PPL 242

Assignment 3

Responsible Lecturer: Yuval Pinter
Responsible TA: Tal Gonen

Submission Date: July 8, 2024

General Instructions

Submit your answers to the theoretical questions in a pdf file called id1_id2.pdf and your code (the ex3 folder), and ZIP those files together into a file called id1_id2.zip.

Do not send assignment related questions by e-mail, use the forum instead. For any administrative issues (milu'im/extensions/etc) please open a request ticket in the Student Requests system.

You are provided with the templates ex3.zip. Unpack the template files inside a folder. From the command line in that folder, invoke npm install, and work on the files in that directory, preferably working in the Visual Studio Code IDE (refer to the Useful Links). In order to run the tests, run npm test from the command line.

Important: Do not add any extra libraries and do not change the provided package.json and tsconfig.json configuration files. **The graders will use the exact provided files**. If you find any missing necessary libraries, please let us know.

Question 1: Theoretical Questions [12 points]

Are the following typing statements true or false? Explain why. **[4x3 points]**

a. {f : [T2 → T3], g : [T1 → T2], a : Number} ⊢ (f (g a)) : T3

b. {f : [T1 → [T2 → Boolean]], x : T1, y : T2} ⊢ (f x y) : Boolean

c. {f : [T1 × T2 → T3], y : T2} ⊢ (lambda (x) (f x y)): [T1 → T3]

d. {f : [T2 → T1], x : T1, y : T3} ⊢ (f x) : T1

Question 2: L52 - Extending Types to Include Set Operations and Type Predicates [18 points]

In this part and the next, we will create a new language, L52, which is based on the L5 you know.
First, we introduce L51 which includes the complex type Union in the following manner:

```
<TExp> ::= <atomicTExp> | <compoundTExp> | <TVar>
<compoundTExp> ::= <procTExp>
                 | (union <TExp> <TExp>)
```

Secondly, we introduce L52 that extends L51 with a type annotation scheme in the following manner:

```
<TExp> ::= <atomicTExp> | <compoundTExp> | <TVar>
<atomicTExp> ::= boolean | number | string | void
                 | any | never
<compoundTExp> ::= <procTExp>
                 | <typePredTExp>
                 | (union <TExp> <TExp>)
                 | (inter <TExp> <TExp>)
                 | (diff <TExp> <TExp>)
<procTExp> ::= ( <TExp>+ -> <TExp> ) | ( Empty -> <TExp> )
<typePredTExp> ::= ( <TExp> -> is? <TExp> )
```

The new forms denote the following:

1. **`any`**: a type which describes the set of all possible values (כל הערכים)
2. **`never`**: a type which describes the empty set (קבוצה ריקה)
3. **`(union <t1> <t2>)`**: a type which describes the set containing the union of the values in `<t1>` and `<t2>` (איחוד)
4. **`(inter <t1> <t2>)`**: a type which describes the set containing the intersection of the values in `<t1>` and `<t2>` (חיתוך)
5. **`(diff <t1> <t2>)`**: a type which describes the set containing the values in `<t1>` that are not in `<t2>` (set difference) (הפרש בין קבוצות)
6. **`(any -> is? number)`**: a type predicate, which is a function of one parameter that returns a boolean value and informs the type checker that if the value is true, the parameter belongs to the type, or otherwise that the value does not belong to the type.

The type predicate mirrors the one familiar from TypeScript. For example, the following:

```
(define (isNumber : (any -> is? number))
  (lambda ((x : any)) : is? number
    (number? x)))
```

Is semantically equivalent to the TypeScript function:

```
const isNumber = (x: any): x is number => typeof x === "number";
```

2.1 We defined the type constructors `diff`, `inter`, and `union` according to set-theoretic principles. For each of the following TExps, write a simplified TExp denoting the same set of values. **[3 pts]**

a. `(inter number boolean)`
b. `(inter any string)`
c. `(union any never)`
d. `(diff (union number string) string)`
e. `(diff string (union number string))`
f. `(inter (union boolean number) (union boolean (diff string never)))`

2.2 Complete the following code snippet (i.e. offer replacements to the **[bracketed gaps]**) such that it will pass type checking in L52 but not in L51. **[8 pts]**

```
;; L52 return type is? boolean
;; L51 return type boolean
(define (isBoolean : (any -> is? boolean))
  (lambda ((x : any)) : is? boolean
    (boolean? x)))

;; Function to complete
(define (good_in_L52 : ((union number boolean) -> [a]))
    (lambda ((z : (union number boolean))) : [b]
        [c]
    ))
```

2.3 Complete the return type for `f` in L52. Explain. **[7 pts]**

```
(define (is_number? : (any -> is? number))
    (lambda ((x : any)) : is? number
        (number? x)))

(define (is_boolean? : (any -> is? boolean))
    (lambda ((x : any)) : is? boolean
        (boolean? x)))

(define f
    (lambda ((x : (union number boolean))) : [answer]
        (if (is_number? x)
            (if (> x 0)
                "positive"
                "negative")
            (if (is_boolean? x)
                x
                1))))
```

Question 3: Implementing L52 [70 points]

In the template, you are given the entire source code for L5. You may modify **all** the files in the template, except for the `tests` file. In addition, we have added skeleton functions with their signature. We advise you to use them. Question 2 implicitly included crucial hints to some of the implementation details you would need to pay attention to.

Syntax

In this section you will be required to update the syntax of L5 to support type predicates and the new data types. Remember, you must add the necessary constructors and predicates.

Part 1 - Extending basic types

Below we provide the concrete syntax for `Any` and `Never`. Modify the L5 syntax to support these data types. **[15 points]**

```
10   // ;; <any-te>        ::= any
11   // ;; <never-te>      ::= never
```

Part 2 - Supporting complex types

Below we provide the concrete syntax for `Union` and `Intersection`. In the template code we already added `Union` to the syntax of L5. Modify the L5 syntax to support the `Intersection` data-type as well. This will involve changes in the `TExp` module.

```
17   // ;; <union-te>     ::= (union <texp> <texp>) // union-te(components: list(te))
18   // ;; <inter-te>     ::= (inter <texp> <texp>) // inter-te(components: list(te))
```

In order to support this type, you would also need to implement `makeDiffTExp = (te1: TExp, te2: TExp): TExp`, which returns a `TExp` of the set difference between `te1` and `te2` or `te1\te2` in set notation, and modify `isSubType` to include the new data types. Make sure that after application of multiple operations, perhaps involving `any` and `never` types, the list of type components is flat (no unnecessary internal hierarchy), ordered lexicographically (use the given `superTypeComparator` function), and contains no duplicates. If the outermost element is an `Intersection`, normalize it to Disjunctive normal form using the provided `dnf` function.

Note: Any TExp is assumed as an atomic type expression and follows the rule:
`makeDiffTExp(any, y) = any` for all y such that `y != any`.

Also Note: The `dnf` function assumes the structure of Intersection type expression contains components as a key that includes all of the intersection sub-TExps. **[25 points]**

Part 3 - Type predicates

Below we provide the concrete syntax for a type predicate. Modify the L5 syntax to support the type predicates. This will entail changes in the structure and parsing of `procExps` in the AST, redefining procedural type expressions as a disjoint union with predicate type expressions in the `TExp` file, and handling type predicate syntax in the `typeOfApp` function and the `typeOfProc` function in the `typecheck` module. **[20 points]**

```
1  // <cexp> ::= <number>
2  //          ...
3  //          ...
4  //          |  ( lambda ( <var-decl>* ) [: [is?]? <TExp>]? <cexp>+ )
5  //          ...
6  //          ...
```

Note: L5 type predicates must only accept one argument.

Part 4 - type predicate effects

The place where type predicates make a difference is the `if` special form. As such, rewrite the `typeofIf` function in the type checking module to support the type predicate logic. This will require implementing the `isTypePredApp` function as well.

In your implementation, take into consideration the new complex and basic type added in parts 1 and 2. Think about possible implications of evaluating predicate types on the type of the whole expression. **[10 points]**