Models of Computation

There are many models of computation. We will see here, but learn in greater detail in a "Theory of Computation" course, that some of them may be more *expressive* than others, while others may turn out to have equal expressive power. Some models allow us to only write terminating programs (we have met the primitive recursive programs earlier). (The word "expressiveness" can be used in two senses: one a formal sense in characterising what set of computations are possible, and the other a more informal sense, where we consider how easy or elegant a language or model is for expressing a particular concept.) We will see some very useful models that are not very expressive, but give us very strong guarantees about properties such as termination and complexity of computations possible within them. Within the traditional models of computation, the most expressive ones are the so-called Turing-complete models, which are conjectured to be equally expressive as one another, and which admit non-terminating programs.

However, from a language design perspective, when describing models of computation, we find they have some things in common.

Data:

1. Every model has some elementary types of data; For example, our toy calculator allows us to compute on *integers* and *booleans*. Other types such as *characters*, *strings* and floating point representations of *reals* are examples of elementary data.
2. Most models have *type constructions* that allow us to create more complex data: these are usually based on set-theoretic or type-theoretic constructions, some of which are very generic in nature — for example, given any types, we can form *cartesian products* (which are manifest as *tuples* or — in a named form — as *records)*, (tagged) *disjoint sums* (which are manifest as *variant records* or *union* types) and in the case of functional languages, *function spaces*. (Function types actually exist in every procedural language, but functions or procedures may not be "first-class citizens". More on this later).
3. Many models also support *recursive data types*. We have seen that OCaml allows us to define structures (some of which are generic type constructions) such as *lists* and *trees*, which are recursively defined.
4. Languages such as SML, OCaml and Haskell allow us to define (parametric) *polymorphic type constructions*, such as *lists* of any given type.

We have seen that OCaml supports all the above-listed types, and in addition allows users to define their own data types and type constructions.

Control:

1. Every model of computation is characterised by the elementary operations it supports. For example, operations on integers or on booleans are the elementary operations we have seen in our toy calculator. Imperative languages such as C or Java contain, as a primitive operation, the storing a value in a memory location (this operation is called an "assignment statement").
2. Elementary operations can be combined in some basic ways.
    1. *Sequential composition*: given two operations, we can perform one before the other. In functional languages, the output of one function can be fed as the input to another function. In a language like C, we perform the actions of one statement and its effects are then given to the next statement listed after the semi-colon.
    2. *Conditional composition*: Almost every useful language allows us to perform (different) operations based on a case analysis. The "if-then-else" statement in C,

Python, Pascal or Java is the typical conditional construct. Some languages (OCaml and C) have an "`if _ then _ else _` " or " `_ ? _: _`" construct on expressions.

3. *Repetition*: Most useful languages allow an operation to be repeated. Sometimes the repetition is iterative and definite — as in a "*for loop*". Some models indefinite iterations until some condition is satisfied (or conversely until it a condition no longer holds) — such as "*while loop*". The most general form of repetition is *general recursion*. We also have more restricted versions of recursion (such as "primitive recursion" that we saw earlier).
4. Some languages that support concurrency also have constructs such as *parallel composition*.

Conditional Expressions

We have already introduced the boolean constructors `T` and `F` in our language.
But what use are these canonical answers? We use these two (distinct) simple constructions as case analysis markers. Whenever we introduce constructors, we also can add to our language operations that perform *deconstruction*.

So our next step with the toy language is to extend it with a typical conditional expression. The intuitive idea is to allow the following case-based evaluation — if the condition E_0 evaluates to *true*, then the compound expression returns the value of expression E_1 , else if E_0 evaluates to *false*, then the value of expression E_2 is returned.

We add a new case in the "abstract grammar" of expressions:

$$E \in Exp \ ::= \ N \mid \mathrm{T} \mid \mathrm{F} \mid x \mid \ldots \mid \text{if } E_0 \text{ then } E_1 \text{ else } E_2$$

Expectedly, 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
         |  IfTE of exp * exp *exp
;;
```

by redefining the type `exp` to include the new case represented using a constructor `IfTE` that takes three `exp`s as arguments. (Note that if we redefine a data type, then any functions we have previously defined using this type also need to be redefined.)

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))
  | IfTE(e0, e1, e2) -> 1 + (max (ht e0)
                                  (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)
  | IfTE(e0, e1, e2) -> 1 + (size e0) +
                            (size e1) + (size e2)
;;
```

Typing rules

We extend the typing rules to deal with the new conditional construct.

$$\textbf{(IfT)}\ \frac{\Gamma \vdash E_0 : \underline{\text{BoolT}} \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \underline{\text{if } E_0 \text{ then } E_1 \text{ else } E_2} : \tau}$$

Note that E_0 must have type BoolT under the type assumptions Γ. And under these very same type assumptions Γ, both the expressions E_1 and E_2 must have the same type τ which may be *any* type. This type τ is the type of the expression $\underline{\text{if } E_0 \text{ then } E_1 \text{ else } E_2}$. The intuition is that irrespective of whether E_0 evaluates to *true* or to *false*, the result is guaranteed to be of the same type τ.

Note that since we have introduced you to a construct whose type depends on the types of its component subexpressions (and agreement between the types of two different component subexpressions), we may also use this opportunity to generalise the typing rule for the equality operation, so that it may apply not only on numeric expressions but also on expressions of any type.

$$\textbf{(EqT')}\ \frac{\Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \underline{E_1 = E_2} : \underline{\text{BoolT}}}$$

Extending the definitional interpreter

We add a new case to the definitional interpreter, which is a definition by cases.

$$eval[\![\text{if } E_0 \text{ then } E_1 \text{ else } E_2]\!]\ \rho = eval[\![\ E_1\ ]\!]\ \rho \text{ if } eval[\![\ E_0\ ]\!]\ \rho = true$$

$$eval[\![\text{if } E_0 \text{ then } E_1 \text{ else } E_2]\!]\ \rho = eval[\![\ E_2\ ]\!]\ \rho \text{ if } eval[\![\ E_0\ ]\!]\ \rho = false$$

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)
  | IfTE(e0, e1, e2) -> let B b0 = (eval e0 rho)
                          in if b0 then (eval e1 rho)
                                   else (eval e2 rho)
;;
```

Note that we use OCaml's built-in conditional form "**`if then else`**" to perform the case analysis.

Big Step (Natural, Kahn-style) Operational Semantics

Let us specify the big-step (Kahn-style) operational rules for the new construct.
Note that we will need two rules: one for the case where E_0 calculates to `T` and another where E_0 calculates to `F`. The point to note here is that the calculation for construct $\text{if } E_0 \text{ then } E_1 \text{ else } E_2$ does not eagerly calculate both E_1 and E_2; instead, the result of expression E_1 (respectively E_2) is calculated depending on the outcome of calculating E_0.

(CalcIfTE-T) $\dfrac{\gamma \vdash \underline{E_0} \Longrightarrow \underline{\mathrm{T}} \quad \gamma \vdash \underline{E_1} \Longrightarrow \underline{A}}{\gamma \vdash \underline{\text{if } E_0 \text{ then } E_1 \text{ else } E_2} \Longrightarrow \underline{A}}$

(CalcIfTE-F) $\dfrac{\gamma \vdash \underline{E_0} \Longrightarrow \underline{\mathrm{F}} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{A}}{\gamma \vdash \underline{\text{if } E_0 \text{ then } E_1 \text{ else } E_2} \Longrightarrow \underline{A}}$

What about Soundness and Completeness of the calculator with respect to the Definitional Interpreter? Both these theorems continue to hold!

Exercise: Prove the new induction step in the Soundness Theorem.

Exercise: Prove the new induction step in the Completeness Theorem.

The Type Preservation theorem also continues to hold!

Exercise: Prove the new induction step in the Type Preservation Theorem.

Exercise: Extend the encoding of the type-checking relation $\Gamma \vdash \underline{E} : \underline{T}$ in PROLOG by incorporating conditional expression in the predicate `hastype(G,E,T)`.

Compilation and execution on a Stack Machine

The stack machine now needs to be modified to deal with conditional execution.

We need a new opcode: Let us call it `COND`. `COND` needs to take two arguments, both of them opcode lists — the first opcode list is executed if the top of the stack has the value *true*, and the second if the top of the stack is *false*.

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

The `compile` function has a new case, which results in a call to `compile` with argument `e0` followed by the single opcode `COND` which has as its two arguments the results of recursive calls to `compile` on `e1` and `e2`.

```
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]
  | IfTE(e0, e1, e2) -> (compile e0) @
                              [COND(compile e1, compile e2)]
;;
```

The stack machine has two new rules for `COND(c1, c2)` — depending on whether the value at the top of the stack is *true* or *false*. Note that the tail recursive call to `stkmc` executes the case-appropriate code `c1` or `c2`, followed by the remaining code `c'`.

Note that in actual machines we do not have such a complex opcode as `COND`, which would be of varying size depending on the structural complexity of the subexpressions `e1` and `e2`. Instead, the compiler will generate code for `e1` and `e2`, and lay them out in some way (usually in sequence) and will insert a jump instruction based on the value on the stack that moves the control (program counter) to the appropriate case to execute, as well as an instruction after the subexpression `e1` (and possibly also after subexpression `e2`) to jump to a point where the execution of the code for the subsequent program `(c')` can be rejoined. It is instructive to see this drawn as a flowchart.

```
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'
  | (B true)::s', COND(c1,c2)::c' -> stkmc g s' (c1 @ c')
  | (B false)::s', COND(c1,c2)::c' -> stkmc g s' (c2 @ c')
  | _, _ -> raise (Stuck  (g, s, c))
;;
```

Pairing and Projection

We now explore an extension to the data types supported in the toy language. Suppose we allow the language to include (binary) cartesian products. This will result in a construction called "pairing" (and so include a corresponding constructor in the OCaml rendition). Most languages just use parentheses to denote tuples, but we will use a stylised form with angular brackets in our *abstract syntax*. And for this constructor, we have two corresponding *deconstructions* — the *projection* of the *first* and of the *second* component of an expression that results in a pair.

Abstract Syntax

$$\underline{E} \in Exp \;::=\; \underline{N} \mid \ldots \mid \underline{\langle\, E_1,\, E_2 \,\rangle} \mid \underline{\mathrm{fst}\ E'} \mid \underline{\mathrm{snd}\ E'}$$

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
         |  IfTE of exp * exp * exp
         |  Pair of exp * exp
         |  Fst of exp  | Snd of exp
;;
```

by redefining the type `exp` to include the new cases represented using a constructors `Pair` that takes two `exp`s as arguments, and `Fst` and `Snd that` operate on a single argument of type `exp`. (Note: We should redefine extend and redefine all functions using the type `exp`).

Types and typing rules

We extend the set Typ of types. Let $\tau, \tau_1, \tau' \in Typ$ *represent* typical types.

$$\tau, \tau_1, \tau' \in Typ \;::=\; \underline{\mathrm{IntT}} \;\mid\; \underline{\mathrm{BoolT}} \;\mid\; \tau_1 \times \tau_2$$

The typing rules are:

$$\textbf{(PairT)}\ \frac{\Gamma \vdash \underline{E_1} : \tau_1 \quad \Gamma \vdash \underline{E_2} : \tau_2}{\Gamma \vdash \underline{\langle\, E_1,\, E_2 \rangle} : \tau_1 \times \tau_2}$$

$$\textbf{(FstT)}\ \frac{\Gamma \vdash \underline{E'} : \tau_1 \times \tau_2}{\Gamma \vdash \underline{\mathrm{fst}\ E'} : \tau_1} \qquad \textbf{(SndT)}\ \frac{\Gamma \vdash \underline{E'} : \tau_1 \times \tau_2}{\Gamma \vdash \underline{\mathrm{snd}\ E'} : \tau_2}$$

Note that in the antecedent (numerator) of the typing rules for the projections, we do not assume that the expression from which we are projecting is necessarily in the form of a pair $\underline{\langle\, E_1,\, E_2 \,\rangle}$.

Extending the definitional interpreter

We first observe that this extension now admits not only pairs of integers and pairs of booleans, but pairs of integers and booleans, and pairs of pairs of integers and booleans, and pairs consisting of an integer and a pair of booleans, and so on. In other words, the set of values now admits a denumerable number of different cases.

$$\mathbb{V} = \mathbf{2} \cup \mathbb{Z} \cup (\mathbb{V} \times \mathbb{V})$$

For the OCaml encoding of values, we introduce a new constructor (`P`) that takes a pair of type `values`.

```
type values = N of int | B of bool | P of values * values;;

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)
  | IfTE(e0, e1, e2) -> let B b0 = (eval e0 rho)
                        in if b0 then (eval e1 rho)
                                 else (eval e2 rho)
  | Pair(e1, e2) -> let v1 = (eval e1 rho)
                    and v2 = (eval e2 rho)
                     in P(v1, v2)
  | Fst(e0) -> let P(v1,v2) = (eval e0 rho)
               in v1
  | Snd(e0) -> let P(v1,v2) = (eval e0 rho)
               in v2
;;
```

Big Step (Natural, Kahn-style) Operational Semantics

Let us specify the big-step (Kahn-style) operational rules for the new constructs.
Note that we will need one rule for pairing, and two rules for the projections (one for each).

$$\textbf{(CalcPair)}\ \frac{\gamma \vdash \underline{E_1} \Longrightarrow \underline{A_1} \quad \gamma \vdash \underline{E_2} \Longrightarrow \underline{A_2}}{\gamma \vdash \underline{\langle\ E_1,\ E_2\ \rangle} \Longrightarrow \underline{\langle\ A_1,\ A_2\ \rangle}}$$

$$\textbf{(CalcFst)}\ \frac{\gamma \vdash \underline{E_0} \Longrightarrow \underline{\langle\ A_1,\ A_2\ \rangle}}{\gamma \vdash \underline{\text{fst}\ E_0} \Longrightarrow \underline{A_1}} \qquad \textbf{(CalcSnd)}\ \frac{\gamma \vdash \underline{E_0} \Longrightarrow \underline{\langle\ A_1,\ A_2\ \rangle}}{\gamma \vdash \underline{\text{snd}\ E_0} \Longrightarrow \underline{A_2}}$$

What about Soundness and Completeness of the Calculator with respect to the Definitional Interpreter? They are preserved. Again!

Exercise: Prove the new induction steps in the Soundness Theorem.

Exercise: Prove the new induction steps in the Completeness Theorem.

The Type Preservation theorem also continues to hold!!

Exercise: Prove the new induction steps in the Type Preservation Theorem.

Exercise: Extend the encoding of the type-checking relation $\Gamma \vdash \underline{E} : \underline{T}$ in PROLOG by incorporating pairing and projections in a predicate `hastype(G,E,T).`

Exercise: (i) Define new opcodes for pairing and the two projection functions.
(ii) Extend the `compile` function to handle the new constructs.
(iii) Extend the `stkmc` function to handle the new opcodes.