User manual

Luca D'angelo Sabin
l.dangelo@udc.es

Jesús Mosteiro García
jesus.mosteiro@udc.es

1 Introduction

This is the manual for our modified version of the lambda calculus interpreter studied in programming languages design. We will go into detail on how to use the extended capabilities, ignoring what was already possible with the third version of the interpreter. For a more in depth look into what technical changes were required to implement these features, we added comments in the code where aproppiate.

2 Basics

2.1 Multi-line expressions

The new version of the interpreter supports line breaks in the middle of a expression. This means that it is now needed to indicate the end of each expression explicitly. For this reason, we have added the token `;;`, which indicates the end of a statement.

```
>> succ


1;;
2
: Nat
```

2.2 Pretty-printing

The function used to create the displayed strings from a term has been replaced. The new version attempts to minimize the number of parenthesis needed to display expressions by taking into account the precedence of each token in the grammar.

It also attempts to make outputs more readable by adding line breaks and indentation where aproppiate.

Here is a comparison between the old and the new printing, both run over the same recursive function:

Input:

```
letrec sum : Nat → Nat → Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
 in
sum;;
```

Old:

```
(lambda n:Nat. (lambda m:Nat. if (iszero (n)) then (m) else (succ ((((fix (
lambda sum:(Nat) → ((Nat) → (Nat)). (lambda n:Nat. (lambda m:Nat. if (iszero (n
)) then (m) else (succ (((sum pred (n)) m)))))))) pred (n)) m))))) : (Nat) → ((
Nat) → (Nat))
```

New:

```
lambda n:Nat.
  (lambda m:Nat.
    if
      iszero n
    then
      m
    else
      succ ((fix lambda sum:Nat → (Nat → Nat).
        (lambda n:Nat.
          (lambda m:Nat.
            if
              iszero n
            then
              m
            else
              succ (sum (pred n) m)
          )
        )) (pred n) m)
  )
: Nat → (Nat → Nat)
```

While the old version generated a difficult to read output, with 60 parenthesis; the new one reduces that number to just 20, while making everything easier to understand.

2.3 Subtyping

This version of the interpreter implements a very simple version of subtyping via polymorphism. Certain types can be subtyped by others in the cases we know the subtype fulfills all the requirements necessary to be used interchangeably in place of the supertype.

Subtyping is recursive in types that reference other types. As an example of this, consider the arrow type. An arrow type `Nat -> A` may be subtyped by `Nat -> B` if `B` is a subtype of `A`.

Details on how subtyping works with other data types are included in their dedicated sections. Here are two more extensive examples.

2.3.1 Example A: Printing pets

```
let print_pet =
  lambda pet : {name: String, age: Nat, species: String}.
    (print_string pet.name; print_string " is a ";
     print_nat pet.age; print_string " year old ";
     print_string pet.species; print_string "."; print_newline())
in (print_pet {name: "John", age: 5, species: "bat"};
    print_pet {name: "Ursula", age: 27, species: "giraffe", height: "like a lot"};
    print_pet {sea_creature: true, age: 2711, name: "Brian", species: "whale",
    epicness: 255})
;;
```

In this example, we define a function to print information about a pet, which is represented by a record with fields corresponding to its name, age, and species.

Then we use the function on 3 other records which fulfill the subtyping.

We can observe how changing the order of the entries and having additional entries doesn't stop us from using the function.

However, if these 3 fields didn't exist in the record with these same names and types, we would get a type error.

The type of the function `print_pet` is `{name: String, species: String, age: Nat} -> Unit`.

2.3.2 Example B: Ordering tuples with associated data

```
sort_tuples = lambda f : ((Nat,) → (Nat,) → Bool). lambda l : [(Nat,)].
  letrec insert : [(Nat,)] → (Nat,) → [(Nat,)] =
    lambda l : [(Nat,)]. lambda x : (Nat,).
      if isnil[(Nat,)] l then (Nat,)[x] else (
        if f (head[(Nat,)] l) x then
          cons[(Nat,)] x l
        else
          cons[(Nat,)] (head[(Nat,)] l) (insert (tail[(Nat,)] l) x)
      )
  in letrec internal : [(Nat,)] → [(Nat,)] → [(Nat,)] =
    lambda l : [(Nat,)]. lambda acc : [(Nat,)].
      if isnil[(Nat,)] l then acc else internal (tail[(Nat,)] l) (insert acc (head
    [(Nat,)] l))
  in internal l (Nat,)[]
;;

is_tuple_more_equal = letrec r : (Nat,) → (Nat,) → Bool =
  L a:(Nat,). L b:(Nat,).
    if iszero a.1 then false else (
      if iszero b.1 then true else r (pred a.1,) (pred b.1,)
    )
in r;;

not = lambda x : Bool. if x then false else true;;

sort_tuples is_tuple_more_equal
  (Nat,)[(4, "data_goes_here"), (3, {name: "value"}), (1, "sorted"), (7, true)];;
sort_tuples (L a:(Nat,). L b:(Nat,). not (is_tuple_more_equal a b))
  (Nat,)[(4, "epic"), (3, {name: "value"}), (1, "sorted"), (7, true)];;
```

This is a more complex example featuring subtyping of tuples, functions, and lists.

In it, we define a function that sorts a list of tuples based on each of the tuples' first elements, which must be of type `Nat`. It takes as argument a comparator function, as well as the list.

Then we define another function to use as comparator for the sorting, which also takes tuples subtyping `(Nat,)`.

Finally, we can use these functions to sort data according to indices indicated by the first element of their tuples. Any tuple can be subtyped into the lists without issue as long as its first element is of type Nat.

The type of `sort_tuples` is `((Nat,) -> ((Nat,) -> Bool)) -> ([(Nat,)] -> [(Nat,)])`.

2.4 Running from source

When building the interpreter, a new executable is generated alongside the familiar `top`: `run`.

`run` receives the path to a source file containing statements as a command line argument.

Before running the program, it checks for lexer, syntax, and typing errors. In case it finds one, it will be reported, along with the line number it appeared at. Once it has ensured the input is valid, it runs all of it silently, only displaying IO operations.

2.4.1 Example

```
append = letrec append : [Nat] → [Nat] → [Nat] =
  lambda l1 : [Nat]. lambda l2 : [Nat]. if isnil[Nat] l1 then l2 else cons[Nat]
 (head[Nat] l1) (append (tail[Nat] l1) l2) in
append;;

name = print_string "Welcome! Please input your name: "; read_string ();;
print_string "Hi, "; print_string name; print_string "!"; print_newline ();;

print_newline (); print_string "Please input a sequence of numbers bigger than
0, and 0 when done."; print_newline ();;
```

```
numbers = letrec input : [Nat] → [Nat] =
  lambda acc : [Nat]. let n = read_nat () in
    if iszero n then acc else input (append acc Nat[n])
in input Nat[];;

print_newline (); print_string "Here are the numbers you typed: ";
letrec print : [Nat] → Unit =
  lambda l : [Nat]. if isnil[Nat] l then () else (
    if isnil[Nat] (tail[Nat] l) then print_nat (head[Nat] l) else (
      print_nat (head[Nat] l); print_string ", "; print (tail[Nat] l)
    )
  )
in print numbers;
print_string "."; print_newline ();;
```

This program can be run as a terminal application by saving it to a source file. Its use of IO operations makes it a good demonstration of the capabilities of the interpreter in this mode.

3 Data types

3.1 String

The String type has been added to the interpreter to represent sequences of characters. It is based on OCaml strings. This type can be referred in the interpreter as `String` when writing typed lambda expressions, and a string chain surrounded by `"` (double quotes) when writing atomic values.

```
>> f = L x:String. x;;
>> f "hello";;
"hello"
: String
```

3.1.1 ”^” operator

It is a binary operator used to concatenate strings.

```
>> "string 1" ^ "string 2";;
"string 1string 2"
: String
```

3.1.2 Subtyping String

The string type may not be subtyped.

3.2 Unit

The Unit type represents the lack of a value and it works very similarly to how it does in OCaml. Its value can be written as `()`, while the type is denoted simply as `Unit`.

Its main use case is as an argument to runnable expressions that don't require any valued arguments, and as the value runnable expressions that don't need to return one take once evaluated. An example of this are the IO kernel functions.

3.2.1 Using semicolons to separate expressions

Along with the unit type and IO operations, we have included a way to separate expressions. By adding `;` in between two expressions, we can chain them like so:

```
print_string "Test "; print_nat 1; print_newline ();;
```

This causes them to be evaluated sequentially, with the resulting values of all but the final expression being ignored.

We achieve this result by translating `t1; t2` to `(L x:Unit.t2) t1` during parsing.

3.2.2 Subtyping Unit

We made the decision to allow any type to subtype Unit. Since the unit type has no contents and isn't expected to support any operations, this means that any value that is passed as a unit type will be ignored.

This behavior can be useful to evaluate expressions we don't need the resulting value of (but some other resulting effect, like IO operations), for example as part of `;` chains.

```
read_nat (); 1;;
```

In this example, the user is prompted to input a Nat, but the input is ignored and the expression evaluates to `1`. This is similar to using `ignore` in OCaml, but implicit.

3.3 List

The List type encapsulates a series of elements, all of the same explicit type. The type itself can be referred to as `[<inner_type>]`, where any type may be used as the element type.

Two types of constructors are supported for lists:

First, they can be created by chaining, using `cons[<inner_type>]` and `nil[<inner_type>]`:

```
>> cons[Nat] 7 (cons[Nat] 8 nil[Nat]);;
Nat[7, 8]
: [Nat]
```

This way of constructing lists is more faithful to how they are represented internally, as a recursive chain of `TmCons` terms (our lists are not supported by OCaml lists).

Alternatively, a type name followed by square brackets is the simplified, more convenient to use syntax:

```
>> Nat[4,5,6];;
Nat[4, 5, 6]
: [Nat]
```

When a list is pretty-printed, it is usually displayed in a format similar to this simplified constructor. However, if the list is not a pure value (it contains expressions that aren't considered values), it is displayed via `cons[<inner_type>]` and `nil[<inner_type>]`.

Lists support the following basic operations (Note that their type is explicitly specified):

3.3.1 head[<inner_type>]

Evaluates as first element in the list.

```
>> head[Nat] Nat[4,5,6];;
4
: Nat
```

3.3.2 tail[<inner_type>]

Evaluates as the list containing all elements but the first one.

```
>> tail[Nat] Nat[4,5,6];;
Nat[5, 6]
: [Nat]
```

3.3.3 isnil[<inner_type>]

Returns a `Bool` indicating whether the list is empty.

```
>> isnil[Nat] Nat[];;
true
: Bool
```

3.3.4 Examples

Get the length of a `[Nat]`.

```
length = letrec length : [Nat] → Nat =
  lambda l : [Nat]. if isnil[Nat] l then 0 else succ (length (tail[Nat] l)) in
length;;

length Nat[1,2,3,4,5,6];;
```

Append a `[Nat]` to another.

```
append = letrec append : [Nat] → [Nat] → [Nat] =
  lambda l1 : [Nat]. lambda l2 : [Nat]. if isnil[Nat] l1 then l2 else cons[Nat]
 (head[Nat] l1) (append (tail[Nat] l1) l2) in
append;;

append Nat[1,2,3] Nat[4,5,6];;
```

Map a `[Nat]` using a given `Nat -> Nat`.

```
map = letrec map : (Nat → Nat) → [Nat] → [Nat] =
  lambda f : (Nat → Nat). lambda l : [Nat]. if isnil[Nat] l then l else
  let val = f (head[Nat] l) in
    cons[Nat] val (map f (tail[Nat] l)) in
map;;

map (prod 2) Nat[1,2,3,4,5,6];;
```

3.3.5 Subtyping List

Lists may be subtyped by another list whose element type is a subtype of this list's element type.

3.4 N-Tuples

N-Tuples aggregate a series of ordered terms, which may be of different types. These terms can be accessed basedd on their position in the tuple via projection. Tuples are based on OCaml lists. Their type is written as `(<inner_type>, <inner_type>,...)`.

```
>> x = (1, "string");;
>> x.1;;
1
: Nat
>> x.2;;
"string"
: String
```

Attempting to access a data field out of bounds causes an error to be thrown.

```
>> x.3;;
type error: Tuple projection out of bounds
```

3.4.1 1-Tuples

Tuples of length one may be defined by adding a comma before the closing parenthesis on an expression. This is similar to python's syntax for their 1-Tuples. This is to avoid ambiguity with single terms enclosed in parenthesis.

```
>> (1,);;
(1,)
: (Nat,)
```

Even though this may seem pointless at first, tuples of a single element can be useful when combined with subtyping, as showcased in subtyping example B.

3.4.2 Subtyping N-Tuples

An N-Tuple may be subtyped by another tuple with the same length or longer than it, if an only if all of the positions that exist in the supertype tuple exist with compatible subtypes in the subtype tuple.

3.5 Records

Records are sets of key-value pairs. Each of the values is a term that can be accessed via named projection. Ordering of the record entries does not ever matter. They are based on an OCaml list of pairs. Their type is written as `{<name>: <inner_type>, <name>: <inner_type>,...}`.

```
>> x = {f1: 1, f2: "str ", f3: (), f4: 9, f5: true};;
>> if x.f5
    then print_string "true"
    else "false";;
true()
: Unit
>> x.f2 ^ x.f2;;
"str str "
: String
```

Whenever we try to access the term corresponding to a ket that doesn't appear in the record, an error is thrown.

```
>> x.g;;
type error: Record projection didn't match
```

3.5.1 Subtyping Records

Records may be subtyped by other records that contain every key in the original record, each corresponding to a type that subtypes the one present in the original.

4 Expressions

4.1 Recursive definitions

Recursive lambda support has been added, and its syntax is similar to the one in OCaml. To define a recursive expression, we use the `letrec` keyword (as opposed to `let`). Internally, the fix combinator is responsible for this functionality, and the translation to it happens during parsing.

4.1.1 Examples

Sums up two `Nat`s.

```
sum = letrec sum : Nat → Nat → Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in sum;;

sum 27 11;;
```

Computes the product of two `Nat`s, using the previously defined `sum`.

```
prod = letrec prod : Nat → Nat → Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then 0 else sum (prod (pred n) m)
 m
in prod;;

prod 27 11;;
```

Gets the nth element of the Fibbonacci sequence, using the previously defined `sum`.

```
fib = letrec fib : Nat → Nat =
  lambda n : Nat. if iszero n then 0 else (if iszero (pred n) then 1 else sum (
fib (pred n)) (fib (pred (pred n))))
in fib;;

fib 11;;
```

Computes the factorial of a `Nat`, using the previously defined `prod`.

```
fact = letrec fact : Nat → Nat =
  lambda n : Nat. if iszero n then 1 else prod (fact (pred n)) n
in fact;;

fact 5;;
```

4.2 Global Definitions

This version of the interpreter supports a new type of statement, alternative to just inputting a term.

Global definitions have a very simple syntax in the form of

```
>> name = "value";;
```

As they affect the global context, it was necessary to restructure the interface the top level has with the lambda module.

During implementation, it was crucial to consider the possibility of redefinitions. We made sure to handle them appropriately for a functional context. That is, values that have already been captured by lambdas remain unchanged, and do not act as references to the global context.

```
a = 1;;
b = lambda x : Unit. a;;
a = 2;;
b ();;
```

In this example, the final statement evaluates as `1`, as expected.

We determined the simplest way to achieve this result was evaluating and replacing all appearances of global definitions before storing new global binds, "capturing" those values.

4.3 Input and Output

The following IO "kernel" functions are supported:

4.3.1 print_nat

Prints the natural number provided to stdout. Acts as if was type `Nat -> Unit`.

```
>> print_nat 10;;
10()
: Unit
```

4.3.2 print_string

Prints the string provided to stdout. Acts as if was type `String -> Unit`.

```
>> print_string "Test";;
Test()
: Unit
```

4.3.3 print_newline

Prints a newline character to stdout. Acts as if was type `Unit -> Unit`.

```
>> print_newline ();;

()
: Unit
```

4.3.4 read_nat

Gets a nat value from stdin. Acts as if was type `Unit -> Nat`.

```
>> read_nat ();;
1
1
: Nat
```

4.3.5 read_string

Gets a string value (single line) from stdin. Acts as if was type `Unit -> String`.

```
>> read_string ();;
Test
"Test"
: String
```