From Fixed Expressions to Formulas

Variables
So far, we have looked at type-checking, evaluating (in a value domain) and calculating (in abstract syntax) expressions. However, "programming" makes sense when one performs the same calculation on *different* inputs?! In that sense, our toy language compiler or a definitional interpreter are programs that can work on different expressions as input — however, the toy language is not yet a "programming language". The next step we will take is to make some parts of expressions vary. Mathematicians do this all the time: for example, they calculate a *formula* $b^2 - 4ac$ for different values of a, b, c. The expressions a, b, c are called input *variables* since they can take on *various* values.

So our first step is to extend the toy language with a new case of expressions — namely, variables. Assume that we have a (denumerable) set of identifiers $\mathscr{X}$ (assume also that these identifiers are different from other symbols in the language), with $x, y, z, x_1, y_1, \ldots \in \mathscr{X}$ *representing* typical variables. (We call $x, y, z, x_1, y_1, \ldots$ "meta-variables" — the actual variables in the language will be particular strings).

Following an "abstract grammatical notation" to characterise the inductively defined set of expressions (i.e., abstract syntax), we can write
$\underline{E} \in Exp \ ::= \ \underline{N} \mid \underline{\mathrm{T}} \mid \underline{\mathrm{F}} \mid \underline{x} \mid \underline{E_1 + E_2} \mid \underline{E_1 * E_2} \mid \underline{E_1 \wedge E_2} \mid \underline{E_1 \vee E_2} \mid \underline{\neg E_1} \mid \underline{E_1 = E_2} \mid \underline{E_1 > E_2}$

We extend the OCaml encoding of the language
```
    type exp = Num of int | Bl of myBool
             | V of string (* variables *)
             |  Plus of exp * exp | Times of exp * exp
             |  And of exp * exp | Or of exp * exp | Not of exp
             |  Eq of exp * exp | Gt of exp * exp
    ;;
```
by redefining the type `exp` to include another case (namely, variables) represented using a constructor `V` that takes a string (the name of a particular variable in the toy language) as an argument.

The functions `ht` and `size` are amended as follows, mapping all variables to have height 0, and size 1.

```
    let rec ht e = match e with
        Num n  -> 0
      | Bl b -> 0
      | V x -> 0
      | Plus (e1, e2)   -> 1 + (max (ht e1) (ht e2))
      | Times (e1, e2)   -> 1 + (max (ht e1) (ht e2))
      | And (e1, e2)  -> 1 + (max (ht e1) (ht e2))
      | Or (e1, e2)  -> 1 + (max (ht e1) (ht e2))
      | Not e1 -> 1 + (ht e1)
      | Eq (e1, e2)  -> 1 + (max (ht e1) (ht e2))
      | Gt(e1, e2)  -> 1 + (max (ht e1) (ht e2))
    ;;
```

and

```
let rec size e = match e with
    Num n  -> 1
  | Bl b -> 1
  | V x -> 1
  | Plus (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Times (e1, e2)  -> 1 +  (size e1) + (size e2)
  | And (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Or (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Not e1 -> 1 + (size e1)
  | Eq (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Gt(e1, e2)  -> 1 +  (size e1) + (size e2)
;;
```

Types and typing rules

Assume we have a set Typ of types. Let $\tau, \tau_1, \tau' \in Typ$ *represent* typical types (again, τ, τ_1, τ' are "meta-variables" ranging over types). So far, we have considered IntT and BoolT as members of Typ. (This will be extended as we proceed later). We have seen earlier typing rules that associate numeric expressions with type IntT and boolean expressions with type BoolT. But what type should we give variables?

Type Assumptions. A *typing assumption* $\Gamma \in \mathscr{X} \rightarrow_{fin} Typ$ is a finite-domain function from variables to types, that is, it associates a type with any variable in its domain.
If Γ is a type assumption, $\underline{x} \in \mathscr{X}$ a variable, and $\tau \in Typ$ a type, we write $\Gamma[\underline{x} : \tau]$ to denote the type assumption that associates the type τ to variable $\underline{x}$ and for other variables in $dom(\Gamma)$, associates them to types exactly as Γ would.
This notion generalises as follows:
If Γ, Γ_1 are type assumptions, then $\Gamma[\Gamma_1]$ denotes the type assumption defined as

$\Gamma[\Gamma_1](\underline{x}) = \Gamma_1(\underline{x})$ if $x \in dom(\Gamma_1)$;
$\Gamma[\Gamma_1](\underline{x}) = \Gamma(\underline{x})$ if $x \in dom(\Gamma) - dom(\Gamma_1)$;
and *undefined* if $\underline{x} \notin dom(\Gamma) \cup dom(\Gamma_1)$.

OK. but WHY??
kya fayda iska

(This is in fact, the standard notion of one finite domain function being augmented by another.) ???

The "has type" relation is now modified to carry a type assumption, written to the left of the "turnstile", to handle the presence of variables within expressions, and by adding a rule to deal with the base case of variables. Note that each statement is modified to read as $\Gamma \vdash \_ : \_$

(NumT) $\dfrac{}{\Gamma \vdash \underline{N} : \underline{\text{IntT}}}$ All numerals $\underline{N}$ have type IntT for *any* Γ

before it was only $\vdash N$: IntT, B: BoolT, now Γ is added

(BoolT) $\dfrac{}{\Gamma \vdash \underline{B} : \underline{\text{BoolT}}}$ All boolean constants $\underline{B}$ have type BoolT for *any* Γ

(VarT) $\dfrac{}{\Gamma \vdash \underline{x} : \Gamma(\underline{x})}$ A variable has the type it is *assumed* to have.

(PlusT) $\dfrac{\Gamma \vdash \underline{E_1} : \underline{\text{IntT}} \qquad \Gamma \vdash \underline{E_2} : \underline{\text{IntT}}}{\Gamma \vdash \underline{E_1 + E_2} : \underline{\text{IntT}}}$

All addition expressions $E_1 + E_2$ have type IntT, provided the subexpressions E_1 and E_2 both have type IntT under the *same* type assumptions Γ.

$$(\textbf{TimesT})\ \frac{\Gamma \vdash E_1 : \text{IntT} \qquad \Gamma \vdash E_2 : \text{IntT}}{\Gamma \vdash E_1 * E_2 : \text{IntT}}$$

All multiplication expressions $E_1 * E_2$ have type IntT, provided the subexpressions E_1 and E_2 both have type IntT under the *same* type assumptions Γ.

$$(\textbf{NotT})\ \frac{\Gamma \vdash E_1 : \text{BoolT}}{\Gamma \vdash \neg E_1 : \text{BoolT}}$$

All negation expressions $\neg E_1$ have type BoolT, provided the subexpressions E_1 have type BoolT under the *same* type assumptions Γ.

$$(\textbf{AndT})\ \frac{\Gamma \vdash E_1 : \text{BoolT} \qquad \Gamma \vdash E_2 : \text{BoolT}}{\Gamma \vdash E_1 \wedge E_2 : \text{BoolT}}$$

All conjunction expressions $E_1 \wedge E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type BoolT under the *same* type assumptions Γ.

$$(\textbf{OrT})\ \frac{\Gamma \vdash E_1 : \text{BoolT} \qquad \Gamma \vdash E_2 : \text{BoolT}}{\Gamma \vdash E_1 \vee E_2 : \text{BoolT}}$$

All disjunction expressions $E_1 \vee E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type BoolT under the *same* type assumptions Γ.

T variable → int / bool

$$(\textbf{EqT})\ \frac{\Gamma \vdash E_1 : \text{IntT} \qquad \Gamma \vdash E_2 : \text{IntT}}{\Gamma \vdash E_1 = E_2 : \text{BoolT}}$$

All numeric equality expressions $E_1 = E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type IntT under the *same* type assumptions Γ.

$$(\textbf{GtT})\ \frac{\Gamma \vdash E_1 : \text{IntT} \qquad \Gamma \vdash E_2 : \text{IntT}}{\Gamma \vdash E_1 > E_2 : \text{BoolT}}$$

All greater-than expressions $E_1 > E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type IntT under the *same* type assumptions Γ.

Modifying the definitional interpreter

How does the definitional interpreter change? Well, what is the value of a variable? Whatever value we give to the variable by a "valuation", i.e., a function $\boxed{\rho \in \mathscr{X} \to \mathbb{V}}$ from variables to values in the set of values $\mathbb{V}$. So each equation for *eval* takes ρ as an additional argument:

$$eval[\![\ \underline{N}\ ]\!]\ \rho = n$$

$$eval[\![\text{T}]\!]\ \rho = true \ \text{ and } \ eval[\![\text{F}]\!]\ \rho = false$$

So ρ tell what values each variable is mapped to.

$\boxed{eval[\![\ \underline{x}]\!]\ \rho = \rho(\underline{x})}$

So use same mappings to find variables in them

$eval[\![\underline{E_1 + E_2}]\!]\ \rho = (eval[\![\underline{E_1}]\!]\ \rho) + (eval[\![\underline{E_2}]\!]\ \rho)$

$eval[\![\underline{E_1 * E_2}]\!]\ \rho = (eval[\![\underline{E_1}]\!]\ \rho) \times (eval[\![\underline{E_2}]\!]\ \rho)$

(where $+, \times$ represent integer addition and multiplication).

$eval[\![\underline{E_1 \wedge E_2}]\!]\ \rho = (eval[\![\underline{E_1}]\!]\ \rho)\ \&\&\ (eval[\![\underline{E_2}]\!]\ \rho)$

$eval[\![\underline{E_1 \vee E_2}]\!]\ \rho = (eval[\![\underline{E_1}]\!]\ \rho)\ \ ||\ \ (eval[\![\underline{E_2}]\!]\ \rho)$

$eval[\![\underline{\neg E_1}]\!]\rho = not\ \ (eval[\![\underline{E_1}]\!]\ \rho)$

(where $\&\&$, $||$, not represent boolean conjunction, disjunction and negation).

$eval[\![\underline{E_1 = E_2}]\!]\ \rho = (eval[\![\underline{E_1}]\!]\ \rho) =^{?} (eval[\![\underline{E_2}]\!]\ \rho)$

$eval[\![\underline{E_1 > E_2}]\!]\ \rho = (eval[\![\underline{E_1}]\!]\ \rho) >^{?} (eval[\![\underline{E_2}]\!]\ \rho)$

(where $=^{?}$, $>^{?}$ represent equality and greater-than comparisons on integers).

The Modified Definitional Interpreter in OCaml

```
let rec eval e rho = match e with
      Num n  -> N n
    | Bl b -> B (myBool2bool b)
    | V x -> rho x
    | Plus (e1, e2)  ->  let N n1 = (eval e1 rho)
                         and  N n2 = (eval e2 rho)
                           in N (n1 + n2)
    | Times (e1, e2)  ->  let N n1 = (eval e1 rho)
                          and  N n2 = (eval e2 rho)
                            in N (n1 * n2)
    | And (e1, e2)  ->  let B b1 = (eval e1 rho)
                        and  B b2 = (eval e2 rho)
                          in B (b1 && b2)
    | Or (e1, e2)  -> let B b1 = (eval e1 rho)
                      and  B b2 = (eval e2 rho)
                        in B (b1 || b2)
    | Not e1 -> let B b1 = (eval e1 rho) in B (not b1)
    | Eq (e1, e2)  -> let N n1 = (eval e1 rho)
                      and  N n2 = (eval e2 rho)
                        in B (n1 = n2)
    | Gt(e1, e2)  -> let N n1 = (eval e1 rho)
                     and  N n2 = (eval e2 rho)
                       in B (n1 > n2)
;;
```

Note that the OCaml interpreter is able to effortlessly infer the type of the modified eval function:

```
val eval : exp -> (string -> values) -> values = <fun>
```

rho

Big Step (Natural, Kahn-style) Operational Semantics

The big-step (Kahn-style) operational semantics also needs to be modified. The calculates relation now needs to return an answer when the input expression is a variable. What answer? Whatever answer the variable is bound to. So we need a *data structure that associates variables to canonical answers*. Let us call this a *"table"*, which is nothing but a *finite-domain function* $\gamma \in \mathcal{X} \rightarrow_{fin} Ans$. Why "finite-domain"? Because a calculator has to operate with finite data structures and not mathematical abstractions such as valuations (which can be infinite).

T Variable → Type

Accordingly, we modify the calculates relation by introducing a table in each of the rules. And we add a rule to deal with the case of variables. To highlight that the table does not change during the calculation process, we place the table to the left of a turnstile.

Variable → Ans ⊆ Value

(CalcNum) $\dfrac{}{\gamma \vdash \underline{N} \Longrightarrow \underline{N}}$ for any γ

T : Variable → Type

(CalcBool) $\dfrac{}{\gamma \vdash \underline{B} \Longrightarrow \underline{B}}$ for any γ

(CalcVar) $\dfrac{}{\gamma \vdash \underline{x} \Longrightarrow \gamma(\underline{x})}$ provided $\underline{x} \in dom(\gamma)$

(CalcPlus) $\dfrac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{N_1} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{N_2}}{\gamma \vdash \underline{E_1 + E_2} \Longrightarrow \underline{N}}$ provided $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$

(CalcTimes) $\dfrac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{N_1} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{N_2}}{\gamma \vdash \underline{E_1 * E_2} \Longrightarrow \underline{N}}$ provided $TIMES(\underline{N_1}, \underline{N_2}, \underline{N})$

(CalcNot) $\dfrac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{B_1}}{\gamma \vdash \underline{\neg E_1} \Longrightarrow \underline{B}}$ provided $NOT(\underline{B_1}, \underline{B})$

(CalcAnd) $\dfrac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{B_1} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{B_2}}{\gamma \vdash \underline{E_1 \wedge E_2} \Longrightarrow \underline{B}}$ provided $AND(\underline{B_1}, \underline{B_2}, \underline{B})$

CalcOr) $\dfrac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{B_1} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{B_2}}{\gamma \vdash \underline{E_1 \vee E_2} \Longrightarrow \underline{B}}$ provided $OR(\underline{B_1}, \underline{B_2}, \underline{B})$

(CalcEq) $\dfrac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{N_1} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{N_2}}{\gamma \vdash \underline{E_1 = E_2} \Longrightarrow \underline{B}}$ provided $EQ(\underline{N_1}, \underline{N_2}, \underline{B})$

(CalcGt) $\dfrac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{N_1} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{N_2}}{\gamma \vdash \underline{E_1 > E_2} \Longrightarrow \underline{B}}$ provided $GT(\underline{N_1}, \underline{N_2}, \underline{B})$

Note that in (**CalcVar**), *no* answer is returned if $x \notin dom(\gamma)$. The calculator gets stuck!

What about Soundness and Completeness of the Calculator with respect to the Definitional Interpreter? How do those statements change?

We need a notion of a table and a valuation *agreeing with each other*. That is, for *every* $\underline{x} \in dom(\gamma)$: $eval[\![\underline{\gamma(\underline{x})}]\!]\ \rho = \rho(\underline{x})$

With this assumption on γ and ρ, it is fairly easy to state and prove Soundness.

Completeness is somewhat harder, and requires an additional assumption — namely that $\underline{x} \in vars(\underline{E}), \underline{x} \in dom(\gamma)$, where $vars(\underline{E})$ denotes the set of variables appearing in expression $\underline{E}$. This condition ensures that the calculator does not get stuck because a variable cannot be looked up in the table.

Exercise: Define the function $vars(\underline{E})$. Note the similarity in structure to `ht`, `size`, and `eval`.

Type Preservation (version 2)

The type preservation theorem requires some additional conditions (apart from the assumption that elementary operations are type-sound).

We say that a table γ is *type-consistent* with a type assumption Γ if for *every* $\underline{x} \in dom(\gamma)$: a
b $\underline{x} \in dom(\Gamma)$ and $\Gamma \vdash \gamma(\underline{x}) : \Gamma(\underline{x})$ c That is, the answer associated with any variable in a table is indeed of the same type associated with it by the type assumption.

Theorem (Type Preservation under $\gamma \vdash \underline{E} \Longrightarrow \underline{A}$)
For all expressions $\underline{E}, \underline{A}$,
for all type assumptions Γ,
for all tables γ type-consistent with Γ,
for all types $\underline{T}$,
if $\Gamma \vdash \underline{E} : \underline{T}$ and $\gamma \vdash \underline{E} \Longrightarrow \underline{A}$, then $\Gamma \vdash \underline{A} : \underline{T}$

Proof (By Induction on the structure/*ht* of $\underline{E}$).
Base cases ($ht(\underline{E}) = 0$)
Subcases ($\underline{E} \equiv \underline{N}$) and ($\underline{E} \equiv \underline{B}$) are essentially unchanged.
There is a new base case: $\underline{E} \equiv \underline{x}$.
Assume $\Gamma \vdash \underline{E} : \underline{T}$. Therefore $\underline{T} = \Gamma(\underline{x})$. (The case of $\underline{x} \notin dom(\Gamma)$ cannot arise from the assumption).
Now $\gamma \vdash \underline{x} \Longrightarrow \gamma(\underline{x})$. Since γ is assumed type-consistent with Γ, $\Gamma \vdash \gamma(\underline{x}) : \Gamma(\underline{x})$.
So $\Gamma \vdash \underline{A} : \underline{T}$

The **Induction Hypothesis** is a suitably modified version of the earlier **IH,** and the cases in the **Induction Step** ($ht(\underline{E}) = 1 + k$) are more or less the same as before, with the appropriate changes.

Exercise: Complete this proof.

Exercise: Encode the type-checking relation $\Gamma \vdash \underline{E} : \underline{T}$ in PROLOG as a predicate `hastype(G,E,T).`

Compilation and execution on a Stack Machine

The stack machine now needs to be modified to incorporate an additional component, namely the table. The configurations are now triples — a table, a stack of values and a code list.

The opcodes need only a small extension however — an opcode to look up a variable in the table. (In practice, we get rid of the variables and use some address/indexing mechanism).

```
type opcode = LDN of int | LDB of bool | LOOKUP of string
              | PLUS | TIMES | AND | OR | NOT | EQ | GT;;
```

The `compile` function therefore has minimal changes == the inclusion on a line for compiling variables

```
let rec compile e = match e with
    Num n  -> [ LDN n ]
  | Bl b ->   [LDB (myBool2bool b) ]    (* Constants *)
  | V x ->   [LOOKUP x]    (* Variables *)
  | Plus (e1, e2)  ->  (compile e1) @ (compile e2) @ [PLUS]
  | Times (e1, e2)  ->  (compile e1) @ (compile e2) @ [TIMES]
  | And (e1, e2)  ->  (compile e1) @ (compile e2) @ [AND]
  | Or (e1, e2)  -> (compile e1) @ (compile e2) @ [OR]
  | Not e1 -> (compile e1) @ [NOT]
  | Eq (e1, e2)  -> (compile e1) @ (compile e2) @ [EQ]
  | Gt(e1, e2)  -> (compile e1) @ (compile e2) @ [GT]
;;
```

The stack machine, now endowed with a table in its configurations, needs to specify how the `LOOKUP(x)` opcode is executed. Otherwise, it is substantially the same (other than now containing a table component). Out of indolence, we have represented a table as a function from strings to *values* (and not answers).

```
exception Stuck of (string -> values) * values list * opcode
list);;

let rec stkmc g s c = match s, c with
    v::_, [ ] -> v  (* no more opcodes, return top *)
  | s, (LDN n)::c' -> stkmc g ((N n)::s) c'
  | s, (LDB b)::c' -> stkmc g ((B b)::s) c'
  | s, (LOOKUP x)::c' -> stkmc g ((g x)::s) c'
  | (N n2)::(N n1)::s', PLUS::c' -> stkmc g (N(n1+n2)::s') c'
  | (N n2)::(N n1)::s', TIMES::c' -> stkmc g (N(n1*n2)::s')
c'
```

```
  | (B b2)::(B b1)::s', AND::c' -> stkmc g(B(b1 && b2)::s')
c'
  | (B b2)::(B b1)::s', OR::c' -> stkmc g (B(b1 || b2)::s')
c'
  | (B b1)::s', NOT::c' -> stkmc g (B(not b1)::s') c'
  | (N n2)::(N n1)::s', EQ::c' -> stkmc g (B(n1 = n2)::s') c'
  | (N n2)::(N n1)::s', GT::c' -> stkmc g (B(n1 > n2)::s') c'
  | _, _ -> raise (Stuck  (g, s, c))
;;
```

The exception `Stuck` now takes 3 arguments, namely the table, stack and opcode list. The only new line is

```
  | s, (LOOKUP x)::c' -> stkmc g ((g x)::s) c'
```

where the value obtained from the table, namely `(g x)`, is pushed onto the stack.