Extending Bindex

Programming language designers often want to experiment with a language by modifying a feature or adding new features. Here, we study some changes to BINDEX.

1 Call-by-Value vs. Call-by-Name

Reconsider the BINDEX substitution model evaluation clause for `bind`:

```
and eval exp =
  match exp with
    ⋮
  | Bind(name,defn,body) ->
      eval (subst1 (Lit (eval defn)) name body)
```

This clause evaluates the definition expresssion `defn` to an integer before substituting the integer for the bound name in the body. This strategy is called **call-by-value** evaluation because each definition expression must first be evaluated to a value before any other evaluation can take place. For example, in the expression

```
(bind c (/ 5 0) 17)
```

the call-by-value strategy results in a divide-by-zero error even though the `c` is never used in the body expression.

There is an alternative strategy, **call-by-name** evaluation, in which the *unevaluated* definition expression is substituted for the bound name. It is easy to modify the BINDEX evaluator to express call-by-name evaluation:

```
and eval exp =
  match exp with
    ⋮
  | Bind(name,defn,body) ->
      (* call-by-name evaluation of bind expression *)
      eval (subst1 defn name body)
```

With this interpreter, the expression `(bind c (/ 5 0) 17)` evaluates to `17` without signalling an error, because `(/ 5 0)` is never evaluated. The ALGOL60 language was an influential early language that used call-by-name evaluation.

In call-by-name, the number of times a definition expression is evaluated is the number of times it appears in the body. For instance, in

```
(bind a (+ 1 2) (* a a))
```

the addition `(+ 1 2)` will be evaluated twice in call-by-name but only once in call-by-value. For reasons of efficiency, most modern languages employ the call-by-value strategy. An important exception is HASKELL, which uses a modified version of call-by-name known as **call-by-need**. Call-by-need evaluates a definition expression *at most* once, but doesn't evaluate it if it is not used. Will will study call-by-need evaluation later in the semester.

2 `sigma`: A Summation Construct

We now consider extending BINDEX with the following construct:

> **(sigma** I_{var} E_{lo} E_{hi} E_{body}**)**
> Assume that I_{var} is a variable name, E_{lo} and E_{hi} are expressions denoting integers that are not in the scope of I_{var}, and E_{body} is an expression that is in the scope of *var*. Returns the sum of E_{body} evaluated at all values of the index variable I_{var} ranging from the integer value of E_{lo} up to the integer value of E_{hi}, inclusive. This sum would be expressed in traditional mathematical summation notation as:
>
> $$\sum_{I_{var}=E_{lo}}^{E_{hi}} E_{body}.$$
>
> If the value of E_{lo} is greater than that of E_{hi}, the sum is 0.

Fig. 1 show some examples using `sigma`.

Mathematical Notation	BINDEX **Notation**	**Value**
$\sum_{i=3}^{7} i$	`(sigma i 3 7 i)`	$3+4+5+6+7=25$

$\sum_{j=1+2}^{2*3} j^2$	`(sigma j (+ 1 2)`
	`(* 2 3)`
	`(* k k))`
		$3^2+4^2+5^2+6^2=86$

$\sum_{j=5}^{1} j^2$	`(sigma j 5 1 (* k k))`	0

$\sum_{i=2}^{5}\sum_{j=i}^{4} i \cdot j$	`(sigma i 2 5`
	`(sigma j i 4`
	`(* i j)))`
		$2\cdot2+2\cdot3+2\cdot4+3\cdot3+3\cdot4+4\cdot4=55$

$\sum_{i=\sum_{k=1}^{3} k^2}^{\sum_{j=1}^{5} j} i$	`(sigma i (sigma k 1 3`
	`(* k k))`
	`(sigma j 1 5 j)`
	`i)`
		$\sum_{i=(1^2+2^2+3^2)}^{1+2+3+4+5} = \sum_{i=14}^{15} = 14+15=29$

Figure 1: Examples of the `sigma` construct.

What changes must be made to the BINDEX implementation in order to add the `sigma` construct? We consider a minimal set of changes. (We will ignore some changes, such as extending the `fold` and `uniquify` functions.)

1. Extend the `exp` data type to include `sigma`:

```
and exp = ...
  | Sigma of var * exp * exp * exp (* name * lo * hi * body *)
```

2. Extend `sexpToExp` to parse `sigma`:

```
and sexpToExp sexp =
  match sexp with
    ⋮
  | Seq [Sym "sigma"; Sym name; lox; hix; bodyx] ->
      Sigma (name, sexpToExp lox, sexpToExp hix, sexpToExp bodyx)
```

3. Extend `expToSexp` to unparse `sigma`:

```
and expToSexp e =
  match e with
    ⋮
  | Sigma(name,lo,hi,body) ->
      Seq [Sym "sigma"; Sym name; expToSexp lo; expToSexp hi; expToSexp body]
```

4. Extend `freeVarsExp` to calculate the free variables of a `sigma` expression (necessary in order for `varCheck` to work):

```
and freeVarsExp e =
  match e with
    ⋮
  | Sigma(name,lo,hi,body) ->
      S.union (S.diff (freeVarsExp body)
                      (S.singleton name))
              (S.union (freeVarsExp lo)
                       (freeVarsExp hi))
```

5. Extend the environment model `eval` function to handle `sigma`:

```
and eval exp env =
  match exp with
    ⋮
  | Sigma(name,lo,hi,body) ->
      foldr (+) 0
        (map (fun i -> eval body (Env.bind name i env))
             (range (eval lo env) (eval hi env)))
```

   There are many ways to perform the summation, but `range`, `map`, and `foldr` are an elegant approach. Note how the evaluation of the `sigma` body expression at a particular index variable `i` is expressed via:

```
(fun i -> eval body (Env.bind name i env))
```

   Only `body` is evaluated in an extended environment, because it is the only subexpression in the scope of the `sigma`-bound name.

6. Extend the `subst` function to handle `sigma` expressions (necessary for the substitution model):

```
let rec subst exp env =
  match exp with
    ⋮
  | Sigma(name,lo,hi,body) ->
      let name' = StringUtils.fresh name in
        Sigma(name',
              subst lo env,
              subst hi env,
              subst (rename1 name name' body) env)
```

7. Extend the substitution model `eval` function to handle `sigma`:

```
and eval exp =
  match exp with
    ⋮
  | Sigma(name,lo,hi,body) ->
      foldr (+) 0
        (map (fun i -> eval (subst1 (Lit i) name body))
             (range (eval lo) (eval hi)))
```

   This is similar to the environment model, except that substitution is use to associate `i` with the index variable in the body expression.

3 Multiple Bindings: `bindpar` and `bindseq`

To explore multiple bindings in BINDEX, we consider extending BINDEX with two constructs that allow multiple bindings:

1. `(bindpar ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`)) `E_{body}`)` binds the names $I_{name_1} \ldots I_{name_n}$ to the values of the expressions $E_{defn_1} \ldots E_{defn_n}$, where these values are determined *in parallel*: all definition expressions are evaluated in the same environment in which the `bindpar` itself is evaluated. The result of the `bindpar` is the result of evaluating E_{body} in an environment that extends the current environment with bindings between all the names and the values of their respective definitions.

2. `(bindseq ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`)) `E_{body}`)` binds the names $I_{name_1} \ldots I_{name_n}$ to the values of the expressions $E_{defn_1} \ldots E_{defn_n}$, where these values are determined *sequentially*: each definition expression is evaluated in the environment of the `bindseq` extended with bindings for the names that appear in the bindings of the `bindseq` that precede it. The result of the `bindseq` is the result of evaluating E_{body} in an environment that extends the current environment with bindings between all the names and the values of their respective definitions.

As an example of the difference between these two binding constructs, consider invoking the following program on the values 10 and 2:

```
(bindex (a b) ; a = 10, b = 2
  (bindpar ((a (/ a b)) ; (/ 10 2) = 5
            (b (- a b))) ; (- 10 2) = 8
    (bindpar ((a (* a b)) ; (* 5 8) = 40
              (b (+ a b))) ; (+ 5 8) = 13
      (+ a b)))) ; (+ 40 13) = 53
```

As indicated by the annotations in the comments, this invocation yields 53 as a result. If we change each `bindpar` to a `bindseq`, the result of invoking the program on the same arguments is 33:

```
(bindex (a b) ; a = 10, b = 2
  (bindseq ((a (/ a b)) ; (/ 10 2) = 5
            (b (- a b))) ; (- 5 2) = 3
    (bindseq ((a (* a b)) ; (* 5 3) = 15
              (b (+ a b))) ; (+ 15 3) = 18
      (+ a b)))) ; (+ 15 18) = 33
```

Many languages have constructs analogous to `bindpar` and `bindseq`. For example, in OCAML, parallel binding is expressed via a `let` followed by any number of `and`s, while sequential binding is expressed by a sequence of `let`s:

```
# let parTest a b =
    let a = a/b
    and b = a-b
          in let a = a*b
          and b = a+b
          in a+b;;
val parTest : int -> int -> int = <fun>

# parTest 10 2;;
- : int = 53

# let seqTest a b =
    let a = a/b in
    let b = a-b in
       let a = a*b in
       let b = a+b in
         a+b;;
val seqTest : int -> int -> int = <fun>

# seqTest 10 2;;
- : int = 33
```

In the SCHEME programming language, `let` is a parallel binding construct but `let*` is a sequential binding construct:

```
(define (par-test a b)
  (let ((a (/ a b))
        (b (- a b)))
    (let ((a (* a b))
          (b (+ a b)))
      (+ a b))))
;Value: par-test

(par-test 10 2)
;Value: 53

(define (seq-test a b)
  (let* ((a (/ a b))
         (b (- a b)))
    (let* ((a (* a b))
           (b (+ a b)))
      (+ a b))))
;Value: seq-test
```

```
(seq-test 10 2)
;Value: 33
```

We now study the the changes we need to make to the BINDEX language implementation in order to add `bindpar` and `bindseq`.

3.1 Extending the `exp` Data Type

Our first step is to extend the `exp` data type to include summands for `bindpar` and `bindseq`:

```
and exp =
    ⋮
  | Bindpar of var list * exp list * exp (* parallel binding of names to defns in body *)
  | Bindseq of var list * exp list * exp (* sequential binding of names to defns in body *)
```

We could represent the bindings as a list of `var`/`exp` pairs, but we instead choose to unzip these into a `var` list and an `exp` list because the unzipped form is more convenient for most processing of these expresssions. We may always assume that these lists have the same length.

3.2 Parsing

In order to parse the new binding expressions, we must extend `sexpToExp`. We begin by defining some auxiliary functions for parsing bindings. The `parseBinding` function parses bindings whose s-expression form is (*I*$_{name}$ *E*$_{defn}$):

```
(* val parseBinding : Sexp.sexp -> (string, exp) *)
let rec parseBinding sexp =
  match sexp with
    Seq [Sym name; defn] -> (name, sexpToExp defn)
  | _ -> raise (SyntaxError ("parseBinding -- invalid binding form: "
                             ^ (sexpToString sexp)))
```

The `parseBindings` function parses lists of bindings whose s-expression form is ((*I*$_1$ *E*$_1$) ... (*I*$_n$ *E*$_n$)):

```
(* val parseBindings : Sexp.sexp -> (strings, exps) *)
and parseBindings sexp =
  match sexp with
    Seq bindingsx -> unzip (map parseBinding bindingsx)
  | _ -> raise (SyntaxError ("parseBindings -- invalid bindings list: "
                             ^ (sexpToString sexp)))
```

We will assume that the names bound in a `bindpar` or `bindseq` are distinct, but `parseBindings` does not verify this assumption. It is easy to extend it verify that that the returned list of strings does not contain any duplicates.

Now we are ready to extend `sexpToBinding`:

```
and sexpToExp sexp =
  match sexp with
     ⋮
  | Seq [Sym "bindpar"; bindingsx; bodyx] ->
      let (names,defns) = parseBindings bindingsx in
        Bindpar(names, defns, sexpToExp bodyx)
  | Seq [Sym "bindseq"; bindingsx; bodyx] ->
      let (names,defns) = parseBindings bindingsx in
        Bindseq(names, defns, sexpToExp bodyx)
```

A subtle point is that the above clauses must come *before* the clause for processing binary applications:

```
  | Seq [Sym p; rand1x; rand2x] ->
      BinApp(stringToBinop p, sexpToExp rand1x, sexpToExp rand2x)
```

Why? Because otherwise the pattern `Seq [Sym p; rand1x; rand2x]` will match both the `bindpar` and `bindseq` forms and an exception will be raised by `stringToBinop` indicating that these are not valid binops!

3.3 Unparsing

In order to unparse the new binding expressions, we must extend `expToSexp`. Unparsing is more straightforward than parsing:

```
and expToSexp e =
  match e with
    ⋮
  | Bindpar(ns,ds,b) -> Seq [Sym "bindpar";
                             Seq (map2 (fun n d -> Seq[Sym n; expToSexp d]) ns ds);
                             expToSexp b]
  | Bindseq(ns,ds,b) -> Seq [Sym "bindseq";
                             Seq (map2 (fun n d -> Seq[Sym n; expToSexp d]) ns ds);
                             expToSexp b]
```

3.4 Free Variables

In the expression `(bindpar ((`I_{name_1} E_{defn_1}`)...(`I_{name_n} E_{defn_n}`)) `E_{body}`)`, only the body expression E_{body} is in the scope of the names $I_{name_1} \ldots I_{name_n}$. So the free variables of this expression are the free variables of all the E_{defn} expressions unioned with the the difference of the E_{body} expression and all the I_{name} identifiers. This calculation is expressed as a clause in the `freeVarsExp` function as follows:

```
and freeVarsExp e =
  match e with
    ⋮
  | Bindpar(names,defns,body) ->
      S.union (freeVarsExps defns)
              (S.diff (freeVarsExp body)
                      (listToSet names))
```

If the extended `freeVarsExp` is in the module `BindexPlus`, we can test it as follows:

```
# open BindexPlus;;

# setToList
    (freeVarsExp
      (stringToExp
        "(bindpar ((a (+ d e))
                   (b (- a f))
                   (c (* b g)))
           (+ (* a b) (/ c d)))"));;
- : BindexPlus.S.elt list = ["a"; "b"; "d"; "e"; "f"; "g"]
```

For (`bindseq` ((I_{name_1} E_{defn_1}) ... (I_{name_n} E_{defn_n})) E_{body}), the calculation of free variables is more complex because the scope of each variable I_{name_i} includes the definition expressions $E_{defn_{i+1}}$ ...E_{defn_n} as well as E_{body}. So the bound variable of a binding needs to be subtracted off from the free variables of the definitions of subsequent bindings. This can be expressed succinctly in `freeVarsExp` clause using `foldr2` as follows:

```
  | Bindseq(names,defns,body) ->
      foldr2 (fun n d fvs ->
                S.union (freeVarsExp d)
                        (S.diff fvs (S.singleton n)))
             (freeVarsExp body)
             names
             defns
```

For example:

```
# setToList
    (freeVarsExp
      (stringToExp
        "(bindseq ((a (+ d e))
                   (b (- a f))
                   (c (* b g)))
           (+ (* a b) (/ c d)))"));;
- : BindexPlus.S.elt list = ["d"; "e"; "f"; "g"]
```

3.5 Environment Model Interpreter

In the environment model, we can extend the `eval` function to handle `bindpar` by using `map` to evaluate all definition expressions in parallel and then using `Env.bindAll` to extend the current environment with bindings that associate the resulting values to the corresponding variables names:

```
and eval exp env =
  match exp with
    ⋮
  | Bindpar(names,defns,body) ->
      eval body (Env.bindAll names (map ((flip eval) env) defns) env)
```

The sequential nature of `bindseq` makes it more challenging to implement its evaluation. We can use `foldl2` to start with the initial environment `env` and incrementally build a sequence of environments that reflect the contribution of each binding.

```
  | Bindseq(names,defns,body) ->
      eval body (foldl2 env
                          (fun e name defn -> Env.bind name (eval defn e) e)
                        names
                        defns)
```

The current definition expression `defn` is evaluated in the extended-environment-so-far `e`, and its value is bound to the current variable name `name` in `e` to make the next environment. The `bindseq` expression returns the result of evaluating the body expression `body` in the environment that results from processing all the bindings.

The form of the `eval` clause for `bindseq` encourages thinking about other meanings for binding constructs that can be obtained by small tweaks to the clause:

- What if `foldl2` is changed to `foldr2`?
- What if the `e` in `(eval defn e)` is changed to `env` (the parameter of `eval`)?
- What if the `e` in `Env.bind ... e` is changed to `env`?

There are eight possible permutations of these possibilities. Which correspond to the meaning of `bindpar`?

3.6 Substitution Model Interpreter

To extend the substitution model interpreter, we must first extend the `subst` function to handle `bindpar` and `bindseq`. Recall that to prevent variable capture, the substitution function gives fresh names to all bound variables before performing substitution on the subexpressions. This is straightforward in the case of `bindpar`, because all the renamings can be performed in parallel:

```
(* val subst : exp -> exp Env.env -> exp *)
let rec subst exp env =
  match exp with
    ⋮
  | Bindpar(names,defns,body) ->
      let names' = map StringUtils.fresh names in
        Bindpar(names',
                map ((flip subst) env) defns,
                subst (renameAll names names' body) env)
                (* or subst body (bindAll names names' env) *)
```

For example:

```
(* make senv, a sample substitution environment *)
# let senv = Env.make ["a"; "b"; "c"; "d"]
                      [BinApp(Add, Var "a", Var "b");
                       BinApp(Sub, Var "a", Var "b");
                       BinApp(Mul, Var "a", Var "b");
                       BinApp(Div, Var "a", Var "b")];;
val senv : BindexPlus.exp Env.env = <abstr>
```

```
# StringUtils.print
   (expToString
    (subst (stringToExp
             "(bindpar ((a (+ d e))
                        (b (- a f))
                        (c (* b g)))
                (+ (* a b) (/ c d)))")
           senv));;
(bindpar ((a.2 (+ (/ a b) e))
          (b.1 (- (+ a b) f))
          (c.0 (* (- a b) g)))
  (+ (* a.2 b.1) (/ c.0 (/ a b)))
 )- : unit = ()
```

Renaming the bound variables in `bindseq` is a fair bit trickier, because we must accurately model the fact that each variable name declared by `bindseq` is bound in the following definition expressions as well as the body expression. We can express this process via the following clause:

```
  | Bindseq(names,defns,body) ->
      let (names',defns',body') = substBindseq (zip (names,defns)) body env
       in Bindseq(names',defns',body')
```

where `substBindseq` is the following recursive function:

```
and substBindseq bindings body env =
  match bindings with
    [] -> ([], [], subst body env)
  | ((name,defn):: bindings') ->
      let name' = StringUtils.fresh name in
      let (names',defns',body') =
        substBindseq bindings' body (Env.bind name (Var name') env)
        in (name'::names', (subst defn env)::defns', body')
```

`substBindseq` processes each binding by extending the current environment with a renaming of the binding's name to a fresh name. It also uses the current environment to perform renaming in the definition of the current binding. This process could also be express via `foldl2` (try it, as an exercise), but the recursive version may be easier to understand.

Here is an example of substituting into a `bindseq` expression, using the sample substitution environment `senv` from above:

```
# StringUtils.print
   (expToString
    (subst (stringToExp
             "(bindseq ((a (+ d e))
                        (b (- a f))
                        (c (* b g)))
                (+ (* a b) (/ c d)))")
           senv));;
(bindseq ((a.3 (+ (/ a b) e))
          (b.4 (- a.3 f))
          (c.5 (* b.4 g)))
  (+ (* a.3 b.4) (/ c.5 (/ a b)))
 )- : unit = ()
```

Now we're ready to tackle evaluation in the substitution model. As to be expected, evaluating `bindpar` is easier than `bindseq`:

```
and eval exp =
  match exp with
    ⋮
  | Bindpar(names,defns,body) ->
      eval (substAll (map (fun defn -> Lit (eval defn)) defns)
                          names
                          body)
  | Bindseq(names,defns,body) -> evalBindseq (zip (names,defns)) body
```

where `evalBindseq` is the following auxiliary function:

```
and evalBindseq names defns body =
   match (names,defns) with
    ([],[]) -> eval body
  | (n::names',d::defns') ->
      let sub = subst1 (Lit (eval d)) n in
        evalBindseq names' (map sub defns') (sub body)
  | _ -> raise (EvalError("shouldn't happen"))
```

In the `bindpar` clause, `eval` is mapped over the definitions to evaluate them in parallel and `substAll` simultaneously substitutes these values for the variable names in the body. Evaluating `bindseq` is expressed in terms of the recursive `evalBindseq` function, which performs the substitution of a binding value for a binding name on all following definitions as well as the body.