


Extra problems, week 6

[Subscribe for email updates.](#)

 PINNED

 No tags yet. [+ Add Tag](#)

Sort replies by: [Oldest first](#) [Newest first](#) [Most popular](#)

Charilaos Skiadas COMMUNITY TA · [a month ago](#) 

As always, these problems go far and beyond what is required/expected from the class, and should only be pursued after you have finished all other requirements for the week. That being said, you might find the problems in the first section a good practice in understanding how MUPL works, and as good tests of your MUPL interpreter.

More MUPL macros/functions

We will write here more MUPL macros and functions. A "MUPL function" is basically a Racket definition of a `(fun ...)` MUPL expression. A "MUPL macro" is a Racket function that takes in MUPL expressions and produces a MUPL expression.

- We will start with "boolean" expressions. A boolean expression is going to be a MUPL expression that always evaluates to either `int 1` (mupl-true) or `int 0` (mupl-false). You might find it convenient to create those synonyms first.
 - Write a MUPL macro `mupl-if` that takes a boolean expression `e1` and two more MUPL expressions `e2`, `e3`. It would evaluate `e1`, and if it is equal to `mupl-true` then it evaluates `e2`, otherwise it evaluates `e3`.
 - Write a MUPL macro `mupl-not` that takes a boolean expression `e` and returns its "negation". So when `(mupl-not e)` is evaluated, it should produce 1 if `e` would have evaluated to `0` and to `0` otherwise).
 - Write a MUPL macro `mupl-and` that takes two boolean expressions, and returns a boolean expression that is the "AND" of the two expressions (and obeying the usual short-circuiting rule that if the first expression evaluates to false then the second will not be evaluated)
 - Similarly write a MUPL macro `mupl-or`
 - Write a MUPL function `mupl-any` that takes a MUPL list of boolean expressions and evaluates them in order until one of them returns `mupl-true`. It then results in `mupl-true`. If it reaches the end of the list, it results in `mupl-false`.
 - Similarly, write a MUPL function `mupl-all` that takes a MUPL list of boolean expressions and evaluates them in order until one of them returns `mupl-false`. It then results in `mupl-false`. If it reaches the end of the list, it results in `mupl-true`.
- Extend the MUPL grammar with a `isapair` primitive, and write a MUPL function `isalist` that takes a MUPL expression and returns `mupl-true` if the expression is a MUPL list, and `mupl-`

`false` if it isn't.

3. Write a MUPL function `mupl-curry` that takes a function that expects a pair and turns it into a curried function.
4. Write a MUPL function `mupl-uncurry` that takes a function of two curried arguments and turns it into a function of a pair.
5. Write a MUPL function `mupl-foldl` that takes in curried form a function "`a*b -> b`" an initial value and a MUPL list. It then basically performs a fold: If the list is empty, it would return the "initial value", otherwise it would form a MUPL pair out of the first element of the list and the initial value, MUPL call the provided function on the pair, and recursively call itself with the rest of the list and the appropriate new "initial value".
6. Write a MUPL function `mupl-filter` that takes in curried form a function that returns `mupl-true` or `mupl-false`, and a MUPL list, and it creates a new MUPL list out of those values from the original list for which the function evaluates to `mupl-true`. You can either do this "from scratch" or use `mupl-foldl`.
7. Write a MUPL function `mupl-append` that takes as argument a MUPL pair of MUPL lists, and appends the second list to the first. (Also do a curried version of the function)
8. Write a MUPL function `mupl-zip` that takes as argument a MUPL pair of MUPL lists, and returns a MUPL list formed from the pairs of the two lists, stopping when one of the lists is empty.
9. Write two functions `racket->mupl` and `mupl->racket` that convert between literal values in the two languages, i.e. they would take any racket construct built out of cons cells, integers, and null, and turn it into a mupl construct built out of a pair, int and a unit (and vice versa).
10. Write a macro `call-curried` so that `(call-curried f exps)` expands into the call of `f` on the first argument of `exps`, and calling the result on the second argument of `exps` and so on. Here `f` is a MUPL expression representing a function in curried form, and `exps` is a Racket list containing MUPL expressions of the arguments to be applied. In other words, what would have been the `(f x y)` call would be written as `(call-curried f (list x y))`.
11. Write a MUPL function `mupl-add` that is the function `f(x,y) = x+y`.

With some of the above under your belt, you can now write something like:

```
(eval-exp (call-curried mupl-foldl (list mupl-add (int 0) (racket->mupl (list 1 2 3 4 5)))))
```

to form and evaluate an expression that performs a fold of the add function on a mupl list of the numbers 1 through 5, with initial value of 0. Still a bit verbose, but try writing it without all these helpers ;).

Mupl Extensions

In the subsequent sections we will consider various extensions to MUPL:

1. First, we port our interpreter to SML
2. Next, we extend/change some of MUPL's mechanics, resulting in a new language, RSL. This new language contains `letrec` semantics, boolean values, and a richer set of operators as well as if-then-else clauses.
3. We then discuss and implement typechecking and type inference in RSL.

Let's get started, shall we?

MUPL in SML

This section will be a translation of the homework problem to SML. As this is closely related to the homework assignment, please refrain from discussing details of your solutions in the forum. Discussions of more general aspects of the problem are of course more than welcome.

We start with declaring the language signature:

```
signature MUPL =
sig
  exception EvalError of string
  structure E : ENV

  datatype muplExp =
    Int of int
  | Var of string
  | Add of muplExp * muplExp
  | Func of string option * string * muplExp
  | IfGreater of muplExp * muplExp * muplExp * muplExp
  | Call of muplExp * muplExp
  | MLet of string * muplExp * muplExp
  | APair of muplExp * muplExp
  | Fst of muplExp
  | Snd of muplExp
  | IsAUnit of muplExp
  | AUnit

  datatype muplValue =
    IntV of int
  | APairV of muplValue * muplValue
  | Closure of { env: muplValue E.env, f: muplExp }
  | AUnitV

  val show: muplValue -> string
  val typ: muplValue -> string
  val evalEnv: muplValue E.env -> muplExp -> muplValue
  val eval: muplExp -> muplValue
end
```

You will notice that we have separated those MUPL expressions that are "values". These can serve as the return type of our `eval` function. This has resulting in some duplications. For instance there is a `Int` type that is a `muplExp`, and an `IntV` which is a `muplVal`. Similarly for pairs and units. Closures did not need to appear in two places, as they are only values.

Note that for functions we have used an option type for the first string, which is meant to represent the

name of the function, to allow for the possibility of anonymous functions (instead of Racket's use of `#f`).

The structure `E` models an abstract "environment", and it should have a signature as follows:

```
signature ENV =
sig
  type 'a env
  val empty: 'a env
  val insert: 'a env -> string * 'a -> 'a env
  val lookup: 'a env -> string -> 'a option
end
```

So simply put, all you can do with environments is start with an empty environment, insert a new binding/pair into that environment to produce a new environment, or look up a string in the environment to find its binding.

You should write a structure `Env` that implements this signature. Then in your implementation of the Mupl evaluation you would set `structure E = Env`. A trivial list-of-pairs implementation would serve our needs just fine, where `insert` is a simple cons operation (`::`) and lookup does a simple linear search. As `lookup` returns an option, there is no need for any exceptions.

Returning to our MUPL signature, note that there are two basic methods, one that evaluates an expression within a specific environment on which to look things up, and another that simply evaluates an expression (starting with an empty environment). `eval` will just call `evalEnv`, so `evalEnv` is the main function you will need to implement, and it will be a large case expression.

Note that the environments contain `muplValue` elements, rather than `muplExp` elements, as only values go into the environment.

Finally, we've added a "show" method that "prints" `muplValue`s, and a `typ` method that returns a string representation of the value's "type", for better error messages, so you can say things like "expected int, but found ...".

Here is a start on the structure Mupl implementing this signature:

```
structure Mupl :> MUPL =
struct
  exception EvalError of string
  structure E = Env
  datatype muplExp =
    ...
  datatype muplValue =
    ...

  fun show v =
    case v of
      ...
```

```

fun typ v =
  case v of
    ...
fun evalEnv env exp =
  case exp of
    Int i      => IntV i
  | Var s      => ...
  | Add (e1, e2) =>
    (case (evalEnv env e1, evalEnv env e2) of
      (IntV i1, IntV i2) => IntV (i1 + i2)
    | (v1, v2) => raise EvalError ("Add expects integers. Instead found: "
      ^ typ (APairV (v1, v2))))

fun eval exp = evalEnv Env.empty exp
end

```

Most cases in `evalEnv` will require a case expression of their own, to check if the subexpressions in them evaluate in the kinds of values they should (e.g. you can only add integers). Just like you have to do for the assignment.

You might find it helpful, to cut down on the number of nested case expressions, to create some helper functions:

```

expectIntPair: mulpValue * mulpValue -> (int * int -> 'a) -> string -> 'a
expectPair: mulpValue -> (mulpValue * mulpValue -> 'a) -> string -> 'a
makeCall: mulpValue * mulpValue -> mulpValue

```

The first function takes two MUPL values, a function that acts on pairs of integers, and a string. If the two values are integers, it extracts the actual numbers and applies the function to them. Otherwise it produces a nice error message using the string. The second function does a similar operation for a value that is meant to be a `APairV`, and is useful for the treatment of the `Fst` and `Snd` cases. The last function takes in two MULP values, expects the first to be a closure, then applies that function/closure to the second value; and produces an error if the first value is not a closure. As this function needs to call `evalEnv`, and in turn will be called by `evalEnv`, you will need to place it in a mutually recursive function along with `evalEnv`, using the keyword `and`.

Before moving on, you should consider how MUPL macros / MUPL functions would look like in SML. Macros for example would be SML functions `mulpExp -> mulpExp`, possibly with more than one `mulpExp` in the argument spot. Mupl functions would be just `mulpExp` expressions, or perhaps functions that given some parameters generate such expressions.

You might like to even write some of the extra problems suggested earlier in SML.

Going further: The birth of RSL

You should now have a basic working interpreter for MUPL. We now proceed to define a somewhat related language that also admits booleans. Let's call it RSL (read "ReSiL", for "real simple language").

It will use the same basic Env structure we defined earlier. Here is its signature:

```
signature RSL =
sig
  exception EvalError of string
  structure E : ENV

  datatype binop = ADD | SUB | MULT | DIV | MOD
  datatype logop = EQ | LT | LE | NEQ
  datatype rslVal =
    IntV of int
  | BoolV of bool
  | PairV of rslVal * rslVal
  | UnitV
  | ClosV of { env: rslVal E.env, f: rslExp }
  | PromV of rslVal option ref (* Promise value, for use with letrec *)
  and rslExp =
    Int of int
  | Bool of bool
  | Var of string
  | Binop of binop * rslExp * rslExp
  | Logop of logop * rslExp * rslExp
  | If of rslExp * rslExp * rslExp
  | Func of string * rslExp
  | Call of rslExp * rslExp
  | Letrec of (string * rslExp) list * rslExp
  | Pair of rslExp * rslExp
  | Fst of rslExp
  | Snd of rslExp
  | Unit

  val show: rslVal -> string
  val typ: rslVal -> string
  val evalEnv: rslVal E.env -> rslExp -> rslVal
  val eval: rslExp -> rslVal
end
```

Let us walk through the differences. First of all, we have an additional type of values, namely booleans. The 5 types of values are ints, bools, pairs of other values, unit, and closures. There is one more type of value, a "promise", which is to be used in connection with letrec, and we will discuss it more when we get to letrec.

In the expression side, there are some changes. First of all, there is a **Binop** expression for the binary operators that expect integers and produce integers. Dividing or Modulo-ing by 0 should result in a runtime error. There is also a **Logop** expression for comparison operators, that expect 2 integers as input and produce a boolean as output. Finally there is a **If** expression for **if-then-else**, which

works as expected (and requires a boolean for the test).

Functions no longer have a name of their own, they only have a spot for the argument name. They will be given names via the `letrec` construction in a moment. Calls work the same.

Finally, `Letrec`. This works like Racket's `letrec`: It takes a list of string-expression pairs, representing bindings, as well as a "body" expression. It then creates an environment in which all of the bindings exist, but may not yet be set to a value. It then proceeds to evaluate each binding in order, in this new extended environment. So this allows for mutual recursion of functions, for example, as they can have a reference to each other in their closure environment, and since they are not going to be executed right away as the bindings occur, the fact that one will precede the other in the code is not a problem.

In order to accomplish this, the system needs to be modified at some level, as it requires the environment to be in place before the bindings are computed, but the correct values for those bindings won't be known until they are computed, which effectively means you need to modify the environment after it has been created. We take a different approach here. The environment will not change, but instead one of the possible values you can assign is a "promise", which is set to be a reference to a value option, so this is a value of type `rslValue option ref`. In order to create the environment, you create bindings to `PromV (ref NONE)` values. Later as you evaluate these bindings, you adjust the values stored in those references.

A new value means that you need to do a bit more work in the branch of your code that deals with evaluating a `Var s` expression. The value returned by this expression may be one of these new promise values, and therefore it needs to go through a further get process to take the actual value out of the reference, or error if the value is not yet set. The other branches of your code do not have to worry about it, as the environment is the only place where these promise values will ever be encountered.

You should have all you need to implement the `Rsl` structure now. Good luck! And if you are feeling adventurous, do the necessary modifications in order to implement this in Racket.

Typechecking and inference

We will now work towards a type-checking/inference system for `rsi`. Let us recall the idea of type checking and type inference.

Before the program is executed, the typechecker processes the program expression, without evaluating it, and assigns types to all expressions along the way. The code to do that would be similar to the large case expression that `evalExp` does, except that it performs a different role. It needs to determine the types of subexpressions, see if those are compatible with the semantics for the expression (are we trying to do an `If` on a pair? Do the two branches have matching types, so that we can assign a type to the whole expression?). The result of this process is either a `TypeCheck` error (we just use the `EvalError` exception for it) or a type for the expression. And instead of using a `rslValue` environment to look values for symbols up, we use a "rslType" environment to look up the computed types for symbols.

But as our functions do not have specified types for arguments and return values, we need a type inference step that would infer those types. The way this process works, in broad terms, is that the typechecker assigns a "variable type" on pretty much every expression along the way, and then it

creates a sets of constraints that these variable types must satisfy. In our setting, these constraints have the form "this type must equal that type". For instance, in a `if-then-else` expression, the variable type of the test part must equal the type `BoolT`, and the two variable types from the `then` and `else` parts must equal each other and also equal the return type of the whole expression. We will discuss this in more detail a bit later, but for now the main point is that the result of the first pass of the typechecker is a "variable type" for the expression, along with a list of constraints that need to be satisfied among various types defined along the way.

The next, and final, step, is a *unification* process. This process goes through each constraint, and determines what that constraint means for the relevant types. Again we will go into this process more later, but essentially this might result in some unresolvable relations, for instance a type that has to be both `IntT` and `BoolT`, in which case a type error is signaled, or it might successfully process all constraints. At this point many variable types have been "instantiated" with concrete types. Any types left have no constraints to them, and can turn into parametric types (`'a`). In particular, this process will have assigned some concrete or parametric type to the variable type of the expression we were trying to evaluate.

That is the big-picture description. Now let's go through it step by step.

First we're going to need a datatype for our RSL types:

```
datatype rslType = IntT
                  | BoolT
                  | UnitT
                  | PairT of rslType * rslType
                  | FuncT of rslType * rslType
                  | ParamT of string
                  | VarT of int * rslType option ref
```

So a value may have type `IntT`, or `BoolT`, or it may have as type a `PairT` type, or a `UnitT` type, or it could be a function type with types for the source and destination, or it could be a parametric type, with some string name to distinguish them.

Finally, `VarT` types are meant to be variable types used during the inference process (like the `X` types you sometimes see in SML error messages). Ideally a successful typechecking of an expression will result in a type for the expression consisting entirely of the first 6 type forms above. We will discuss the form of `VarT` a bit further down, but for now let us say that the option ref part would be instantiated with `ref NONE`. The integers are there mostly for printing purposes, i.e. so you can see `X1` and differentiate it from `X2`, but could be omitted. Types can be separated by their refs, because two refs are equal exactly if they are literally the same ref, and their contents are never inspected (Trying `ref NONE = ref NONE;` in the REPL and getting false back is always fun).

Before moving on, it would be nice to create some helper functions:

```
newParamType: unit -> rslType
newVarType   : unit -> rslType
typeToString: rslType -> string
```


The first two create new types, maintaining an internal counter to always generate a new type. For the `VarT` this is easy, as you can simply have a counter than you keep increasing (or if you have omitted the integer part, then you just need to create a `ref NONE` every time, which you would need to do anyway). For the parameter types, you will want to devise some system that starts by assigning "a" to the first request, then "b" and so on, then moves on to double letters and so on. An internal counter along with a function to convert a number to a string would do. The main requirement is that no two calls to one of these methods should ever produce the "same" answer.

Finally, the third method just creates a nice printout of the type, and you can arrange this however you want.

A proper typechecking process will probably try to annotate each expression with its type, for which it will probably require a new datatype for "typed expressions", essentially duplicating the existing datatype. We will focus on a more moderate goal: Assembling the series of constraints imposed on the type, then running an inference trying to determine what each actual type the various type variables created along the way should be, as well as the type of the resulting expression. So our main function will be something like:

```
getConstraints: (string * rslType) E.env -> expr -> rslType * (rslType * rslType) list
```

So given an expression and an environment indicating what variables are bound to what types, we return a pair of the type of the expression (most likely a variable type at this point) along with a list of "constraint" pairs, where a constraint is simply a pair of types that are required to be the same. For example, here is what might happen in the "Add" portion of `getConstraints`, which is as usual a large case expression on the various `expr` types:

```
1 Add (e1, e2) =>
  let val t = newVarType()
    val (t1, cons1) = getConstraints env e1
    val (t2, cons2) = getConstraints env e2
    val allCons = (t, t1) :: (t1, t2) :: (t2, IntT) :: (cons1 @ cons2)
  in (t, allCons)
end
```

There are some variations we could have taken, but essentially:

1. We create a new type to represent our result
2. We compute the types and constraints from the two subexpressions
3. We require that the two types from the subexpressions be equal to each other, be integers, and be equal to our return type. In addition to all the constraints from the subexpressions.

You could for example have simply returned `t1` instead of `t`, and not create `t` at all.

A similar pattern will take place for each branch, some branches being easier than others. One branch that is different than the evaluation step is the function definitions. For a function definition you need to create one new variable type for the argument, and proceed to typecheck the body of the function in an appropriate environment. You then create a function type from the type of the body and the type of the argument, and return that as the function expression's type (or you can create a new variable type

and set a constraint for it to equal this function type). But the important thing is that the "getConstraints" process will go through the body right away, as opposed to what happens in the evaluation step, where the function is simply stored as a closure, and the body is processed only during a call.

The typechecking of a call on the other hand does not need to look at the function's body. But it does need to create two new types, say `X1` and `X2`. It then would add the constraint, that the operator's type equals `FuncT (X1, X2)`, and that the operand's type equals `X1`. And it will return `X2` as the result type.

So this would complete the first part of the typechecking process, creating a set of "type variables" and a list of constraints those variables have to satisfy.

At this point you should probably go ahead and implement the `getConstraints` method.

The next step in the process would be to "unify" those constraints.

The unification algorithm essentially goes through each constraint in the list, and runs a `unify` function on the two types in the constraint. If at least one of the types involved is a variable type, then the process will end up updating the option ref part of the type with the "instantiation", and we will discuss this process in a little more detail. The result of a call to `unify` will be either a unit, indicating the types were "unified", i.e. compatible, and perhaps some variable type fields were updated to reflect the new relation, or a raised exception indicating that there were incompatible types (for example we were unifying an `intT` with a function type). Any types that remain uninstantiated at the end of the process can be "instantiated" as parametric types.

The process has a little bit of a union-find feel to it. For example, if we encounter two variable types that haven't been instantiated yet (i.e. they have `NONE` as their instantiation field), and we are asked to unify them, then we will need to set one of the type's field to be `SOME t` where `t` is the other type. What this means in future relations is that you may have to follow the instantiation path for multiple terms to find the "real" type, before working on it.

In general, every variable type will be either in the form `VarT (ref NONE)` or in the form `VarT (ref (SOME t))`, in which case we say that `VarT` has been instantiated with type `t`, and basically should be treated as of type `t` for most interactions from this point on.

We will now describe the `unify` function in some detail. The implementation will be left to you. Throughout `t1, t2` are the two types passed to `unify(t1, t2)`. At this point in the process, we do not have any parametric types. We will talk about "concrete" types if they are not `VarT` or `ParamT` types.

- Two bool types, two int types, two unit types, are all ok and there is nothing more to do.
- Two pairs will trigger unify calls on the matching components (i.e. unify on the two first components, and unify on the two second components), and if those are ok then we're ok.
- Two function types will do the same.
- If one of the types is a variable type that is already instantiated to `SOME t`, then continue (recurse) the unify call with this `t` instead of the variable type.
- If both types are the same variable type, then we're ok, nothing to do.
- If one type is an uninstantiated variable type, then instantiate the variable type with the other type. There is one exception to this rule. If the variable type appears somewhere in the concrete type,

then you should raise an expression (essentially we shouldn't be allowing an equation that says something like `X = (X, _)`, a type `X` cannot equal a pair type where one of the components is `X` again). So you need to implement some sort of `containsType t1 t2` method that tells you whether `t2` has somewhere in it a `t1`.

- All other possibilities would be a typecheck error.

And this is it! If this process successfully goes through all the constraints, then our function has typechecked, and we can follow the instantiation of the expression's type to determine the expression's actual type. If along the way we encounter any uninstantiated variable types, these can now be instantiated to parametric type, as there are no constraints left on them.

And you have just successfully done type checking and type inference!

For reference, here is the type-relevant part of the `RSL signature` that you may want to implement in the `Rsl` structure:

```
datatype rslType =
  IntT
| BoolT
| PairT of rslType * rslType
| UnitT
| FuncT of rslType * rslType
| ParamT of string
| VarT of int * rslType option ref

val newParamType: unit -> rslType
val newVarType  : unit -> rslType
val typeToString: rslType -> string

val containsType: rslType -> rslType -> bool
val unify: rslType * rslType -> unit
val resolve: rslType -> rslType

val getConstraints: rslType E.env -> rslExp -> rslType * (rslType * rslType) list
val getAllConstraints: rslType E.env -> rslExp list -> rslType list * (rslType * rslType) list
val typecheck: rslExp -> rslType
```

Not all of these need to be exposed, but you get to see their signatures. `containsType t1 t2` returns true if the type `t1` appears somewhere inside `t2`. This is one of the cases, if `t1` is a variable type, where the unification process must declare failure.

`typecheck` will essentially start by calling `getConstraints` on the expression and an empty environment, getting back a type for the expression and a list of constraints. It will then apply `unify` to each pair in the constraints list, then try to "resolve" a type before returning it. We will talk a bit more about resolve in a minute, but for now here is the general form of the typecheck, in semi-pseudocode:

```

fun typecheck exp =
  let val (t, cons) = "call getConstraints"
  in ("apply unify to all the cons"; resolve t)
  end handle EvalError s => "print nice message"

```

Now for `resolve`:

- bools, ints, units, parametric types resolve to themselves
- pairs and functions resolve to corresponding pairs/functions of the resolved components
- a `VarT (ref SOME t)` resolves to the result of resolving `t`
- a `VarT (ref NONE)` is essentially a variable type with no constraints, so we need to set it to a new parametric type and resolve to that type.

And that's it! You have now performed a full typecheck of the system, plus type inference!

Having gotten a typechecking process in place, you can now revisit the eval process, and require it to do a typechecking before the evaluation. This can remove some unnecessary tests, for instance in the binary operator setting you can expect your values to be integers, or things would not have typechecked. You therefore do not need to do a case expression any more, and can instead rely on some functions you may write, like `getInt`, which takes the integer out of a `IntV` value, and raises an error otherwise (and in practice the error raising part should not be happening).

And beyond

Well, this completes the "official" part of the extra problems. Here are some directions to go in. Pick whichever one sounds more interesting.

1. Extend the equality and non-equality operators to work on more than just integers.
2. Extend the `LE` and `LT` operators to work on any two things of the same type formed out of pairs and integers (via lexicographic ordering).
3. Add new constructs in the language to allow for "immutable lists of a single type". So you would need something like `Lst of rslExp list`, `LstV of rslValue list`, `LstT of rslType` (notice, no list needed here, must be a single type), `Head of rslExp`, `Tail of rslExp`, `IsNull`.
4. Allow for tuples/records (eval is not too bad, but typechecking might prove interesting). For records you need to add two new parts to the expression datatype, a `Rec of (str * exp) list` to form records and a `Field of str * exp` to extract a field from a record. For values you just need a `RecV of (str * value)`. For typechecking, you will need some way to represent the idea that "this record needs to have these 3 fields of these types, but can also have any number of fields of other types".
5. Allow for more complicated forms on the left-hand-side of a letrec binding (essentially a start on pattern-matching). So one should be able to do `"letrec (a,b) = pr"` and then `pr` would need to be an expression that evaluates to a pair and `a,b` would be set to its components.
6. Add a new datatype `rslTypedExp` that has type annotations associated with each expression. These annotations may be optional (so represented as `rslType option`), and there should be a way to turn a `rslExp` into a `rslTypedExp`.
7. Right now we have implemented functions as having one argument. Amend the definition to instead expect a possibly larger, but fixed, number of arguments.
8. Add support for `ref`s. This is a considerable adjustment, depending on how you implement it. It

will definitely complicate the type-checking. To implement refs, the simplest way would be to use SML's built-in support for refs. So you would need a new value constructor, `RefV of rs1Value ref`, and a new type constructor `RefT of rs1Type`. Then you need 3 new expressions: `Ref of rs1Exp`, `Assign of rs1Exp * rs1Exp` and `Bang of rs1Exp` (reminiscent of `!`). `Ref e` should evaluate `e`, then create a `RefV` with that resulting value wrapped around a ref. `Assign (e1, e2)` would evaluate `e1`, then evaluate `e2`. `e1` should be a `RefV` in which case the type inside the `RefT` should match the type of `e2`, and we need to store `e2`'s value into the `RefV`'s reference. Finally, `Bang e` evaluates `e`, and it should evaluate to a `RefV`, in which case it extract the value stored in the ref. The bang expression will typecheck only if `e` typechecks to a `RefT t`, and then `t` is the return type of the expression. You also need to take great care to from now on evaluate the parts of the expressions in the correct order, as each expression can now have side-effects. Lastly, in order to be able to really benefit from refs, you will need some way to chain expressions together, so you would need some sort of "Sequence" expression, that evaluates to the last expression in a list, or something like that (or perhaps just two expressions, evaluate the first but ignore its return value, then evaluate to the second).

9. (This is probably mutually exclusive with 8) Implement "lazy evaluation" semantics in RSL. You can either try to add lazy evaluation everywhere, or just on the arguments of function calls. This question is deliberately left rather vague.

↑ 6 ↓ · flag

Charilaos Skiadas · COMMUNITY TA · a month ago 🔒

Updated with a lot of MUPL goodness.

↑ 0 ↓ · flag

[+ Comment](#)

Anonymous · a month ago 🔒

So I was porting the interpreter to SML and was implementing syntactic sugar. I think SML is really nice for creating internal DSLs because it allows you to define your own infix ops (it's a shame the fixities can't be exported). Now I have this:

```
local
  infix 2 <-
  infix 2 <--
  infix 2 <:
  infix 9 ++
  infixr 3 :=>
  infix 4 \>
  infix 5 </
  infixr 6 <\
  infixr 1 $
```

```

infixr 8 otherwise
open Mupl
val op $ = Util.$
in
val e1 = eval $ Var "foo" <- Int 1 $ Add (Int 100, Var "foo")
val xs = fromList [Int 1, Int 2, Int 3]
val e2 = eval $ Fst o Snd o Snd $ xs
val e3 = eval $ Var "a" <- Int 200
              $ Var "b" <- Var "a" ++ Var "a" ++ Var "a" ++ Var "a" ++ Var "a"
              $ Var "c" <- Int 30
              $ Var "a" ++ Var "b" ++ Var "c" ++ Int 4
val dec = "%n" :=> "%n" ++ Int ~1
val e4 = eval $ dec </ Int 100
val e5 = eval $ "%inc" <- "%n" :=> "%n" ++ Int 1
              $ dec </ "%inc" <\ Int 0
val e6 = (toList o eval)
          $ "%map" <- map
          $ "%+n" <- "%n" :=> "%map" </ ("%x" :=> "%x" ++ "%n")
          $ "%+n" </ Int 100 </ xs

val mul = "%*" <: "%n" :=> "%m" :=>
          (when ("%m") $ ("%n" ++ ("%*" </ "%n" </ dec <\ "%m"))
           otherwise (Int 0))

val e7 = eval $ mul </ Int 64 </ Int 16
val e8 = (toList o eval) $ map </ mul <\ e7 </ xs
end

```

Started off promising I think, but then kinda turned unreadable towards the end. I think it's because of my bad choice of symbols. I assume this sort of thing is done often in ML (it stands for meta-language after all). Is there a standard set of symbols used for things like this?

↑ 0 ↓ · flag

[Charilaos Skiadas](#) COMMUNITY TA · [a month ago](#) 🔒

I'd say the moment you have to put "Int" in front of your integers and "Var" in front of each variable access, the game's over. I think using it as a DSL would have more to do with using its usual evaluation, but creating functions and values in such a way as to make expressing yourself easy.

I don't really know of any standards for the symbols. And there's such a thing as overdoing it with them ;).

Very interesting effort though!

↑ 0 ↓

· flag

+ Comment

Anonymous · a month ago

2. Extend the MUPL grammar with a `isapair` primitive...

do you mean extend using (struct isapair (e)) ?

↑ 0 ↓ · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Correct.

↑ 1 ↓ · flag

+ Comment

Anonymous · a month ago

Thank you! Unfortunately I won't have time to do the extra homework. Is it possible for you to publish this as markdown format so I'll be able to save it and work on it after the course is over?

↑ 0 ↓ · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Done: <https://github.com/skiadas/programming-language-extras>

Might brush it up later.

↑ 2 ↓ · flag

Anonymous · a month ago

Thanks!!

↑ 0 ↓ · flag

+ Comment

New post

To ensure a positive and productive discussion, please read our [forum posting policies](#) before posting.

B*I*

Link

<code>



Pic

Math

Edit: Rich ▾

Preview

☐ Make this post anonymous to other students☒ Subscribe to this thread at the same time

Add post