CSE 526, Spring 2020: HW4 Supplementary Material

Last Updated: 05/05/2020 at 1:49pm

In this homework you are expected to write an interpreter for the lambda-calculus-based language with all the simple extensions we have gone through in class (Chaps. 3, 9, and 11 of the text book). This document relates the material in the text book with the Prolog-based front-end supplied to you for HW4.

1 Syntax and Representation

The following relates the syntactic forms for different language aspects/features as described in the text book, with the abstract syntax representation as Prolog terms, as well as the concrete syntax accepted by the Prolog-based front end (parser) supplied as a part of HW4 material.

1.1 Booleans

This is from language **B** in Chapter 3.

Book notation	Informal meaning	Abstract syntax	Concrete Syntax
t	*Terms*	`E`	E
`true`	*constant true*	`true`	`true`
`false`	*constant false*	`false`	`false`
`if` t `then` t `else` t	*conditional*	`if(`E_1`,`E_2`,`E_3`)`	`if(`E_1`,`E_2`,`E_3`)`

We use E to denote expressions. For abstract syntax, `E` denotes Prolog terms representing the abstract syntax of the expressions. For concrete syntax, E represents the textual notation used to write the expressions.

1.2 Natural Numbers

This is from language **NB** in Chapter 3.

in addition to Sec. 1.1			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
`0`	*constant zero*	`zero`	`0`
`succ` t	*successor*	`succ(`E_1`)`	`succ` E_1
`pred` t	*predecessor*	`pred(`E_1`)`	`pred` E_1
`iszero` t	*zero test*	`iszero(`E_1`)`	`iszero` E_1

In addition to `0`, the front end successfully parses natural numbers (sequence of digits) and converts them to the successor notation. For instance, the front end treats 2 in a manner equivalent to `succ succ 0`.

1.3 Lambda

This is from language $\lambda_{\rightarrow}$ in Chapter 9.

We use T to represent type expressions. For abstract syntax, `T` denotes Prolog terms representing the abstract syntax of the type expressions. For concrete syntax, T represents the textual notation used to write the type expressions.

The notation also includes the following: a in abstract syntax denotes a Prolog atom. v in concrete syntax denotes an identifier, which is a sequence of alphanumeric or underscore character beginning with an alphabet.

in addition to those up to Sec. 1.2			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
Expressions			
x	*variable*	`var`(a)	v
λx:T. t	*abstraction*	`lambda`(a, `T`, `E`)	`\` v `:` T `.` E
t t	*application*	`app`($\mathtt{E}_1$, $\mathtt{E}_2$)	E_1 E_2
Types			
`nat`	*natural numbers*	`nat`	`nat`
`bool`	*Booleans*	`bool`	`bool`
$T \rightarrow T$	*type of functions*	`arrow`($\mathtt{T}_1$, $\mathtt{T}_2$)	T_1 `->` T_2

In addition to the constructs from simply typed lambda calculus, $\lambda_{\rightarrow}$, we will allow an implicitly-typed abstraction with the same syntax as OCAML. More precisely, we will permit an implicitly typed abstraction, written in concrete syntax as `fun` v `->` E, and represented in abstract syntax as `lambda`(a, `E`).

1.4 Unit and Sequence

This is from Chapters 11.2 and 11.3.

in addition to those up to Sec. 1.3			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
Expressions			
`unit`	*constant* `unit`	`unitconst`	`()`
$t_1; t_2$	*sequence*	`seq`($\mathtt{E}_1$, $\mathtt{E}_2$)	E_1 `;` E_2
Types			
`Unit`	*unit type*	`unittype`	`unit`

1.5 Let Bindings

This is from Chapter 11.5.

in addition to those up to Sec. 1.4			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
Expressions			
`let` x = t `in` t	*let binding*	`let`(a, E$_1$, E$_2$)	`let` v `=` E_1 `in` E_2 `end`

In contrast to OCAML and the book's notation, note that the concrete syntax for this homework has "`end`" to mark the end of a let-binding.

In addition to the above notation for `let`, the front end allows a more expressive form for defining functions. The front end can successfully parse text of the form

`let` f x_1 x_2 $\cdots$ x_n `=` e_1 `in` e_2 `end`

and generate an abstract syntax term equivalent to

`let` f `=` `fun` x_1 `->` (`fun` x_2`->` $\cdots$ (`fun` x_n`->` e_1)$\cdots$) `in` e_2 `end`

This syntactic sugar makes function definitions easier to write and read. Note, however, that this derived form uses an implicit type (OCAML-style) for the defined function.

1.6 Pairs

This is from Chapter 11.6.

in addition to those up to Sec. 1.5			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
Expressions			
{t, t}	*pair*	`pair`(E$_1$, E$_2$)	`(` E_1 `,` E_2 `)`
t.1	*first projection*	`fst`(E)	E `. 1`
t.2	*second projection*	`snd`(E)	E `. 2`
Types			
$T_1 \times T_2$	*product type*	`product`(T$_1$, T$_2$)	T_1 `*` T_2

Note the change in notation for projection when going from the concrete syntax to abstract syntax. The components of a pair are accessed using the "dot" notation in concrete syntax, but this is represented in abstract syntax using two different symbols for projecting the first and the second components (`fst` and `snd`, respectively).

1.7 Records

This is from Chapter 11.8.

in addition to those up to Sec. 1.6			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
Expressions			
$\{l_i = t_i^{i=1..n}\}$	*record*	`record`(L) where L is a list whose i-th element is of the form $(a_i, \mathtt{E}_i)$	`{` v_1`=`E_1 `,` v_2`=`E_2 `,` ... v_n`=`E_n `}`
t.l	*projection*	`project(E,` a`)`	E `.` v
Types			
$\{l_i : T_i^{i=1..n}\}$	*type of records*	`recordtype`(L) where L is a list whose i-th element is of the form $(a_i : \mathtt{T}_i)$	`{` v_1`:`T_1 `,` v_2`:`T_2 `,` ... v_n`:`T_n `}`

1.8 Variants

This is from Chapter 11.10.

in addition to those up to Sec. 1.7			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
Expressions			
`case` t `of` $\langle l_i = x_i \rangle \Rightarrow t_i^{i=1..n}$	*case*	`case`$(E,\ L)$ where L is a list whose i-th element is of the form $(a_i, a_i', \mathtt{E}_i)$	`case` E `of` `<`v_1`=`v_1'`>=>`E_1 `|` `<`v_2`=`v_2'`>=>`E_2 `|` ... `<`v_n`=`v_n'`>=>`E_n
$\langle l = t \rangle$ as T	*tagging*	`tag(`a`, E, T)`	`<` v = E `>` `as T`
Types			
$\langle l_i : T_i^{i=1..n} \rangle$	*type of variants*	`varianttype`(L) where L is a list whose i-th element is of the form $(a_i : \mathtt{T}_i)$	`<` v_1`:`T_1 `,` v_2`:`T_2 `,` ... v_n`:`T_n `>`

1.9 Recursion

This is from Chapter 11.11.

in addition to those up to Sec. 1.8			
Book notation	Informal meaning	Abstract syntax	Concrete Syntax
Expressions			
`fix` t	*fixed point of t*	`fix(E)`	`fix` E
`letrec` x:T = t `in` t	*recursive let binding*	`letrec(`a, `T`, $\mathtt{E}_1$, $\mathtt{E}_2$`)`	`letrec` a `:` T `=` E_1 `in` E_2 `end`
`letrec` x = t `in` t	*recursive let binding (implicitly typed)*	`letrec(`a, $\mathtt{E}_1$, $\mathtt{E}_2$`)`	`letrec` a `=` E_1 `in` E_2 `end`

Similar to the treatment of `let`, the front end allows a more expressive form for defining recursive functions. The front end can successfully parse text of the form

`letrec` f x_1 x_2 $\cdots$ x_n `=` e_1 `in` e_2 `end`

and generate an abstract syntax term equivalent to

`letrec` f `=` `fun` x_1`->` (`fun` x_2 `->` $\cdots$ (`fun` x_n `->` e_1)$\cdots$) `in` e_2 `end`

This syntactic sugar makes function definitions easier to write and read.

2 Syntactic Conventions

2.1 Comments

A concrete program may contain comments, which extend from `(*` and end at the next `*)`. All text between these sentinels are ignored when generating the abstract syntax tree. A comment may occur wherever a whitespace is permitted.

2.2 Precedence and Associativity

The front end treats application, assignment and sequencing to be the three main binary operators. All of them associate to the *right*. Note that for application, this associativity is unnatural. Among these operators, application has the highest precedence and sequence has the lowest precedence.

Record/pair selection using the "dot" operator is considered as a unary operation, since there is only one expression involved in that operation. Among unary operations, "dot" has the lowest precedence. Other operators such as `succ`, `ref` etc. have equal (high) precedence.

Parantheses "`(`" and "`)`" can be used to group operations.

For types, the two binary type constructors "`->`" and "`*`" have the same precedence and are both right associative. As in the case of expressions, parantheses can be used to group type expressions as well.

3 Organization

The material for this HW is split into different files:

1. `lexer.pl`: Converts a string to a sequence of tokens.
2. `parser.pl`: Converts a string to an abstract syntax tree; uses `tokenize` defined in `lexer.pl`.

   You may want to experiment with the parser to get a clear understanding of the mapping from concrete syntax to abstract syntax.
3. `util.pl`: Offers `subst(E1, X, E2, E3)`, a predicate that given an expression `E1`, a variable `X` and another expression `E2` substitutes all free occurrences of `X` in `E2` with `E1`. The result of the substitution is returned in `E3`. This predicate performs alpha renaming as needed.
4. `toplevel.pl`: Implements wrappers to parse contents of a file and to (1) evaluate an expression given as a string or in a file by repeated single-step evaluations; and (2) evaluates the type of a given expression.

   Predicate `singlestep(E1, E2)` is used in `toplevel.pl` to represent single-step evaluation: expression `E1` evaluates to expression `E2` in one step. This predicate needs to be implemented (in `interp.pl`); a template of the definitions for this predicate is given in `interp.template`. The template encodes the single step call-by-value semantics for the simply-typed lambda calculus and Booleans.

   Predicate `typeof(G, E, T)` is used in `toplevel.pl` to represent type judgement: expression `E` has type `T` in a given type environment `G`. This predicate needs to be implemented (in `typeinfer.pl`); a template of the definitions for this predicate is given in `typeinfer.template`. The template encodes the type checking rules for the simply-typed lambda calculus and Booleans.

Note that all the given material is geared for use with SWI Prolog as well as XSB Prolog.