main-correctness-theorem]
     = (I e)                           [exec-def]])

Theorem: (forall ?e:TC.Exp
           (= (TC.run-vm (TC.compile ?e))
              (TC.I ?e)))
```

`List.join.right-empty` is the following property:

```
(forall L . L ++ [] = L).
```

Some learned lessons

- *Computable definitions (and executable specifications in general)*:
  - the ability to compute with the logical content of sentences is a powerful tool for theory development.
  - it allows us to mechanically test our definitions and conjectures.
  - it enables *model checking* or *automated testing* which may falsify a conjecture, rendering any attempt to prove the conjecture futile, and further providing a counter-example.

Some learned lessons

- *Computation in the service of notation*: can greatly enhance the readability of specifications and proofs.
  - dynamic overloading,
  - precedence levels,
  - input expansions,
  - output transformations,
  - simple static scoping combined with the ability to name symbols and even variables
  - the use of procedures for building sentences with similar structure,
    - e.g., defining the inductive predicate for a proof by structural induction.

Some learned lessons

- *Assuming more to prove more*:
  - We have again demonstrated the usefulness of induction strengthening.
- *Proof abstraction*:
  - Using methods to package up recurring bits of reasoning is invaluable for structured proof engineering.
- *Fundamental datatypes*:
  - Here we have used lists to model programs as sequences of instructions.
  - A vast range of interesting systems and processes can be represented using basic structures: numbers, lists, options (e.g., useful for modeling error handling), ordered pairs, sets, and maps.