CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Section 17.2

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

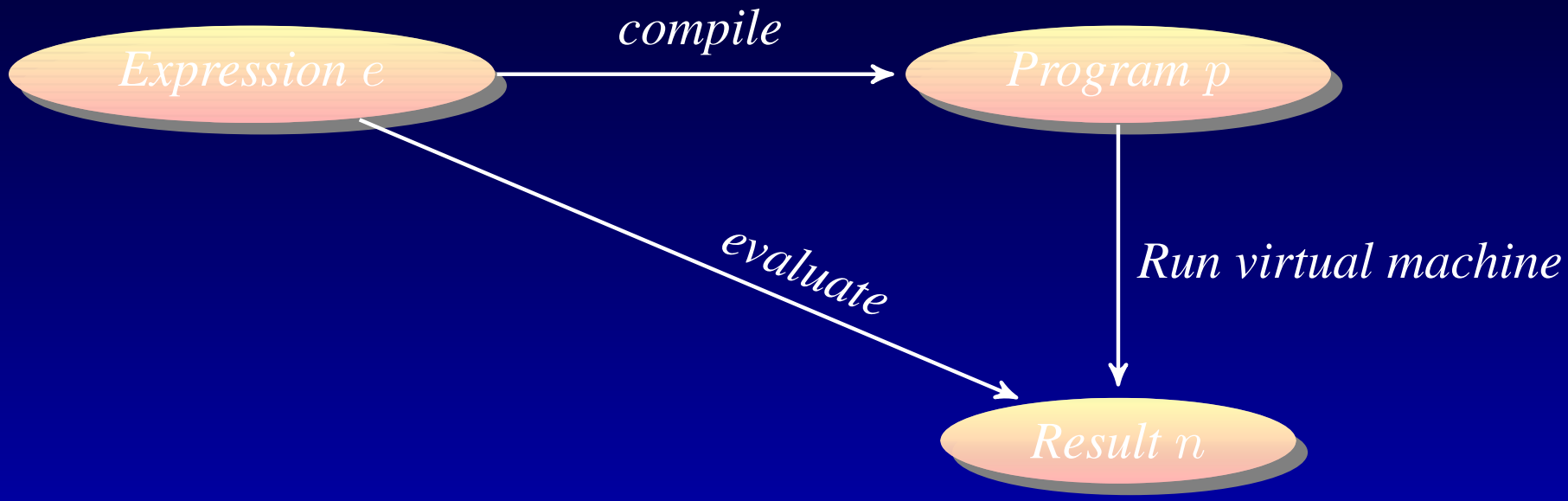
Error Handling in the Toy Compiler

Goal: to study techniques to represent and reason about programming languages, including using options to formalize errors.

- *toy compiler for numeric expressions*
  - *defining the interpreter*
  - *an instruction set and a virtual machine*
  - *compiling numeric expressions*
  - *correctness*
- handling errors explicitly

Correctness—commuting diagram

Recall the correctness property for the toy compiler:



The property states that if we apply the virtual machine semantics 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)
```

Toy Compiler

We specified numeric expressions and the interpreter `I` as follows:

```
module TC {
  datatype Exp := (const N)
                | (sum Exp Exp)
                | (diff Exp Exp)
                | (prod Exp Exp)
                | (quot Exp Exp)

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

  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)]
  ...
}
```

Toy Compiler

Our instruction set and virtual machine `run-vm` as follows:

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

define-sort Program := (List Command)
define-sort Stack := (List N)

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

declare run-vm: [Program] -> N [[(alist->clist id)]]
assert* vm-def := [(run-vm prog = prog wrt [])]
```

Toy Compiler

And finally, our compiler `compile` as follows:

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

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])]
```

Correctness proof

Our correctness proof used an equality chain:

```
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)))
```

Correctness—inductive proof

The main lemma was proved by induction on expressions:

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

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)
    }
```

Correctness—basis step

With the basis step 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]])
```

Correctness—inductive steps method

And 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]])
 }
```

Handling errors explicitly

Our first specification did not deal with errors, neither in the interpreter nor in the compiler or virtual machine.

- What happens if the interpreter is used to evaluate a division by `zero`?
- What happens if the empty program is applied on the empty stack?
- What happens if a binary command is executed on a stack with only one member (or no members!)?
- What happens if the virtual machine encounters a division by zero?

Our specification so far has not provided answers to these questions.

Handling errors explicitly

- A legitimate way of dealing with errors is not to say anything about them.
- This technique is known as *underspecification*, and it results in a theory with so-called *loose semantics*: the theory has many different models, i.e., there are very many different functions that satisfy the demands made by the theory.
- Consider, for instance, our specification of the interpreter `I`. That specification does not determine a *unique* function from `Exp` to `N`. It does not tell us, for instance, the value of the term

```
(I (0 quot 0)).
```
(1)

- As it stands, there are many different functions from `Exp` to `N` that satisfy the given axioms, and it is in that sense that the specification is loose.

Handling errors explicitly

From an operational perspective, you can observe this by trying to evaluate a term such as (1):

```
> (eval I (0 quot 0))

Unable to reduce the term:

(TC.I (TC.quot (TC.const zero)
               (TC.const zero)))

to a normal form.

Unit: ()
```

Athena cannot find any information in the assumption base that can be used to obtain a canonical natural number from this input term.

Handling errors explicitly

The underspecification can be traced to the definition of division on natural numbers:

```
extend-module N {

  declare /: [N N] -> N [300 [int->nat int->nat]]

  module Div {
    assert* def :=  [(x < y ==> x / y = zero)
                     (~ x < y & zero < y ==> x / y = S ((x - y) / y))]
  }
```

- If the first argument (the *dividend*) is less than the second argument (the *divisor*), then the result of the division is 0.
- If the dividend x is not less than the divisor y *and* if the divisor is positive, then the result of the division is one more than the result of dividing $x - y$ by y.

Handling errors explicitly

- In situations in which we want to state exactly when errors arise and how they are to be handled, and to be able to reason about them, underspecification is not the right approach.
- Such situations are not uncommon, especially when we are defining machines. We then want our specification to handle errors explicitly, and thus to be a theory with so-called tight—rather than loose—semantics.
- In other words, we want our theory to have a unique model (up to isomorphism). In this case, for instance, it should pick out unique functions for `I` and for `exec`.

Handling errors explicitly

We will handle errors using the `Option` datatype.

- In the case of the interpreter `I`, for instance, the new signature will be:

```
declare I': [Exp] -> (Option N)
```

- We prime the name of the interpreter (`I'`) to distinguish it from the original version.
- Intuitively, this says that `I'` takes an expression and *might* return a natural number, if all goes well; otherwise it will return `NONE`.
- Options are pervasive in functional programming, and their use for handling anomalous situations is more realistic (and perhaps more intuitive) than other approaches.

Extending the compiler with error handling

The equation for the `const` constructor is simple enough:

```
(I' const n = SOME n)
```

For other constructors, we need to distinguish between success and failure. Consider, e.g., a term of the form:

$$\texttt{(I' } e_1 \texttt{ + } e_2\texttt{)}.$$

In the underspecification approach, the algorithm was simple:

- Recursively obtain the values of the two subexpressions e_1 and e_2 and then return their sum.

Now, however, we must keep in mind that `I` returns a natural number *option*, and hence that the recursive invocations of `I` on either of the two subexpressions might fail (i.e., they might return `NONE`).

Extending the compiler with error handling

So the algorithm now is this:

- If the recursive evaluation of *both* subexpressions succeeds, that is, both e_1 and e_2 return values of the form `(SOME` n_1`)` and `(SOME` n_2`)`, respectively, then we return `(SOME (`n_1 `+` n_2`))`.
- But if either recursive call fails, then we fail as well.

So we have two axioms for sums, one for success (a "positive" axiom), and one for failure (a "negative" axiom).
The positive axiom is this:

```
(I' e1 = SOME n1 & I' e2 = SOME n2 ==> I' e1 + e2 = SOME n1 + n2)
```

And the negative axiom is this:

```
(I' e1 = NONE | I' e2 = NONE ==> I' e1 + e2 = NONE)
```

The treatment of subtractions and products is similar.

Extending the compiler with error handling

Division must be handled a little differently.

The positive axiom for division must ensure not only that the recursive calls to the two subexpressions succeed, but also that the value returned by the second subexpression (the divisor operand) is nonzero:

```
(I' e1 = SOME n1 & I' e2 = SOME n2 & n2 =/= zero ==>
   I' e1 / e2 = SOME n1 / n2)
```

Symmetrically, the negative axiom must fail if either of the two subexpressions fails or if the divisor yields a zero value:

```
 (I' e1 = NONE | I' e2 = NONE | I' e2 = SOME zero ==> I' e1 / e2 = NONE)
```

Extending the compiler with error handling

Thus we arrive at the following definition:

```
assert* I'-def :=
  [(I' const n = SOME n)

   (I' e1 = SOME n1 & I' e2 = SOME n2 ==> I' e1 + e2 = SOME n1 + n2)
   (I' e1 = SOME n1 & I' e2 = SOME n2 ==> I' e1 - e2 = SOME n1 - n2)
   (I' e1 = SOME n1 & I' e2 = SOME n2 ==> I' e1 * e2 = SOME n1 * n2)
   (I' e1 = SOME n1 & I' e2 = SOME n2 & n2 =/= zero ==>
                                        I' e1 / e2 = SOME n1 / n2)

   (I' e1 = NONE | I' e2 = NONE ==> I' e1 + e2 = NONE)
   (I' e1 = NONE | I' e2 = NONE ==> I' e1 - e2 = NONE)
   (I' e1 = NONE | I' e2 = NONE ==> I' e1 * e2 = NONE)
   (I' e1 = NONE | I' e2 = NONE | I' e2 = SOME zero ==>
                                        I' e1 / e2 = NONE)]
```

Extending the compiler with error handling

We go on to test the new interpreter:

```
> (eval I' 2 sum 3)

Term: (SOME 5)

> (eval I' 3 prod 5)

Term: (SOME 15)

> (eval I' 11 sum 3 prod 3)

Term: (SOME 20)

> (eval I' 18 quot 6)

Term: (SOME 3)
```

Extending the compiler with error handling

Including some error-producing expressions:

```
> (eval I' 4 quot 0)

Term: NONE:(Option N)

> (eval I' 4 quot (2 diff 2))

Term: NONE:(Option N)
```

Extending the compiler with error handling

Let us revisit the semantics of exec. It will now return a natural number option:

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

When the program to be executed is empty, we can distinguish two subcases: The stack is also empty, or it is not. The first subcase now leads to an error (NONE result), while the second is handled as before but wrapped inside a SOME constructor:

```
define exec-axiom-empty-1 := (nil wrt' nil = NONE)

define exec-axiom-empty-2 := (nil wrt' n::_ = SOME n)

define exec-axiom-push :=  # No change
  ((push n)::prog wrt' stack = prog wrt' n::stack)
```

Extending the compiler with error handling

We continue with the binary commands.

When can their execution go wrong? Only if the stack does not have enough operands. So we'll assert a negative axiom for each such command:

```
define neg-op-axioms :=
  (flatten (map lambda (op) [(op::_ wrt' [] = NONE)
                             (op::_ wrt' [_] = NONE)]
                [add sub mult div]))
```

On the positive side for binary commands except division (no change), we have:

```
define pos-op-axioms :=
  [(add::prog  wrt' n1::n2::stack = prog wrt' (n1 + n2)::stack)
   (sub::prog  wrt' n1::n2::stack = prog wrt' (n1 - n2)::stack)
   (mult::prog wrt' n1::n2::stack = prog wrt' (n1 * n2)::stack)]
```

Extending the compiler with error handling

Finally, for the division operation we have:

```
define pos-div-axiom :=
  (n2 =/= zero ==>
    div::prog wrt' n1::n2::stack = prog wrt' (n1 / n2)::stack)

define neg-div-axiom :=
  (n2 = zero ==> div::prog wrt' _::n2::stack = NONE)
```

We collect all relevant equations in a single list and assert them:

```
assert* exec'-def :=
  (join [exec-axiom-empty-1 exec-axiom-empty-2 exec-axiom-push]
        pos-op-axioms neg-op-axioms [pos-div-axiom neg-div-axiom])
```

Extending the compiler with error handling

The virtual machine is defined as before, except that its signature is now different:

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

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

Extending the compiler with error handling

We can now test the new semantics:

```
> (eval [] wrt' [5])

Term: (SOME 5)

> (eval [(push 3)] wrt' [])

Term: (SOME 3)

> (eval [(push 3) (push 4) add] wrt' [])

Term: (SOME 7)
```

Extending the compiler with error handling

Including some error-producing programs:

```
> (eval run-vm' [(push 1) add])

Term: NONE:(Option N)

> (eval run-vm' [(push 0) (push 4) div])

Term: NONE:(Option N)
```

The compiler has the same signature and definition as before, so we are done with the specification!

Correctness of compiler with error handling

The previous correctness property needs to be slightly modified to ensure that the expression being compiled has *some* value:

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

Let's first check to see whether we might have missed anything obvious:

```
define conjecture := (forall e . correctness' e)
```

```
> (falsify conjecture 10)
```

```
Term: 'failure
```

Correctness of compiler with error handling

Because correctness is now conditional, we will often find ourselves assuming a hypothesis of the form

$$\texttt{(I' } e \texttt{ = SOME } n\texttt{).} \tag{2}$$

When e is a complex (nonconstant) expression, there are certain useful conclusions we can derive from (2). Suppose, for instance, that e is of the form $\texttt{(}e_1 \texttt{ sum } e_2\texttt{)}$. Then, from the assumption $\texttt{(I' } e_1 \texttt{ sum } e_2 \texttt{ = SOME } n\texttt{)}$, we should be able to conclude that there exist natural numbers n_1 and n_2 such that:

1. $\texttt{(I' } e_1 \texttt{ = SOME } n_1\texttt{)}$;
2. $\texttt{(I' } e_2 \texttt{ = SOME } n_2\texttt{)}$; and
3. $(n = n_1 + n_2)$.

Correctness of compiler with error handling

In other words, if (I' e_1 sum e_2 = SOME n) holds, then the recursive evaluations for the two subexpressions e_1 and e_2 must have succeeded, and moreover, the results returned by them must be related to n in the appropriate way.

The reasoning required to derive this conclusion can be packaged into a generic unary method, `get-lemma`, that takes any premise of the form

$$\texttt{(I' } e_1 \ \textit{op} \ e_2 = \texttt{SOME } n\texttt{)},$$

where *op* is either `sum`, `diff`, or `prod` (we will handle `quot` with a separate method); and returns the preceding existential quantification.

Correctness of compiler with error handling

The `get-lemma` has the following structure:

```
define get-lemma :=
  method (premise)
    match premise {
      (= (I' (exp-op:(OP 2) e1 e2))
         (SOME n)) =>
        let {[_ num-op:(OP 2)] := (exp-op->cmd-and-num-op exp-op)}
          (!force (exists n1 n2 .
                    I' e1 = SOME n1 &
                    I' e2 = SOME n2 &
                    n = n1 num-op n2))
    }
```

where **force** can be replaced by an appropriate deduction.

Correctness of compiler with error handling

We define a similar method, `get-div-lemma`, to obtain a similar result for divisions. Here the input premise will be of the form `(I'` e_1 `div` e_2 `= SOME` n`)`, and the method will conclude that there exist natural numbers n_1 and n_2 such that:

1. `(I'` e_1 `= SOME` n_1`)`;
2. `(I'` e_2 `= SOME` n_2`)`;
3. `(`n_2 `=/= zero)`; and
4. $n = n_1$ `/` n_2.

Correctness of compiler with error handling

So the body of the conclusion here is slightly more complicated in that it includes an additional conjunct: that n_2 is nonzero. Here is the skeleton of this method:

```
define get-div-lemma :=
  method (premise)
    match premise {
     ((I' (e1 quot e2)) = (SOME n)) =>
       (!force (exists n1 n2 .
                  I' e1 = SOME ?n1 &
                  I' e2 = SOME ?n2 &
                  n2 =/= zero &
                  n = n1 / n2))
    }
```

Correctness of compiler with error handling

Let us now proceed with the inductive proof of the desired property:

`(forall e . correctness' e).`

The basis case for expressions of the form `(const` k`)` follows:

```
define basis-step :=
  method (k)
    pick-any n:N prog:Program stack:Stack
      assume hyp := (I' const k = SOME n)
        let {k=n := (!chain->
                      [(SOME k) <-- (I' const k)  [I'-def]
                                  = (SOME n)      [hyp]
                    ==> (k = n)                   [option-results]])}
          (!chain
            [(compile const k ++ prog wrt' stack)
           = ([(push k)] ++ prog wrt' stack)      [compiler-def]
           = (push k :: prog wrt' stack)      [List.join.left-singleton]
           = (prog wrt' k::stack)                 [exec'-def]
           = (prog wrt' n::stack)                 [k=n]])
```

Correctness of compiler with error handling

The inductive-step method for the first three expression constructors:

```
define istep :=
  method (exp)
    match exp {
     ((some-symbol op:(OP 2)) e1 e2) =>
        pick-any n:N prog:Program stack:Stack
          assume hyp := (I' e1 op e2 = SOME n)
             let {[cmd num-op:(OP 2)] := (exp-op->cmd-and-num-op op);
                  [ih1 ih2] := [(correctness' e1) (correctness' e2)];
                  lemma := (!chain->
                                [hyp ==> (exists n1 n2 .
                                           I' e1 = SOME n1 &
                                           I' e2 = SOME n2 &
                                           n = n1 num-op n2)
                                 [get-lemma]])
                }
```

Correctness of compiler with error handling

```
        pick-witnesses n1 n2 for lemma spec-lemma
          (!chain
            [(compile (e1 op e2) ++ prog wrt' stack)
           = ((compile e2 ++ compile e1 ++ [cmd]) ++ prog wrt' stack)
               [compiler-def]
           = (compile e2 ++ compile e1 ++ [cmd] ++ prog wrt' stack)
               [List.join.Associative]
           = ([cmd] ++ prog wrt' n1::n2::stack)
               [ih1 ih2]
           = (cmd::prog wrt' n1::n2 ::stack)
               [List.join.left-singleton]
           = (prog wrt' (n1 num-op n2)::stack)
               [exec'-def]
           = (prog wrt' (n :: stack))
               [spec-lemma]])
      }
```

Correctness of compiler with error handling

The inductive step for quotients is similar:

```
define istep-div :=
  method (exp)
    match exp {
     (quot e1 e2) =>
       pick-any n:N prog:Program stack:Stack
         assume hyp := (I' e1 quot e2 = SOME n)
           let {[ih1 ih2] := [(correctness' e1) (correctness' e2)];
                lemma := (!chain->
                          [hyp ==> (exists n1 n2 .
                                    I' e1 = SOME n1 &
                                    I' e2 = SOME n2 &
                                    n2 =/= zero &
                                    n = n1 / n2)
                           [get-div-lemma]])}
```

Correctness of compiler with error handling

```
            pick-witnesses n1 n2 for lemma spec-lemma
              (!chain
                [((compile (e1 / e2)) ++ prog wrt' stack)
                = ((compile e2 ++ compile e1 ++ [div]) ++ prog wrt' stack)
                    [compiler-def]
                = (compile e2 ++ compile e1 ++ [div] ++ prog wrt' stack)
                    [List.join.Associative]
                = ([div] ++ prog wrt' n1::n2::stack)
                    [ih1 ih2]
                = (div::prog wrt' n1::n2::stack)
                    [List.join.left-singleton]
                = (prog wrt' (n1 N./ n2)::stack)
                    [exec'-def]
                = (prog wrt' n::stack)
                    [spec-lemma]])
    }
```

Correctness of compiler with error handling

The main correctness proof now becomes:

```
define main-correctness-theorem' :=
by-induction (forall e . correctness' e) {
    (const k) => (!basis-step k)
  | (e as (_ sum _))  => (!istep e)
  | (e as (_ diff _)) => (!istep e)
  | (e as (_ prod _)) => (!istep e)
  | (e as (_ quot _)) => (!istep-div e)
  }
```

Correctness of compiler with error handling

And the top-level result can now be derived in a few lines:

```
conclude compiler-correctness' :=
  (forall e n . I' e = SOME n ==> run-vm' compile e = SOME n)
    pick-any e:Exp n:N
      assume hyp := (I' e = SOME n)
        (!chain
          [(run-vm' compile e)
        = (compile e wrt' [])          [vm-def']
        = (compile e ++ [] wrt' [])    [List.join.right-empty]
        = ([] wrt' n::[])              [main-correctness-theorem']
        = (SOME n)                     [exec'-def]])

Theorem: (forall ?e:Exp
           (forall ?n:N
             (if (= (TC.I' ?e:Exp)
                    (SOME ?n:N))
                 (= (run-vm' (TC.compile ?e:Exp))
                    (SOME ?n:N)))))
```

Correctness of compiler with error handling

- The correctness result we have established for the error-handling version of the compiler is only partial.
- It says only that *if* the evaluation of an expression e produces *some* natural number n, then running the virtual machine on the program obtained by compiling e will result in the same n.
- It does not, however, say anything about the relationship between the interpreter and the virtual machine for those expressions e whose evaluation fails.
- If the evaluation of e produces `NONE`, then what happens if we compile e and run the virtual machine on the output program?
- Will we also get `NONE`, or might we end up with some number on the top of the stack?

Correctness of compiler with error handling

- The result we have so far does not address the previous question.
- To do so, we must also prove the following:

```
(forall e . I' e = NONE ==> run-vm' compile e = NONE). (3)
```

The main inductive property to be derived follows (called `correctness-conv`, for the converse direction):

```
define (correctness-conv e) :=
  (forall prog stack .
    I' e = NONE ==> compile e ++ prog wrt' stack = NONE)
```

Testing the conjecture that every expression has this property:

```
define conjecture := (forall e . correctness-conv e)

> (falsify conjecture 30)
Term: 'failure
```

Correctness of compiler with error handling

Once we have shown that every expression e has the property `correctness-conv`, let us call this `main-correctness-theorem-conv`, then (3) can be derived as follows:

```
conclude compiler-correctness-conv :=
  (forall e . I' e = NONE ==> run-vm' compile e = NONE)
    pick-any e:Exp
      assume (I' e = NONE)
        (!chain
          [(run-vm' compile e)
         = (compile e wrt' [])          [vm-def']
         = (compile e ++ [] wrt' [])    [List.join.right-empty]
         = NONE                         [main-correctness-theorem-conv]])
```

Correctness of compiler with error handling

- Now `main-correctness-theorem-conv` is again derived by induction on the structure of expressions, with the proof following the same outline as the inductive proof of the first direction of the correctness result.
- We write one method for the basis step, parameterized by the natural number that is the argument to `const`, and one method for the three inductive-step cases. Division is again best handled by a different method.
- The basis step is somewhat different, in that the main assumption of the property we are now trying to prove, namely, that the evaluation of the expression produces `NONE`, is incompatible with constant expressions, as these always produce some natural number, never `NONE`.

Correctness of compiler with error handling

Therefore, the assumption yields an inconsistency with the definition of the interpreter and the result follows vacuously:

```
define (basis-step-conv k) :=
  pick-any prog:Program stack:Stack
    assume hyp := (I' const k = NONE)
      let {_ := (!absurd
                  (!chain-> [true
                             ==> (I' const k = SOME k)   [I'-def]])
                  (!chain-> [hyp
                             ==> (I' const k =/= SOME k) [option-results]]))}
        (!from-false (compile const k ++ prog wrt' stack = NONE))
```

Correctness of compiler with error handling

It will help to have a parameterized lemma, encoded as a method analogous to the `get-lemma` method, to draw some useful conclusions from the assumption that the evaluation of an expression produces `NONE`.

Let us call this one `get-lemma-conv`:

```
define get-lemma-conv :=
  method (premise)
    match premise {
      (= (I' (_ e1 e2)) NONE) =>
        (!force (I' e2 = NONE | I' e2 =/= NONE & I' e1 = NONE))
    }
```

The conclusion we draw here is that either the evaluation of the right subexpression `e2` fails, or else the evaluation of `e2` does not fail but that of the left subexpression (`e1`) does.

Correctness of compiler with error handling

A similar lemma will come in handy specifically for division: If we know that the evaluation of `(e1 quot e2)` fails, then we may conclude one of the following: Either the evaluation of `e1` or `e2` fails, or else the value of the right subexpression, `e2`, is `zero`:

```
define get-lemma-conv-div :=
  method (premise)
    match premise {
      (= (I' (quot e1 e2)) NONE) =>
        (!force (I' e1 = NONE | I' e2 = NONE | I' e2 = SOME zero))
    }
```

Correctness of compiler with error handling

After these two lemma-producing methods have been defined, methods `istep-conv` and `istep-conv-div` can be written to appear in the inductive proof of `main-correctness-theorem-conv` below:

```
define main-correctness-theorem-conv :=
  by-induction (forall e . correctness-conv e) {
    (const k) => (!basis-step-conv k)
  | (e as (_ sum _))  => (!istep-conv e)
  | (e as (_ diff _)) => (!istep-conv e)
  | (e as (_ prod _)) => (!istep-conv e)
  | (e as (_ quot _)) => (!istep-conv-div e)
  }
```

The two inductive methods (one for the division and one for the remaining three expression constructors) are left as exercises.

Correctness of compiler with error handling

The final correctness theorem follows:

```
conclude compiler-correctness'' :=
    (forall e . run-vm' compile e = I' e)
  pick-any e
  (!cases
    (!uspec opt-no-junk (I' e))
    assume (I' e = NONE)
      (!chain [(run-vm' compile e) = NONE   [compiler-correctness-conv]
                                   = (I' e) [(I' e = NONE)]])
    assume (exists n . I' e = SOME n)
      pick-witness n' for (exists n . I' e = SOME n) some-n'
        (!chain [(run-vm' compile e) = (SOME n') [compiler-correctness']
                                     = (I' e)    [some-n']]))

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

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 options to model errors and 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, ordered pairs, sets, and maps.