Programming Languages

Departamento de Engenharia Informática, Técnico Lisboa

MEIC P4 24.25

Project Statement (Phase 2 v0.1)

© Luís Caires (luis.caires@tecnico.ulisboa.pt | https://luiscaires.org)

Objectives of Phase 2

Implement an interpreter for the X++ functional-imperative language

- The starting point for Phase 2 is your interpreter for L1++. You are expected to:
  - Base your implementation in the big-step environment-based semantics studied in the course, extended with the following features.

- Baseline (graded for up to 18)
  - Static Type-checking and Type Definitions
  - Labeled Product and Labeled Sum Types with width and depth subtyping

- Points of Excellence (each of the 2 items below graded +1/20)
  - SasyLF type preservation proof for (a fragment of) the X++ type system (see note on bonus description later in this doc).
  - Recursive Type Definitons and Static Type checking for these Recursive types

X++ (Functional Core) done in Phase 1

$x, y, z \in$ *Var*

M, N *(Terms)* $::=$

$|\ x$ *(variable)*

$|\ b$ *(boolean)*

$|\ n$ *(integer)*

$|\ M\ op\ M$ *(operation)*

$|\ \lambda x{:}A.M\ |\ MN$ (lambda-calculus)

$|$ let $x = N$ in M *(definition)*

$|$ if(M, N, R) *(conditional)*

X++ (Mutable state) done in Phase 1

$x, y, z \in$ *Var*

M, N *(Terms)* $::=$

$\ldots$

$|\, \ell$ *(reference)*

$|$ box(M) $|$ $!M$ $|$ $M := N$ *(state)*

$|$ $M; N$ $|$ while(M, N) *(actions)*

X++ (Phase 2 - static typed language)

$x, y, z \in$ *Var*

M, N *(Terms)* $::=$

$\cdots$

$\mid \{\ l_1 = M_1, \ldots, l_n = M_n\ \}$ *(record)*

$\mid M.l$ *(field select)*

$\mid l(M)$ *(variant)*

$\mid$ match M of $\{\ l_i(x) \rightarrow N_i\ \}$ *(case)*

$\mid M :: N \mid$ nil *(lists)*

$\mid$ match M of $\{$ nil $\rightarrow N \mid x :: y \rightarrow M\ \}$ *(list case)*

A, B *(Types)* $::=$

$\mid$ int *(integers)*

$\mid$ bool *(booleans)*

$\mid A \rightarrow B$ *(functional type)*

$\mid$ ref(A) *(reference type)*

$\mid$ list(A) *(list type)*

$\mid \{l_1 : A_1, \ldots, l_n : A_n\}$ *(labeled product type)*

$\mid [l_1 : A_1, \ldots, l_n : A_n]$ *(labeled sum type)*

X++ (Phase 2 - concrete syntax)

- **Type Expressions** % see grammar fragment in the drive folder
  - TE ::= TF (**->** TE) ? % right associative
  - TF ::= **unit** | **int** | **bool** | **string** | **ref** <T> | **list** <T> |

    struct { (**id :** T)* } | **union** { (**id :** T)* } | **(** TE **)** | **id**

  - **You will need to create ASTType nodes to represent the AST of type expressions.**

- **Scoped Type Definitions** % add to the grammar rule for the Let non-terminal as in
  - Let ::= Seq | (**let id** = T;)+Seq | (**type id** = T;)+ Seq

X++ (Phase 2 - notes on semantics)

- The semantics of X++ conforms with the big step operational semantics covered in the lectures.
- X++ will retain the lazy lists of Phase 1, with static type **list**<T> for any type T.
- The **match** construct will apply identically to union types and to lists, using the same syntax for list patterns as in Phase1, and *label*(x) patterns for unions, where label is a union type label.
- Strings values will include string literals, equipped with a concatenation operation, syntactically overloaded with the arithmetic addition operator +.
- Moreover, concatenation of a string with a value of any other type will convert the later to its tostr() representation, e.g the expression “foo = “+(2*3) will evaluate to the string “foo = 5”.

Static Type Checking

SubTyping Rules

$$\frac{}{A <: A} \qquad \frac{A <: B \quad B <: C}{A <: C} \qquad \frac{A <:> B}{\text{ref}(A) <: \text{ref}(B)} \qquad \frac{C <: A \quad B <: D}{A \to B <: C \to D}$$

$$\frac{A_i <: B_i \quad (\text{all } i \in 1 \ldots k)}{\{l_1 : A_1, \ldots, l_k : A_k, \ldots, l_n : A_n\} <: \{l_1 : B_1, \ldots, l_k : B_k\}}$$

$$\frac{A_i <: B_i \quad (\text{all } i \in 1 \ldots k)}{[l_1 : A_1, \ldots, l_k : A_k] <: [l_1 : B_1, \ldots, l_k : B_k, \ldots, l_n : B_n]}$$

Static Type Checking

Typing Rules (Basic Functional Core)

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \Gamma \vdash n : \mathsf{int} \qquad \Gamma \vdash b : \mathsf{bool}$$

$$\frac{\Gamma \vdash M : \mathsf{int} \quad \Gamma \vdash N : \mathsf{int}}{\Gamma \vdash M + N : \mathsf{int}} \qquad \frac{\Gamma \vdash M : \mathsf{int} \quad \Gamma \vdash N : \mathsf{int}}{\Gamma \vdash M \leq N : \mathsf{bool}} \qquad \frac{\Gamma \vdash M : \mathsf{bool} \quad \Gamma \vdash N : \mathsf{bool}}{\Gamma \vdash M \mathbin{\&\&} N : \mathsf{bool}}$$

$$\frac{\Gamma \vdash N : C \quad C <: A \quad \Gamma \vdash M : A \to B}{\Gamma \vdash MN : B} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}$$

$$\frac{\Gamma \vdash M : \mathsf{bool} \quad \Gamma \vdash M : A \quad \Gamma \quad \Gamma \vdash R : A}{\Gamma \vdash \mathsf{if}(M, N, R) : A} \qquad \frac{\Gamma \vdash N : B \quad \Gamma, x : B \vdash M : A}{\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M : A}$$

Static Type Checking

Typing Rules (References)

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{box}(M) : \mathsf{ref}(A)}$$

$$\frac{\Gamma \vdash M : \mathsf{ref}(A)}{\Gamma \vdash\ !M : A} \qquad \frac{\Gamma \vdash M : \mathsf{ref}(A) \quad \Gamma \vdash N : A}{\Gamma \vdash M := N : A}$$

Static Type Checking

Typing Rules (lists)

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N :: \mathsf{list}(A)}{\Gamma \vdash M :: N : \mathsf{list}(A)} \qquad \frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{list}(A)}$$

$$\frac{\Gamma \vdash M : \mathsf{list}(A) \quad \Gamma \vdash N : C \quad \Gamma, x{:}A, y{:}\mathsf{list}(A) \vdash R : C}{\Delta \vdash \mathsf{case}\ M\ \mathsf{of}\ \{\ \mathsf{nil} \to N \mid x :: y \to R\ \} : C}$$

Static Type Checking

Typing Rules (Products and Sums)

$$\frac{\Gamma \vdash M_i : A_i \quad (\text{all } i = 1, \ldots, k)}{\Gamma \vdash \{l_1 = M_1, \ldots, l_n = M_n\} : \{\ l_1 : A_1, \ldots, l_n : A_n\ \}}$$

$$\frac{\Delta \vdash M : \{\ l_1 : A_1, \ldots, l_n : A_n\ \}\ (l_i \in l_1, \ldots l_n)}{\Delta \vdash M.l_i}$$

$$\frac{\Gamma \vdash M : A_i}{\Gamma \vdash l_i(M) : [\ l_1 : A_1, \ldots, l_n : A_n\ ]}$$

$$\frac{\Delta \vdash M : [\ l_1 : A_1, \ldots, l_n : A_n\ ] \quad \Delta, x{:}A_i \vdash N_i : C}{\Delta \vdash \text{case } M \text{ of } \{\ l_i(x) \rightarrow N_i\ \} : C}$$

SASYLf proof

- This part will address the mechanised proof (in SasyLF) of type safety for a fragment of our language where we just consider pairs as the sole product types.

- Your starting point is the file **SasyLFSafe.slf** to be found in the course Google drive.

- You should add the missing typing rules for the pairs and complete the proof of **theorem preservation: forall dt: * |- e : tau forall ds: e -> e' exists * |- e' : tau.**

- You wil get an **extra bonus credit 1/21** if instead of pairs you model labeled records in SasyLF.

Recursive Types

- The X++ language should support recursive types like in the following example.

```
type Btree = union {  Nil: unit, Node: NodeT } ;
type NodeT = struct { left: Btree, val: int, right: Btree };
let countNodes =
     fn t:Btree => {
        match t {
               Nil(_) -> 0
          |    Node(p) -> 1 + (countNodes (p.left)) + (countNodes (p.right))
        }
     };  … ;;
```

- More examples and tests will be released early next week in the Course Drive.

Some Practical Features

1 - the executable should be executable from a sh script named “x++” as in

```
luis@macbook ~ > x++
# 2;;
2
^D
luis@macbook ~
```

2 - We should also be able to run code from a file indicated in the command line as in

```
luis@macbook ~ x++  hello.xpp
1 2 3 4 5 6 7 8 9 10
luis@macbook ~
```

Submission Instructions

1 - Submit all code to a gitlab / github repo shared with me (luis.caires@tecnico.ulisboa.pt)

2 - Include a small report (pdf) briefly explaining how you have implemented the static type checker and possibly the recursive types extension.

NOTE: the contents of the shared repo **must not be changed** after the deadline.

3- You may use the software available in the gitlab RNL standard installation and javacc (installation requested).

4 - Include a top level script “makeit”, that I may use to compile all your source code by typing “\.makeit” at the command line.

5- Include some X++ code examples to demonstrate your interpreter. You must include original new examples of your own, not just the ones I gave in the lectures.

Use the Slack to clarify any aspects