PPL192

Assignment 4

Responsible Lecturer: Michael Elhadad

Responsible TA: Michal Sharoni

General Instructions:

Provide answers to Part 1 in a file named part1.pdf.

Provide answers to Part 2 in a file named part2.l5.

Provide answers to Part 3 by updating all the files in the folder "types" of the files provided in the assignment4_template.zip

Submit all the files of your solution in a file id1_id2.zip which has the same structure as assignment4_template.zip

Part 1: Theoretical Questions

1. Recall that the code for evaluating a **set!** expression in the L4-box interpreter is:

```
const evalSet = (exp: SetExp, env: Env): Value | Error => {
    const v = exp.var.var;
    const val = L4applicativeEval(exp.val, env);
    if (isError(val))
        return val;
    else {
        const bdg = applyEnvBdg(env, v);
        if (isError(bdg)) {
            return Error(`Var not found ${v}`)
          } else {
              setFBinding(bdg, val);
              return undefined;
          }
      }
  };
```

Can we implement set! evaluation as a nonspecial form? Explain why.

2. Recall the usage of box in the interface we saw in class:

```
// Box-Env is defined inductively by the following cases:
// * <box-env> ::= <global-env> | <extended-box-env>
// * <global-env> ::= (global-env frame) // global-
env(frame:Box(Frame))
// * <extended-box-env> ::= (extended-box-env frame enclosing-env)
//      // extended-box-env(frame: Frame, enclosing-env: Box-env)
//
// Frame:
// * <fbinding> ::= (var val)
//   binding(var:string, val:Box(Value))
// * <frame> ::= (frame (var val)*)
//   frame(bindings:List(fbinding))
// applyFrame(frame, var) => val
```

If we remove the usage of **Box** in the current implementation of the **fbinding** datatype, would we still be able to implement recursive functions using **define** and **letrec**? Explain why.

3. For each one of the typing statements below, note whether they are **true or false** and explain why**:**

```
e. {f:[T1->T2], h:Number} |- (f g): T2
f. {f:[T1->T2], h:Number} |- (f h): T2
g. {f:[T1->T2], x:T1} |- (f x x): T2
```

4. Perform the typing inference algorithm for the expression:

```
((lambda (x1) (if (> x1 7) #t #f)) 8)
```

Show all the relevant steps.

5. a. Write the equivalent asynchronous version of the following synchronous code. Provide

```
function f (x : number): number {
    return 1/x
}
function g (x : number): number {
    return x*x
}
function h (x : number): number {
    return f(g(x))
}
```

the full type annotations for the functions.

b. Write the equivalent asynchronous version of the given synchronous code using error handling (callbacks with two parameters (err, res) like is used for Node functions such as **fs.readFile**).

6. Given the following function:

```
(define even?
 (lambda (n)
  (if (= n 0) #t (odd? (- n 1)))))

(define odd?
 (lambda (n)
  (if (= n 0) #f (even? (- n 1)))))

(define f
 (lambda (x)
  (if (even? x) x (- x 1))))
```

Perform the CPS-transformation on the function.

Part 2: Programming: Lazy Lists in L5

1. Consider the following Implementation of the unlabeled-tree interface in L5. In such trees, interior nodes have no label, leaves have labels.

```
(define make-tree list)

(define add-subtree cons)

(define make-leaf (lambda (d) d))

(define empty-tree empty)

(define first-subtree car)

(define rest-subtrees cdr)

(define leaf-data (lambda (x) x))

(define composite-tree? pair?)

(define leaf? (lambda (t) (not (list? t))))

(define empty-tree? empty?)
```

a. **Add the type definition** of all the functions in this interface according to the L5 type. These type definitions can be polymorphic (contain TVar) where appropriate.

b. Write a function **tree->leaves** which receives an unlabeled-tree parameter and returns an ordered list of the labels which appear in the leaves of the tree. Provide the full contract for the function.

c. Write a function **tree->lz-leaves** which receives an unlabeled-tree parameter and returns a lazy-list of the labels which appear in the leaves of the tree.

Write a function **same-leaves?** which receives two unlabeled-tree parameters and returns #t if they have the same leaves, otherwise a pair containing the first different labels in the lists of leaves. The function should use **tree->lz-leaves** and stop as soon as a node in the lists of leaves is found different.

For example:

```
(same-leaves?     '((a b) c)
                  '(a (b c)))  ➔ #t
(same-leaves?     '(a (b c))
                  '(a d c)) ➔ '(b . d)
```

Write the code in L5 – which is not compatible with Racket because of the type annotations.

In order to test your code, write a TypeScript file to execute your program (we will use our own tester to execute your code). You do not need to submit this file – and you should develop it inside a folder where the L5 interpreter code is present.

Part 3: Programming: Type Checking with Type Unions

In the L5 type system we have studied in class and its implementation, the only compound type expression that is supported is for Procedures (ProcTExp). In this question, we will extend the Type Checker of L5 so that it also supports Type Unions.

In all parts of this question, we do NOT consider type expressions of type TVar.

3.1 Extend the Syntax of TExp to support type union

The concrete syntax for type union expression is:
(TExp | TExp | ...)

Implement at least the following steps:

1. Add **UnionTExp** as part of the **TExp** type language (modify the file **TExp.ts**):
2. Modify the **TExp** type itself
3. Modify the parser: modify the function **parseCompoundTExp**
4. Modify **unparseTExp** to support **UnionTExp**

Guideline:

In the AST representation of nodes of type UnionTExp, make sure you **normalize** the list of type components, so that equivalent type expressions which are only syntactic variants of each other are represented internally in the same way. Syntactic variants of Type Unions are possible because the Union operation is **commutative** and **associative**.

For example:

The following two type expressions should have the same AST:

1. **(number | boolean)**
2. **(boolean | number)**

The following three type expressions should have the same AST:

1. **(number | boolean | string)**
2. **(number | (boolean | string))**
3. **((number | boolean) | string)**

To neutralize such syntactic variants, make sure all type unions in the Type Expression internal representation are flattened recursively and sorted in a stable manner. The sorting order should be using the string obtained when unparsing the flattened list of components in the union (that is, any type component which is not a union in the flattened union). For example, the two type expressions (number | boolean) and (boolean | number) will be parsed internally into the same UnionTExp with a list of TExp values corresponding [<TExp for boolean>, <TExp for number>] because the unparsed expressions sort as "boolean" and "number".

Make sure to define the TypeScript type of UnionTExp with a type declaration that is as precise as possible (does not allow values which would not be legal for a canonical representation of a union).

Use code similar to the following ramda code to properly sort TExp values inside the UnionTExp constructor (naturally this is not exactly the right sort function, you must adapt it to your needs):

```
import { sort } from 'ramda';
import { unparseTExp, makeNumTExp, makeBoolTExp, TExp } from './TExp';
console.log(sort((a: string, b: string): number => a < b ? -1 : a === b ? 0 : +1,
["c", "a"]));
console.log(sort((a: TExp, b: TExp) => unparseTExp(a) < unparseTExp(b) ? -1 : 1,
[makeNumTExp(), makeBoolTExp()]));
// Outputs: [ 'a', 'c' ] [ { tag: 'BoolTExp' }, { tag: 'NumTExp' } ]
```

3.2 Extend Type Equivalence check to support type union

In the type checker code, the function **checkEqualType(te1, te2)** determines whether two types are compatible. In the L5 type system we studied in class, this function verifies that te1 and te2 are not Error and are exactly identical.

With the introduction of Type Unions, the verification of type compatibility becomes more complex. For example, consider this program in L5 with type union:

```
(define f (lambda ((x : (number | boolean))) x)
(define (n : number) 2)
(f n)
```

In this case, the call to **(f n)** is type checked correctly even though the type **number** is not exactly equivalent to **(number | boolean)**.

As defined in the description of Type Compatibility in the TypeScript type system presentation we reviewed in Chapter 1 https://www.cs.bgu.ac.il/~ppl192/wiki.files/class/notebook/1.3TypescriptTypeChecking.html#Type-Compatibility-is-NOT-Symmetric:-Subtyping the relation between te1 and te2 in this check is actually not symmetric.

For example:

```
(define g (lambda ((x :number)) x)
(define (n : (number | boolean)) 2)
(g n)    ← this is not well typed
```

We say that T1 and T2 are compatible if T1 can be used in the places where T2 is expected.

In other words, T1 is compatible with T2 when T1 is a subtype of T2.

The type compatibility rules introduced by Type Union are the following:

1. AtomicType1 and AtomicType2 are compatible only if they are identical.
2. AtomicType1 and UnionType2 are compatible if AtomicType1 appears in the list of UnionType2.
3. AtomicType1 and ProcExpT2 are not compatible.
4. UnionType1 and AtomicType2 are not compatible.
5. UnionType1 and UnionType2 are compatible if for each type T1i in the list of components in Type1 there exists a type T2j in the list of components in Type2 such that T1i is compatible with T2j.
6. UnionType1 and ProcTExp2 are not compatible.
7. ProcTExp1 and AtomicType2 are not compatible.
8. ProcTExp1 and UnionType2 are not compatible.
9. ProcTExp1 and ProcTExp2 are compatible when:
   Consider that ProcTExp1 = (T11 * … * T1n -> T1r) and
   ProcTExp2 = (T21 * … * T2n -> T2r)
   a. Both types have the same number of parameters n (which can be 0)
   b. For i in (1…n): T2i is compatible with T1i
   c. T1r is compatible with T2r

NOTE: Observe in the comparison of procedural types (ProcTExp1 and ProcTExp2) that to conclude that ProcTExp1 <= ProcTExp2 we require that T2i <= T1i – the order is REVERSED.
In contrast, the order is NOT reversed for the return types (T1r <= T2r).

This property is called **contravariance** of argument types and **covariance** of return types for procedure types. It is explained nicely in this article if you want to understand why this surprising rule holds: https://medium.com/@michalskoczylas/covariance-contravariance-and-a-little-bit-of-typescript-2e61f41f6f68

To experiment with the semantics of this type system, you can write expressions in TypeScript. The property of argument contravariance is enforced in the TypeScript type system ONLY when the compiler uses the (non-default) option **–strictFunctionTypes**. We achieve this behavior in TypeScript by updating the file **tsconfig.json** and verifying that this option is set under the value "compilerOptions":

```
{
  "compilerOptions": {
    /* Basic Options */
    ...
    "strictFunctionTypes": true,              /* Enable strict checking of function types. */
    ...
  }
}
```

Implement the following steps:

1. Define the function checkCompatibleTypes(te1, te2) which compares two Type Expressions and returns true if te1 is a subtype of te2, false otherwise.
2. Provide at least 9 positive tests and 9 negative tests of type expression pairs to validate your function according to the rules specified above in the file **checkCompatibleTypes-test.ts**
3. Change the code of the L5 type checker to use **checkCompatibleTypes** instead of **checkEqualType**.
4. Provide at least 4 examples of fully type-annotated L5 expressions that pass type checking including procedures with at least one parameter and unions in the file u**nionTypeChecker-test.ts**.

3.3 Extend the IfExp Typing Rule in L5

The typing rule we presented in class for If Expressions in L5 introduces 2 constraints:
(see https://www.cs.bgu.ac.il/~ppl192/wiki.files/class/notebook/3.1TypeChecking.html#Type-Checking-Compound-Expressions)

The If-exp typing rule is thus specified as follows:

```
For every type environment _Tenv,
        expressions _test, _then, _else
        type expression _S:
If   _Tenv |- _test : Boolean
     _Tenv |- _then : _S
     _Tenv |- _else : _S
Then _Tenv |- (if _test _then _else) : _S
```

With union types, we can now support If Expressions with different return types for the **then** and for the **else** branch of If Expressions. For example, the type of:

```
(if (> x 0) x #t)
```

Will be: **(number | boolean)** as opposed to an error.

1. Write the new typing rule for If Expression. Make sure the type of the IfExp is a union only when it is necessary.
2. Modify the function typeofIf according to the new typing rule.
3. Provide at least 4 examples involving IfExp to validate this function in **unionTypeChecker-test.ts**.

בהצלחה!