OCanren Documentation

JetBrains Research

Mar 20, 2023

CONTENTS

CHAPTER

ONE

THE TUTORIAL

1.1 Say "Hello World!" in OCanren

We first execute the program, and then we read and explain through the source code line by line.

1.1.1 Executing the Program

The source code has to be compiled and linked, for which you would need the Makefile. Now open terminal under your local copy ofthe `helloWorld` directory, and:

```
make
```

This would produce a native-code file `hello.opt`, execution of which by:

```
./hello.opt
```

will print `hello world!` in your terminal.

1.1.2 Reading the Program

The first line:

```
open OCanren
```

makes the names from the module OCanren available for later use.

1.1.3 Important Interfaces

The source code of the module OCanren resides in OCanren.ml (It does not have an accompanying `.mli` file). Inspecting the content thereof, we shall see that basically it includes three modules Logic, Core and module Stream, and finally defines the Std module. You should open the module OCanren at the beginning of every source file where you use OCanren features.

The second line:

```
let str = !!("hello world!\n")
```

associates the value name `str` with the expression that is the prefix operator `!!` (named *primitive injection*) applied to the string literal `"hello world!\n"`.

1.1.4 Internal Representation

The operator `!!`, provided by the module Logic, makes type conversion to the OCanren internal representation, something like what a calculator program does when it receives an input string "1" and converts it to the integer or floating-point type for further processing. We could see from the interface that:

```
val (!!) : 'a -> ('a, 'a logic) injected
```

The `injected` type constructor is provided by the module Logic as an abstract type (so we do not concern ourselves with its implementation).

More information about various typed being used and the meaning of type parameters of the *('a, 'b) injected* could be get from *Digesting the Types*.

Logic Variables

The `logic` type constructor which appears in the type of `!!` above is also provided by the module `OCanren.Logic`. It takes one type parameter and its type representation is exposed: we could see from the module interface that it has two constructors `Var` and `Value`, representing respectively a *logic variable* and a *concrete value* over/of the parameter type, in the sense that wrt. the arithmetic expression `1 + x` we say that `x` is a logic variable over the integer type and `1` is a concrete value of the integer type.

1.1.5 Making a Query

The 3rd line:

```
let _ =
  List.iter print_string @@
  Stream.take ~n:1 @@
  run q (fun q -> ocanren { q == str }) project
```

is divided into three sub-expressions by the right associative infix operator `@@` that is provided by OCaml's core library Stdlib. The most important sub-expression is: `run q (fun q -> ocanren { q == str }) project` whose most important part is: `q == str`.

Note: The piece of code on the discussion uses an OCanren-specific syntax extension that doesn't appear in other languages of miniKanren family. See *OCanren-specific Camlp5 syntax extension* for details.

Next we start with explaining the inner most and the most important part: `q == str`, followed by its immediate enclosing environment which is the *run* statement:

```
run q (fun q -> ocanren { q == str }) project
```

and finally the top most expression for taking and printing answers.

1.1.6 Syntactic Identity and Unification

Syntactic identity between two expressions $expr_1$ and $expr_2$ (of the same type) is denoted using two equation symbols: $expr_1 == expr_2$. Usually we are given two different expressions both of which have zero or more sub-expressions considered as logic variables, and we are interested in finding (type sensitive) substitutes for these logic variables so that the resulting expressions are syntactically identical. Finding such substitutes is know as *unification* of the original expressions.

Example In order for both `(x + 1) == (2 + y)` and `Node (x,1) == Node (2,y)` to be true, we replace `x` by `2` and `y` by `1`, making both sides of == the expression `2 + 1` or `Node (2,1)` respectively. We now unified `(x + 1)` with `(2 + y)`. Moreover, `Node (x,1)` is unified with `Node (2,y)`.

Example How to substitute the logic variable `z` so that `z == "hello world!"` ? Trivial: replacing `z` with the constant `"hello world!"`. This is essentially what our program does: solving a unification problem.

1.1.7 The OCanren Top Level: the *run* expression

We can parse the `run ...` expression following the syntax below, which is given in EBNF except that occurrences of the meta-identifier `etc` signifies omission: there is no single syntactic category named `etc`.

```
top level = 'run',  size indicator, goal, answer handler ;

size indicator =  'one' | 'two' | 'three' | 'four' | 'five'
                | 'q'   | 'qr'  | 'qrs'   | 'qrst' | 'qrstu'
        | '(', size indicator, ')'
        | 'succ', size indicator ;

goal = 'fun', parameters, '->', goal body ;

parameters = etc ;

goal body = 'ocanren', '{', pretty goal body, '}' ;

pretty goal body = etc ;

answer handler = 'project' | etc ;
```

A goal asks: what values shall be assumed by the parameters so that the proposition as given by the goal body (in which these parameters are expected to occur) holds?

The `ocanren { }` environment in a goal body instructs the Camlp5 preprocessor to transform on the syntactic level the `pretty goal body` into the more verbose and less intuitive calls of OCanren's functions. As a reference, the rules by which the preprocessing is done is given in `pa_ocanren.ml` where we could find, for example, the == symbol is transformed into the name `unify`.

Todo: fix links

The number of parameters shall agree with the size indicator, where `q` ... `qrstu` are just alternative names for `one` ... `five` respectively. If there are more than five parameters, the successor function `succ` can be applied to build larger size indicators, e.g., `(succ (succ five))` is for seven parameters.

The answer handler is a type converter from the OCanren internal representation to a user-level representation. When there is no free logic variables in the answer we use `project`. The `Not_a_value` exeption (provided by OCan-

ren.Logic) is thrown if we use `project` as the handler but the answer contains free logic variables: in this case some other handler shall be used.

Todo: Reference reifiers

The `run` function and the size indicators are provided by module `OCanren.Core`. Basic answer handlers are provided by module `OCanren.Logic`.

1.1.8 Taking and Displaying Answers

The top level constructs a lazy stream out of which an arbitrary number of answers could be pulled, subject to answer availability. We use a stream instead of a finite list to hold the answers because generally the set of all answers is enumerable. Omission of the optional argument `~n` of `take` means "take all". Finally we use OCaml standard functions to iterate through the taken list of answers and print each one in the terminal. Our program has only one answer.

1.2 Digesting the Types

OCanren is designed for typed relational programming. Two points here: it is typed, and it is relational. We shall now study how to work with the types. This clears the way so that we can then focus on the relational part.

We have seen that the OCanren internal representation of a string has a type of the form `('a, 'b) injected` and we have named it an *injected type*, referring to the injection of data from user level representation into the internal representation. This type expression involves several subtleties that are, when combined together, not apparent. In this lesson we break down such type expressions into their very components, so that the reader can appreciate the construction of these internal types and can build his own.

1.2.1 Fully Abstract Types

First we need a notion of *abstract type*. OCaml also has a notion of abstract type which refers to a type constructor whose equation and representation are hidden from the user and is considered incompatible with any other type. However, the fully abstract type that we are talking about here is a different concept, and it comes from the fact that some recurive types can be defined in the following way.

Todo: Mention where fully abstract types were 1st time introduced.

Say we want to define a polymorphic list type:

```
module MyList = struct
  type ('a, 'b) t = Nil | Cons of 'a * 'b
end
```

The type constructor `MyList.t` is called an *fully abstract list type* for it not only abstracts over the list member type by means of the type parameter `'a`, but also (and more importantly) abstracts over the list tail type or in other words over the list type itself by means of the type parameter `'b`. We can use the fully abstract list type to define other useful types of lists, as we shall see next.

1.2.2 Ground Types

The usual definition of the recursive list type can be decomposed into the three finer steps:

1. Abstracting over the self type.
2. Instantiating the abstract type by self type.
3. Equating the instance with the self type to close the loop.

As in:

```
(** Defining the ground list type from the abstract type *)
module MyList = struct
  type ('a, 'b) t = Nil | Cons of 'a * 'b   (* 1 (step 1) *)
  type 'a ground = ('a, 'b) t
      constraint 'b = 'a ground             (* 2 (steps 2 & 3) *)
end
```

Equation `(* 1 *)` is for step 1. Equation `(* 2 *)` is for steps 2 and 3: if you instantiate `'b` with `'a ground` in `(* 1 *)`, you would get (literally):

```
type ('a, 'a ground) t = Nil | Cons of 'a * 'a ground  (* 1b *)
```

Then by `(* 1b *)` and `(* 2 *)` we have:

```
type 'a ground = Nil | Cons of 'a * 'a ground  (* 2b *)
```

Equation `(* 2b *)` is the usual definition of a list type, which we call a *ground list type*.

The abstract list type can also be used to define logic list types.

Note: The type definition *type 'a ground = ('a, 'a ground) t* above require an non-conventional compiler switch *-rectypes*. It allows more liberal types definitions by disable infamously known in the area of logic programming "occurs check". Without this switch we can't not construct a type ground list which type system considers equal to predefined `Stdlib.list` . While using this switch we pay by seeing less trivial type errors in compile time.

Todo: Actually in the moment we can't declare the list type which is the same as predfined one. We need a small patch.

1.2.3 Logic Types

In a relational program, a list engages with logic variables (like `X, Y, Z`, capitalized as in Prolog) in cases like:

1. `Cons (1,Nil)` and `Nil` — No logic variable occurrence at all. The lists are actually ground.
2. `Cons (X, Nil)` and `Cons (X, Cons (Y, Nil))` and `Cons (1, Cons (X, Cons (Y, Nil)))` — There are only unknown list members.
3. `Cons (1,Y)` — There is only an unknown sub-list.
4. `Cons (X,Y)` and `Cons (X, Cons (Y, Z))` and `Cons (X, Cons (3, Cons (Y, Z)))` — There are both unknown list members and an unknown sub-list.
5. `X` — The list itself is wholly unknown.

Due to possible presence of logic variables in various ways shown above, the concept of a list in a relational program is more general than the concept of a ground list. We call them *logic lists*, for which we now define a type.

Observe that for cases 1-4, we have some knowledge about the structure of the list: we know whether it is empty or not because there is a top level constructor to inspect. We call such logic lists *guarded*.

Todo: I would recommend to use the term *partially ground* instead of *guarded*. What do you think, Yue Li?

But for case 5, we have no idea about the structure of the list for there is no top level constructor to provide a clue: we call it a *pure logic list*, which is just a logic variable. This is an important distinction needed for typing logic lists, and we summarize it as follows:

```
logic list          = pure logic list
                    | guarded logic list;

pure logic list     = logic variable;

guarded logic list  = 'Nil'
                    | 'Cons', '(', logic list member, logic list, ')';
```

The type for a (polymorphic) logic list can then be implemented with mutual recursion as follows:

```
(** A logic list type definition *)
type 'b logic_list  =  Value of 'b guarded_logic_list
                    |  Var   of int * 'b logic_list list
and  'b guarded_logic_list  = ('b, 'b logic_list) MyList.t
```

where the constructors `Value` and `Var` are used to distinguish a guarded logic list from a pure logic list. Moreover, The `Var` constructor's `int` argument uniquely identifies a pure logic list, and the second argument is a (possibly empty) list of logic lists that can be used to instantiate the pure logic list.

Todo: Say explicilty about disequalty constraints

Todo: Discuss with Yue Li why concept of guarded types is 'illuminating'.

Example. Below are some inhabitants of the type `int logic_list` :

```
(** case 1: a guarded logic list *)
Value Nil
(** case 1: a guarded logic list which is an integer
 *  cons'ed to another guarded logic list *)
Value (Cons (1, Value Nil))
(** case 3: a  guarded logic list which is an integer
  * cons'ed to a pure logic list*)
Value (Cons (1, Var (1,[])))
(** case 5: a pure logic list *)
Var (1,[])
```

In all examples above we could see that the inhabitants are logic lists where logic variables may only denote unknown sub-lists. This is because the parameter of `logic_list` is instantiated by a ground type (`int`). To allow logic variables as list members (as in cases 2 and 4), we need to define the type of *logic number* and use it as the type parameter instead of `int`, as follows.

We define the Peano numbers. A *Peano number* is a natural number denoted with two symbols `O` and `S` with auxiliary parentheses `()`. The symbol `O` is interpreted as the number zero, and the symbol `S` a successor function. Then the number one is denoted `S(O)`, two `S(S(O))`, three `S(S(S(O)))` and so on. Peano numbers are frequently used in relational programming, where they appear like: - `O`, `S(O)` — Ground (Peano) numbers. - `X`, `S(X)`, `S(S(X))` — Numbers with a logic variable `X`.

Regarding all these as *logic numbers*, we distinguish:

- `X` — The pure logic number.
- `O`, `S(O)`, `S(X)`, `S(S(X))` — Guarded logic numbers.

We can define abstract, ground and logic Peano number types as well:

```
(** Abstarct, ground and logic Peano number types *)
module Peano = struct
  type 'a t    = O | S of 'a             (** Abstract *)
  type ground  = ground t                (** Ground *)
  type logic   = Value of guarded        (** Logic  *)
               | Var of int * logic list
  and  guarded = logic t                 (** ... and Guarded *)
end
```

Similar to logic lists, a logic number is either

- a pure logic number (e.g., `X`), or
- a guarded logic number that is either `O` or `S` applied recursively to a logic number.

Pure and guarded logic numbers are again distinguished using constructors `Var` and `Value` respectively.

Example. Below are some inhabitants of the type `Peano.logic` :

```
(** a pure logic number X *)
Var (1,[])
(** a guarded logic number which is the constructor [O] *)
Value O
(** a guarded logic number S(X) which is the constructor [S] applied to
   a (pure) logic number X *)
Value (S (Var (1,[])))
(** a guarded logic number S(O) which is the constructor [S] applied to
   a (guarded) logic number which is the constructor [O] *)
Value (S (Value O))
(** a guarded logic number S(S(X)) *)
Value (S (Value (S (Var (1,[])))))
```

Then the type `Peano.logic logic_list` has the following inhabitants:

```
Value Nil                                          (* case 1 *)
Value (Cons (Value (S (Value O)) , Value Nil))     (* case 1 *)
Value (Cons (Var (1,[]), Value Nil))               (* case 2 *)
Value (Cons (Value (S (Value O)) , Var (2,[])))    (* case 3 *)
Value (Cons (Var (1,[]), Var (2,[])))              (* case 4 *)
Var (1,[])                                         (* case 5 *)
```

Therefore, when we talk about a list of numbers in relational programming, we are actually talking about a logic list of logic numbers.

More abstraction over logic types

Compare the types of logic lists and logic numbers (reproduced below):

```
(* Comparing the types of logic lists and logic numbers *)

(* The logic list type *)
type 'b logic_list  =  Value of 'b guarded_logic_list
                    |  Var   of int * 'b logic_list list
and  'b guarded_logic_list  = ('b, 'b logic_list) MyList.t

(* logic number type. Excerpt from module Peano *)
type logic   = Value of guarded
             | Var of int * logic list
and  guarded = logic t
```

We could see that they both involve the constructors `Value` and `Var` with similar argument structures: the `Value` constructor's argument is always a guarded type, and the `Var` constructor's first argument is always `int` and second argument is always a `list` of the logic type itself. This imlpies that we can extract these common parts for reuse , by equating them to a new type constructor with one type parameter that abstracts from the guarded types, as follows:

```
(** The new, reusable type constructor for defining logic types *)
module MyLogic = struct
  type 'a logic = Value of 'a | Var of int * 'a logic list
end
```

Next time when we what to define `('a1, ..., 'an) Something.logic`, instead of writing:

```
(** longer logic type definition  *)
module Something = struct
  type ('a1, ..., 'an, 'self) t = (* ... type information omitted *)
  type ('a1, ..., 'an) logic = Value of ('a1, ..., 'an) guarded
                             | Var of int * ('a1, ..., 'an) logic list
  and ('a1, ..., 'an) guarded = ('a1, ..., 'an, ('a1, ..., 'an) logic) t
end
```

we could write:

```
(** shorter logic type definition  *)
module Something = struct
  type ('a1, ..., 'an, 'self) t = (* ... type information omitted *)
  type ('a1, ..., 'an) logic =  ('a1, ..., 'an) guarded MyLogic.logic
  and ('a1, ..., 'an) guarded = ('a1, ..., 'an, ('a1, ..., 'an) logic) t
end
```

for we can derive the longer from the shorter using `MyLogic` (the reader may write down the derivation as an exercise). As examples: the logic list type can be rewritten as:

```
(** Defining the logic list type using [MyLogic.logic] *)
module MyList = struct
  type ('a, 'b) t = Nil | Cons of 'a * 'b
  type 'b logic   =  'b guarded MyLogic.logic
  and 'b guarded  = ('b, 'b logic) t
end
```

and the logic number type as:

```
(** Defining the logic number type using [MyLogic.logic] *)
module Peano = struct
  type 'a t   = O | S of 'a
  type logic  =  guarded MyLogic.logic
  and guarded = logic t
end
```

Or even shorter, skipping the guarded types:

```
(** Concise definitions of abstract and logic types
    for lists and Peano numbers *)

module MyList = struct
  type ('a, 'b) t = Nil | Cons of 'a * 'b
  type 'b logic   =  ('b, 'b logic) t MyLogic.logic
end

module Peano = struct
  type 'a t   = O | S of 'a
  type logic  =  logic t MyLogic.logic
end
```

1.2.4 Injected Types

The `injected` type constructor collects the corresponding ground and logic type constructors, to which we assign the name `groundi` (read "groun-dee"):

Todo: Rename groundi to injected in the source code, and in the tutorial after that

```
(** Complete definitions of injected types
    for lists and Peano numbers *)

module MyList = struct
  type ('a, 'b) t = Nil | Cons of 'a * 'b
  type 'a ground = ('a, 'a ground) t
  type 'b logic =  ('b, 'b logic) t MyLogic.logic
  type ('a, 'b) groundi = ('a ground, 'b logic) injected
end

module Peano = struct
  type 'a t = O | S of 'a
  type ground = ground t
  type logic =  logic t MyLogic.logic
  type groundi = (ground, logic) injected
end
```

The `injected` type constructor is abstract in the sense that its type information is hidden from the user. Therefore we do not concern ourselves as to what an inhabitant of an injected type looks like.

Injecting non-recursive types

This is even simpler: no need to abstract over self.

For example, logic pairs:

```
module MyPair = struct
   type ('a1, 'a2) t = 'a1 * 'a2
   type ('a1, 'a2) ground = ('a1, 'a2) t
   type ('b1, 'b2) logic =  ('b1, 'b2) t MyLogic.logic
   type ('a1, 'a2, 'b1, 'b2) groundi = (('a1, 'a2) ground, ('b1, 'b2) logic) injected
end
```

We can now talk about:

```
(** Pair of Peano numbers *)
module PP = struct

  (** Ground pairs of ground Peano numbers, like (O, O) and (O, S(O)) *)
  type ground = (Peano.ground, Peano.ground) MyPair.ground

  (** Logic pairs of logic Peano numbers, like (X, S(Y)), Y and (X, X) *)
  type logic = (Peano.logic, Peano.logic) MyPair.logic

  (** Injected pairs of Peano numbers (abstract type) *)
  type groundi = (Peano.ground, Peano.ground, Peano.logic, Peano.logic) MyPair.groundi
            (* = (ground, logic) injected *)

end

(** Peano number * Peano number list --- Pairs *)
module PPL = struct
  type ground = (Peano.ground, Peano.ground MyList.ground) MyPair.ground
  type logic  = (Peano.logic,  Peano.logic MyList.logic) MyPair.logic
  type groundi = (* = (ground, logic) injected *)
    (Peano.ground,
     Peano.ground MyList.ground,
     Peano.logic,
     Peano.logic MyList.logic) MyPair.groundi
end
```

As an exercise, the reader may define the injected types for pairs of polymorphic lists, and lists of polymorphic pairs.

Injecting non-regular recursive types

A non-regular recursve type is a parameterized type constructor in whose recurisve definition at least one type parameter is instantiated (See also this). Injection of non-regular recursive types is not discussed here, and, frankly speaking, never required in relational progrmming in OCanren.

1.2.5 Compiling the Program

The types that we learnt in this lesson are put together in the file digTypes.ml which can be compilied successfully using the lightweight Makefile, where we need the `-rectypes` compiler option to deal with the rather liberal recurisve types that appear in this lesson.

The use of `MyLogic.logic` and `MyLogic.injected` instead of (resp.) `OCanren.logic` and `OCanren.injected`

Note that we defined the module `MyLogic` for pedagogical purposes only, so that we do not have to refer to the OCanren package during compilation. The reader is encouraged to find the corresponding definitions in the OCanren module Logic by himself.

1.2.6 Conclusion

OCanren works on injected types that are defined via abstract, ground and logic types. The table below organizes these types into four levels by dependency.

Level No.	Level Name
1	Fully Abstract
2	Ground
3	Injected
4	Logic

In principle, OCanren can be implemented without injected types by performing unification on logic types. But it will hurt the performance a lot. Detailed motivation about 'injected' typed they can get in the *paper Typed Embedding of Relational Programming Language*.

We give templates for definig injected types:

```
open OCanren

(** Template of an injeced, regular recursive type *)

module Something = struct
  type ('a1, ..., 'an, 'self) t = (* ... add type information here *)
  type ('a1, ..., 'an) ground = ('a1, ..., 'an, ('a1, ..., 'an) ground) t
  type ('b1, ..., 'bn) logic =  ('b1, ..., 'bn, ('b1, ..., 'bn) logic) t OCanren.logic
  type ('a1, ..., 'an, 'b1, ..., 'bn) groundi = (('a1, ..., 'an) ground, ('b1, ..., 'bn)␣
↪logic) injected
end

(** Template of an injeced, non-recursive type *)

module Something = struct
  type ('a1, ..., 'an) t = (* ... add type information here *)
  type ('a1, ..., 'an) ground = ('a1, ..., 'an) t
  type ('b1, ..., 'bn) logic =  ('b1, ..., 'bn) t OCanren.logic
  type ('a1, ..., 'an, 'b1, ..., 'bn) groundi = (('a1, ..., 'an) ground, ('b1, ..., 'bn)␣
↪logic) injected
end
```

The reader may apply these templates to define his own types. OCanren is for typed relational programming.Two points here: it is typed, and it is relational. We have now studied how to work with the types. This clears the way so that we can then focus on the relational part.

Allusion to OCanren standard libraries

As examples, we defined types of Peano numbers, and polymorphic lists and pairs, each showing the four-level structure. The `Peano`, `MyList` and `MyPair` modules correspond to the OCanren standard libraries `OCanren.Std.Nat`, `OCanren.Std.List` and `OCanren.Std.Pair` respectively.

1.3 A Simple Data Base

In this lesson we learn the basic form of relational programming: defining and querying a data base. In particular, we build a toy data base of the 32 control characters in the ASCII table, associating each character with its integer number and description, for example: `BS` with number 8 and description "Back space".

The program is a bit long, yet simple in the sense that the relation defined therein is not recursive: it is a straightforward listing of the data base, which is the very basic form of relational programs.

1.3.1 Reading the Program

The structure of the program is as follows:

1. The initial opening statement
2. Type definitions and utilities
3. The ASCII control characters type - Injection utilities
4. The logic string type
5. The data base as a relation `ascii_ctrl`
6. Some queries on the database

Read the definition of `ascii_ctrl` as:

Values *c*, *n* and *s* form the relation *ascii_ctrl* iff *c* is NUL and *n* is 0 and *s* is the string "Null", or *c* is SOH and *n* is 1 and *s* is the string "Start of heading", or ..., or *c* is US and *n* is 31 and *s* is the string "Unit separator".

Read the query:

```
(** Find the control characters in a given range *)
let _ =
  List.iter print_endline @@
    Stream.take ~n:18 @@
      run q (fun s ->
          ocanren {fresh c,n in Std.Nat.(<=) 0 n
                                & Std.Nat.(<=) n 10
                                & ascii_ctrl c n s}) project
```

as: Print at most 18 possible values of *s*, such that exist some *c* and *n* where *n* ranges from 0 to 10 inclusive, and the tuple *(c, n, s)* satisfies the relation *ascii_ctrl*.

OCanren will print the following (eleven) strings:

```
Null
Start of heading
Start of text
End of text
End of transmission
Enquiry
Ackonwledge
Bell
Back space
Horizontal tab
Line Feed
```

We could see that the relational program specifies a relation, and it has been used to find missing elements of a tuple that is claimed to satisfy some constraint of which the relation is a part. In so doing, we did not tell the program *how* to find these missing elements. It was the semantics of the programming language that did this automatically. We explain the syntax and semantics next.

1.3.2 Syntax of a Formula

The notion of a formula in OCanren is different from that in logic programming, i.e., the Horn clause subset of first-order predicate logic. Instead it is quite close to formulae in the system MALL.

A formula is either atomic, or is compound and built from atomic formulae using conjunction (`&`), disjunction (`|`) and existential quantification (`fresh`). Atomic formulae are built from predicate symbols followed by their arguments. There are only two predicate symbols `==` and `=/=`. A formula is allowed to be infinitely long. Formulae can be abbreviated by finitely-represented (possibly recurisive) definitions.

Example. Atomic, compound, named and infinite formulae:

- `x == y` and `1 =/= 2` are two atomic formulae.
- By the definition `foo x y := x == 1 & y =/= 2`, we can use `foo x y` to abbreviate the compound formula `x == 1 & y =/= 2`.
- By the recursive definition `is_nat x := x == 0 | fresh y in x == S y & is_nat y` we can use `is_nat x` to abbreviate the infinitely long formula:

```
x == 0
| fresh y1 in
  x == S y1 & { y1 == 0
              | fresh y2 in
                y1 == S y2 & { y2 == 0
                             | fresh y3 in
                               y2 == S y3 & { ... }}}
```

We now give the concrete syntax of a formula in OCanren.

```
formula  = atomic formula
     | compound formula
     | named formula
     | '{', formula, '}' ;

atomic formula = value, '==', value | value, '=/=', value;

compound formula = formula, '&', formula
```

(continues on next page)

(continued from previous page)

```
                | formula, '|', formula
        | 'fresh', lparams, 'in',  formula;

named formula = formula name, ' ', values;

formula name definition = 'let', ['rec'], let-binding, {'and', let-binding};

let-binding =  formula name, [':', typexpr, '->', 'goal' ], '=',
               'fun', fparams, '->', 'ocanren','{', formula, '}' ;

lparams = param, {',', param};
fparams = param, {' ', param};
values  = value, {' ', value};
```

The scope of `fresh...in` extends as far as possible. `&` binds tighter than `|`. A formula always has the type `goal` (this type constructor is provided by the module Core). The braces `{}` could be used for explicit grouping, as in `{ x == 1 | x == 2 } & y == 0`.

1.3.3 A Note on the Concept of a *Goal*

In logic programming, we call the formula which we want to refute a *goal*. This term (i.e., goal) is inherited by the modern successor of logic programming, which is called *relational programming*. However, the semantics of a *goal* nevertheless changes: it is no longer something that we want to refute, but something for which we want to find variable substitutions so that it is true. In other words:

- Logic programming is proof by contradiction: we want to find variable substitutions so that a formula *F* is true, but what we do is to find substitutions so that the negation of F is false.
- Relational programming is proof by straightforward construction without the logical detour of "negation of negation".

1.3.4 The Semantics of a Formula

A formula has two semantics: the *declarational semantics* and the *operational semantics*. The way in which the reader is advised to read the relation definition and the query is actually part of the declarational semantics. The operational semantics concerns how the answers shall be searched for (mechanically), which is part of the implementation of the language.

Declaratively, the two predicate symbols == and =/= means respectively "syntactic equality" and "syntactic disequality". The logic connectives mean as usual, and a value just denote itself as a syntactic object. The operational semantics of OCanren is a set of stream manipulation rules attached to the logic connectives and the predicate symbols, and formulae are viewed as functions taking a stream member as input and returning a stream. We explain the operational semantics of OCanren in more detail below. Firstly the concept of a *stream*.

Streams

A stream is a generalization of a list: members of a list can be put on one-on-one correspondence with members of some *finite* subset of the natural numbers, whilst members of a stream can be put on one-on-one correspondence with members of some possibly infinite subset of the natural numbers. Intuitively, the imaginary, infinitely long sequence of all natural numbers itself is an example of a stream. The sequence of all integers `...-3 -2 -1 0 1 2 3...` is a stream too, equivalently `0 1 -1 2 -2 3 -3 ...` is a stream of integers too, but they are in different order than in previous stream.

The set of all streams can also be defined in the more technical, *coinductive* manner as follows:

1. Let **FS** be an operator whose input is a set of sequences and whose output is also a set of sequences. A sequence is said to be composed of its members drawn from a set of possible members.
2. The output of **FS** is constructed by:
    1. Starting with an empty set, to add members to it incrementally;
    2. Adding the empty stream;
    3. Extending each sequence of the input set with an arbitrary member, then adding the results.
3. The set St of all streams is the *largest* set that is a fixed-point of **FS**, in other words, **FS** (St) = St and St is a superset of st for all **FS** (st) = st.

Example If we restrict sequence members to integers, and let the input be `{123, 111}`, which is the set whose members are the sequences `123` and `111`. One possible output of **FS** operating on the input is `{e, 0123, 5111}` where `e` is the empty stream. Another possible output is `{e,1123, 1111}`. In neither case the output equals the input, which is quite usual. The two notable exceptions are the set Lmin of all lists of integers, and the set Lmax of all finite and infinite sequences of integers. They are both fixed-points of **FS**, known as the *least fixpoint* and the *greatest fixpoint*. Lmax is also the set of all streams of integers.

Note that in a typical inductive specification we could require that the set being defined is the samllest fixed-point. Here instead we ask for the *largest*, hence the *coinductive manner*.

Todo: It looks like very complex description of a stream but maybe it is only for me

---

Substitution

A *substitution* is a list of substitution components. A *substitution component* (for short: *component*) is a pair (*lvar*, *value*) where *lvar* is a logic variable. A substitution component (*lvar*, *value*) can be *applied* to some value *valuepre*, so that all occurrences of *lvar* in *valuepre* are simultaneously replaced by *value*, and the result is another value *valuepost*. A component is *applicable* if applying it would make a difference. To apply a substitution is to repeatedly apply its components until none is applicable.

Example Applying `[(x, Cons(1,y)); (y, Cons(2,z)); (z, Nil)]` to `Cons(0,x)` results in: `Cons(0, Cons(1,Cons(2,Nil)))`.

Formulae as Stream Builders

A formula is a stream builder as far as the operational semantics is concerned. It takes a substitution $subst_{in}$ as input and returns a stream of substitutions as output:

$subst_{in} \xrightarrow{\text{formula}} subst_{out}, subst_{out}, subst_{out}, \ldots$

For each substitution *substout* in the returned stream, applying the concatenation *substin* ^ *substout* makes the formula true in the sense of the declarational semantics.

Todo: Yue Li, what you meant saying 'applying concatenation'?

Example. Given as input the empty substitution `[]` :

- The formula `x == Cons(1, Nil)` returns the stream that consists of the substitution `[(x, Cons(1,Nil))]`.
- The formula `x == Cons(1, Nil) & y == Cons(2, x)` returns the stream that consists of the substitution `[ (x, Cons(1,Nil)); (y, Cons(2,x)) ]`.
- The formula `is_nat x` returns the stream that consists of the substitutions `[(x, O)]`, `[(x, S(y1));(y1, O)]`, `[ (x, S(y1)); (y1, S(y2)); (y2, O) ]`, ...
- The formula `1 == 1` returns the stream whose only member is `[]`.
- The formula `1 == 2` returns the empty stream: there is no way to make the formula true.

Disjunction as stream interleaving

Example. Let s_1 denote the stream of all positive intergers, and s_2 the stream of all negative intergers. The result of interleaving s_1 with s_2, denoted $s_1|zip|s_2$ is `1, -1, 2, -2, ...`, and $s_2|zip|s_1$ is `-1, 1, -2, 2, ...`.

The disjunction $F_1 \mid F_2$ of two formulae F_1, F_2 is itself a formula on the top level, so it is a stream builder, taking a substitution as input and returns a stream of substitutions. It builds the output stream by interleaving the two streams built separately by F_1 and F_2, both of which share the same input as their immediate top level formula. In more formal terms:

$$(F_1 \mid F_2)\ substin \quad = \quad (F_1\ substin)\ |zip|\ (F_2\ substin)$$

Every substitution from the output stream (concatenated with the input) makes either of the two disjuncts true.

Conjuction as a Stream Map-Zipper

To *map-zip* a stream builder F with a stream $s := m_1, m_2, m_3, ...$ (denoted $F\ mzip\ s$), is to apply F individually to each member m_k of the stream, resulting in streams s_k, and then zip all s_k together.

$$\begin{aligned} & Fmzips \\ = \quad & F\ mzip\ m_1,\ m_2,\ m_3, \ldots \\ = \quad & F\ m_1 zip\ (F\ m_2\ zip\ (Fm_3\ zip\ (\ldots))) \\ = \quad & s_1\ zip\ (s_2\ zip(s_3\ zip(\ldots))) \end{aligned}$$

Example. Let F be a stream builder that works like this: $F\ n = n, n, n, \ldots$ Then:

$$
\begin{aligned}
& F\ mzip\ 1,2,3 \\
= \quad & F_1\ zi\ (F_2\ zip\ F_3) \\
= \quad & 1,1,1,\ldots\ zip\ (2,2,2,\ldots\ zip\ 3,3,3,\ldots) \\
= \quad & 1,1,1,\ldots\ zip\ 2,3,2,3,\ldots \\
= \quad & 1,2,1,3,1,2,1,3,\ldots
\end{aligned}
$$

A conjunction $F_1 F_2$ provides the input substitution to F_1 first, and then map-zips the output of F_1 with F_2 :

$(\mathrm{F}_1\ \ F_2)subst_{in}\ =\ F_2 mzip(F_1 subst_{in})$

Every substitution from the output stream (concatenated with the input) makes both of the two conjuncts true.

1.3.5 Working with GT and Camlp5

We use packages GT and Camlp5 in OCanren programs. The influence of GT Camlp5 syntax extension is that we can use the `@type` syntax to define types, which convenienty generates useful functions for the defined type, for example, a *show* function that converts values of the defined type into a string, which we use to print the result of a query. Camlp5 expands the content of the `ocanren{}` quotation, allowing us to write readable code.

The @type syntax

In OCanren, type constructors are often defined by :

```
type definition = '@type', typedef, 'with', plugins

plugins = plugin { ',' , plugin }

plugin  ::= 'show' | 'gmap' | etc
```

where the syntactic category `typedef` is the same as that of OCaml, and the category `etc` signifies omission: the most frequently used plugins in OCanren are *show* and *gmap*, providing for the defined type a string conversion function (like Stdlib.string_of_int) and a structure preserving map function (a generalization of List.map) respectively. The other less used plugins are not shown here.

A type definition of the form `@type <typedecl> with <plugins>` is expanded at the syntactic level by GT into:

1. A type definition of the usual form `type <typedecl>`, where the value of `<typedecl>` is preserved, and
2. Several (auto-generated) plugin definitions.

The effect of syntactic transformation, including what the `@type` definitions become after expansion, can be viewed by adding the "dump source" option `-dsource` in the Makefile as explained in a comment line there. For instance, the `String` module:

```
 (** {2  The logic string type} *)
module String = struct
  @type t = GT.string with show
  @type ground = t with show
  @type logic = t OCanren.logic with show
  type groundi = (ground, logic) injected
end
```

would be expanded into this, where we could see that besides the type constructor definitions a lot more codes have actually been auto-generated to support any GT plugin that the user may request.

Note in the `LString` module that the type constructor name `string` is qualified by the module name `GT`, for we need to use the GT version of the string type which provides the useful plugins and otherwise it is the same as the OCaml built-in string type. Plugins are (auto-)created inductively: GT provides plugins for base types and rules for building plugins for compound types from component types.

Todo: Write properly part about syntax extensions and port this part there

---

Todo: I will clarify this a bit. We do not use the GT version of `string` type, in reality it is a just type alias: `module GT = struct type string = Stdlib.string ... end`. What is really happening here, is that functions for showing and gmapping string type are located in module GT. So we need 1) either write `GT.string` instead of `string` and preprocessor will generate `GT.show GT.string` instead of `GT.show string`, 2) or make `open GT` somewhere about and use type `string` without fully qualified name. **

The injection functions and the `ocanren {...}` quotation

The signature of the `ASCII_Ctrl.Inj` module shall explain itself. For every value constructor, an accompanying injection function shall be defined (either by the user or auto-generated by the tool noCanren), whose name shall be the same as the constructor name except that the first letter is set to lower case. In the `ocanren{...}` quotation, wherever a value constructor occurs, its corresponding injection function is implicitly called. Hence the `let open ASCII_Ctrl.Inj in` statement that preceeds the body of the `ascii_ctrl` relation. The quotation in the body of `ascii_ctrl` is expanded as follows:

```
let ascii_ctrl =
  (fun c n s ->
     let open ASCII_Ctrl.Inj in
     OCanren.disj
       (OCanren.conj (OCanren.unify c (nUL ()))
           (OCanren.conj (OCanren.unify n (OCanren.Std.nat 0))
             (OCanren.unify s (OCanren.inj (OCanren.lift "Null")))))
       (OCanren.disj
           (OCanren.conj (OCanren.unify c (sOH ()))
             (OCanren.conj (OCanren.unify n (OCanren.Std.nat 1))
                 (OCanren.unify s
                   (OCanren.inj (OCanren.lift "Start of heading")))))
           (OCanren.disj
             (OCanren.conj (OCanren.unify c (sTX ()))
                 (OCanren.conj (OCanren.unify n (OCanren.Std.nat 2))
                   (OCanren.unify s
                       (OCanren.inj (OCanren.lift "Start of text")))))
(* ... etc *)
```

The above code excerpt is also from what is displayed on the terminal after compiling the source with the "dump source" option `-dsource`.

1.3.6 Conclusion

A program in OCanren is understood with respect to its syntax and semantics. We define types in four levels, using the `@type` syntax of GT. We define injection functions for value constructors. We then define formulae in OCanren formula syntax, which are put in the `ocanren{}` quotation powered by Camlp5. In set theory when we think about a relation, we are actually thinking about a function *R* that can be applied to its arguments and return either true or false, like this:

$$arg_1, \ldots, arg_n \rightarrow R \rightarrow true \mid false$$

But in relational programming, when we think about a relation *R*, the most important thing is not that *R* is a function, but $R(arg_1, \ldots, arg_n)$ *in whole* is a function, i.e., we regard what is known by logicians as a formula, as a function whose input is an initial variable substitution and whose output is the set of all possible variable substitutions where each member when combined with the initial substitution makes the formula true, like this:

$$subst_{in} \rightarrow R(arg_1, \ldots, arg_n) \rightarrow subst_{out}, subst_{out}, subst_{out}, ...$$

1.4 A Library for Peano Arithmetic

We hope the reader will learn the following techniques from this lesson:

- *Advanced injection functions* Defining injection functions for value constructors of variant types, using the Fmap family of module functors `Fmap`, `Fmap2`, `Fmap3`, etc., which are provided by the module Logic.
- *Reification and Reifiers* Defining reifiers to convert data from the injected level to the logic level, again with help from the Fmap family of module functors.
- *Overwriting the show Function* Overwriting, or redefining the "show" function for values of a logic type, to allow for more concise and human readable printing of them.
- *Relations on Peano Numbers* Defining (possibly recursive) relations, e.g., comparison, addition and division on Peano numbers.
- *Scrutinizing Relations* Making queries to relations using combinations of unknown arguments.
- *Analyzing the search behaviour* Analyzing why a query returns certain answers.
- *Modifying the search behaviour* Reordering the conjuncts within the body of a relation definition to modify the way in which the relation searches for answers in a given query.
- *The trick of generate and test* Programming a relation so that answers to certain queries are found by brute-force.
- *The formula parser* Observing that the implementation of the `ocanren {}` quotation takes care of the precedence, associativity and scope of the logic connectives, and replaces constructors of variant types by injection function names, and primitive values by their injected versions.
- *Building a library* Writing and testing a library in OCanren.

The techniques are presented in detail in sections below, to which the labels (**T.1**, **T.2**, etc) are linked. Each section is self-contained and could be read independent of other sections.

The library has a systematic test file, which can be compiled (and linked) and executed by running the following shell commands in (your local copy of) the lesson directory:

```
make && ./peano.opt
```

A copy of the result of the test is answers.txt that is obtained using the shell command `./peano.opt > answers.txt`.

1.4.1 Advanced Injection Functions

The primary injection operator is `!!` which is used to cast primitive values (such as characters and strings) and constant constructors of variant types (particularly whose type constructors do not have a type parameter) from the ground level to the injected level. For those variant types whose type constructors have one or more type parameters, the primitive injection operator is inadequate. We use instead *advanced injection functions* to build injected values, which are defined using distribution functions provided by the Fmap family of module functors together with the injection helper `inj`, all from the module Logic. In our Peano Arithmetic library implementation, the following block of code defines advanced injection functions `o` and `s` for the abstract Peano number type `Ty.t`, which correspond respectively to the value constructors `O` and `S` :

```
module Ty = struct
  @type 'a t = O | S of 'a with show, gmap
  let fmap = fun f d -> GT.gmap(t) f d
end;;

include Ty
module F = Fmap(Ty)

let o () = inj @@ F.distrib O
let s n = inj @@ F.distrib (S n)
```

The general workflow of defining advanced injection functions is as follows:

1. We begin with a variant type whose type constructor `t` has one or more type parameters. This is always an OCanren abstract type, such as the abstract Peano number type or the abstract list type.
2. We count the number of type parameters of the type constructor in order to choose the suitable module functor from the Fmap family: for one type parameter, use `Fmap` ; for two type parameters, use `Fmap2`; three type parameters, `Fmap3` and so on.
3. We request the `gmap` plugin for the type constructor, and use it to define a function named `fmap` simply by renaming.
4. We put the definitions of the type constructor `t` and the `fmap` function in one module, and suppy that module as a parameter to the chosen Fmap family module functor. The result is a module `F` with three functions one of which is `distrib`, the distribution function.
5. For each value constructor of the type `t`, we define a function whose name is the same as the value constructor except that the initial letter is set to lower case. For example, `Cons`, `S` and `NUL` become respectively `cons`, `s` and `nUL`.

- For each value constructor `Constr0` of no argument, define:

```
let constr0 = fun () -> Logic.inj @@ F.distrib Constr0
```

- For each value constructor `Constr1` of one argument, define:

```
let constr1 = fun x -> Logic.inj @@ F.distrib (Constr1 x)
```

- For each value constructor `Constru` of u (> 1) arguments, define:

```
let constru = fun x1 ... xu -> Logic.inj @@ F.distrib @@ Constru (x1, ..., xu)
```

In the definition of a typical advanced injection function, the value constructor takes arguments which are at the injected level, and the combination of `inj` and `distrib` serves to inject the top level value while preserving the structure of constructor application. If we explain by a schematic where a pair of enclosing square brackets `[]` signifies the injected status of the enclosed data, we would say that:

- An advanced injection function `constr` converts a value of the form `Constr ([arg1], ..., [argn])` to a value of the form `[Constr (arg1, ..., argn)]`. In other words,
- The injection function `constr` takes arguments `[arg1], ..., [argn]` and builds a value of the form `[Constr (arg1, ..., argn)]`.

We advise the reader to find in the interface of the Logic module the Fmap module functor family and the functors' argument types (which are module types): that would provide a more formal explanation of what advanced injection functions do and why they are defined in the given manner.

1.4.2 Reification and Reifiers

Say we have a logic variable `x` and a substitution `[(x, Lam(z,y)); (y, App(a,b))]` that associates `x` with the term `Lam(z,y)` and `y` with `App(a,b)` where `y, z` are also logic variables. We would like to know what `x` is with respect to the substitution. It is straightforward to replace `x` by `Lam(z,y)` but since `y` is associated with `App(a,b)` we can further replace `y` in `Lam(z,y)`, and finally we get the term `Lam(z,App(a,b))` . Although there is still an unbound part `z`, we have no further information about how `z` might be instantiated, so we leave it there. What we have done is called *reification* of the logic variable `x` : we instantiate it as much as possible, but allowing unbound logic variables to occur in the result. A *reifier* is a function that reifies logic variables.

We know that there are primary and advanced injection functions. Similarly there are primary and advanced reifiers: the primary reifier `Logic.reify` reifies logic variables over base types (like character and string) and simple variant types (i.e., those that have only constant constructors). Advanced reifiers are for logic variables over variant types whose type constructors have one or more type parameters and there exist non-constant (value) constructors. The Peano Arithmetic library defines an advanced reifier for the Peano number type:

```
let rec reify = fun env n -> F.reify reify env n;;
```

Advanced reifiers are defined using the Fmap module functor family. The correct Fmap module functor for defining the reifier for a type is the same as that selected for defining advanced injection functions for the same type. The result of applying the correct Fmap module functor is a module that provides, besides a distribution function, a reifier builder named `reify`, e.g., `F.reify` in the case of our library. Note there is an abuse of names: the name `reify` has been used for both reifiers and reifier builders. If a type constructor takes other types are parameters, then the reifier for the top level type is built from reifiers for the parameter types: we build "larger" reifiers from "smaller" reifiers. The Peano number reifier is recursive because the Peano number type is recursive: the reader should refer to the signature of `F.reify` and see how the types of the reifier and the reifier builder fit together.

1.4.3 Overwriting the *show* Function

The default *show* function for a variant type converts values of that type to strings in a straightforward way, e.g., a logic Peano number representation of the integer 1 would be converted to the string `"Value(S(Value O))"` whilst "the successor of some unknown number" could be `"Value(S(Var(1,[])))"`. These are not too readable.

The Logic module has already redefined the *show* function for the type `Logic.logic` so that the above values would instead be converted to strings `"S(O)"` and `"S(_.1)"` respectively, omitting the verbose constructors `Value` and `Var` and displaying variables in the form `_.n` where `n` is the first parameter of the constructor `Var`. The redefinition happens within the record value `logic` which has a field `GT.plugins`. This record value origins from the `@type` definition of the type constructor `Logic.logic` and is auto-generated by the GT package. The field `GT.plugins` is an object with several methods, one of which is `show` : other plugins (or methods) keep their default meanings but `show` is redefined.

However, when there are too many repetitions of the constructor `S`, the *show* function as redefined in the Logic module is no longer suitable. Our Peano Arithmetic library therefore offers a further customized redefinition just for displaying logic Peano numbers, converting those values without free variables directly to Arabic numbers and those with free variables a sum between an Arabic number and the symbol `n`.

In like manner, the reader may: - Redefine the *show* function to behave in other ways, or - Redefine other plugins by modifying the `GT.plugins` field, or - Redefine plugins for other types.

Some additional remarks on the last point: the `@type` definition of a type constructor `typeconstr-name` generates a record value also named `typeconstr-name` of the type `GT.t`. This could be viewed by adding the `-i` option as indicated in the Makefile:

```
BFLAGS = -rectypes -g -i
```

See also the GT source.

1.4.4 Relations on Peano Numbers

This section teaches the reader how to read and write relation definitions.

The reader is already familiar with reading and writing functions in OCaml. To read a function, just look at the type annotation (if any) to determine what are the input types and what is the output type, and then inspect the function body to see how the inputs are processed to produce the output. To write a function, first decide the type of the function, and then design the internal procedure that produces the output from the input.

In OCanren, a relation is a function only at the language implementation level, and as users our experience with functions do not transfer well when it comes to reading and writing relations. That's why relational programming claims the status of being a unique programming paradigm distinct from imperative programming and functional programming. Working with relations requires learning a new way of thinking: *declarative* thinking.

Relation definitions are declarative, meaning that it first of all states a proposition. The emphasize is on "what" rather than "how". It is the language implementation that takes care of "how", but the user of the language should foucs on "what". For example, look at the addition relation:

```
let rec add a b c =
  ocanren{ a == O & b == c
         | fresh n, m in
           a == S n & c == S m & add n b m}
```

It says nothing about how to compute the sum `c` of two numbers `a` and `b`, instead it only says what conditions must be satisfied so that the addition relation exists among the three numbers `a`, `b` and `c` — if `a` equals `O` and `b` equals `c`, or, if `a` equals `S n` and `c` equals `S m` and the numbers `n,b,m` also satisfy the addition relation, for some `n,m`. No other way is given in which we can establish the addition relation among three numbers.

Another example is the "less than" relation:

```
let rec lt a b =
  ocanren{ fresh n in
           b == S n &
             { a == O
             | fresh n' in
               a == S n'
               & lt n' n }}
```

It says that `a` is less than `b` if there exist `n`, such that `b` equals `S n`, and either `a` equals `O` or there exist `n'` such that `a` equals `S n'` and `n'` is less than `n`.

Other relations in the library shall be read in this way, and they are all written with the declarative reading in mind. The reader is encouraged to write a relation for subtraction: `sub a b c` iff `a - b = c`, or, put in another way: iff `b` is `O` and `a` is `c`, or `b` is `S n` and `a` is `S n'` and `sub n' n c`.

1.4.5 Scrutinizing Relations

Taking the "less than" relation as an example, we can ask questions like:

- Is zero less than one ? Is one less than two ? Is one less than zero ? Is two less than one?
- What is less than five ? Five is less than what ?
- What is less than what ?

The first set of questions above is for *checking*: we provide concrete numbers and ask if they satisfy the relation. The remaining two sets of questions are for *searching*: looking for numbers that satisfy the relation. Note that the questions are organized: there coud be no unknown, one unknown or two unknowns, and each argument position of the relation might be an unknown. In general, for a relation of N arguments, the total number of kinds of questions we can ask is (R is the number of unknowns in NCR):

NC0 + NC1 + NC2 + ... + NCN-1 + NCN

Running the test shows that OCanren answers all the questions well. For example, the goal:

```
fun q -> ocanren { lt 0 (S 0) & lt (S 0) (S(S 0)) }
```

asks about what is `q` so that zero is less than one and one is less than two, and the answer is just a free variable `n` meaning that `q` could be any number and the relation always holds between the given numbers. The similar goal:

```
fun q -> ocanren { lt (S 0) 0 | lt (S(S 0)) (S 0) }
```

asks about what is `q` so that one is less than zero or two is less than one. There is no answer, meaning that there is no `q` to make the relation hold between the given numbers.

The goal below asks what is less than five:

```
fun q -> ocanren { lt q (S(S(S(S(S 0))))) }
```

For this goal the answers `0,1,2,3,4` are found, which is quite satisfactory.

The relations `lte, add, div, gcd` are also questioned systematically in the test file.

Note that the addition relation can perform subtraction, and the division relation can do multiplication. For instance, the goal below asks "What adds 4 equals to 7 ?" and whose answer is "3":

```
fun q -> ocanren { add q (S(S(S(S 0)))) (S(S(S(S(S(S(S 0))))))) }
```

This amounts to performing the subtraction `7 - 4`. The next goal asks "What divided by 5 equals 3 with remainder 0 ?" and the answer is "15":

```
fun q -> ocanren { div q (S(S(S(S(S 0))))) (S(S(S 0))) 0 }
```

It amounts to the multiplication `3 * 5`.

1.4.6 Analyzing the Search Behaviour

When asking the `lt` relation "what is less than 5" using the goal:

```
fun q -> ocanren { lt q (S(S(S(S(S 0))))) }                                    (G.1)
```

OCanren returns 0,1,2,3,4. Let's see why. It really is a matter of definition: we defined `lt a b` to be a certain formula `(Eq.1)` and now we substitute 5 for `b` in the formula `lt a b` followed by several steps of simplification then we get a formula `(Eq.12)` that literally says `a` shall be 0, 1, 2, 3 or 4. Below are the details.

We reproduce the definition of `lt` in the followinig simplified form:

```
lt a b = fresh n in b == S n
       & { a == 0 | fresh n' in a == S n' & lt n' n }
                                                                               (Eq.1)
```

Now replace `b` by `(S(S(S(S(S 0)))))` in `(Eq.1)`, we get:

```
lt a (S(S(S(S(S 0))))) = fresh n in (S(S(S(S(S 0))))) == S n
                       & { a == 0 | fresh n' in a == S n' & lt n' n }
                                                                       (Eq.2)
```

Replace `(S(S(S(S(S 0)))))` `==` `S n` by `(S(S(S(S 0))))` `==` `n` in `(Eq.2)`, we get:

```
lt a (S(S(S(S(S 0))))) = fresh n in (S(S(S(S 0)))) == n
                       & { a == 0 | fresh n' in a == S n' & lt n' n }
                                                                       (Eq.3)
```

In `(Eq.3)`, remove `fresh n in (S(S(S(S 0))))` `==` `n`, then replace all free occurences of `n` by `(S(S(S(S 0))))`. The top level `&` and the braces are no longer needed, so also being removed. We get:

```
lt a (S(S(S(S(S 0))))) = a == 0
                       | fresh n' in a == S n'
                  & lt n' (S(S(S(S 0))))                                (Eq.4)
```

From `(Eq.1)` to `(Eq.4)` what we have done is to provide a concrete value (the Peano number 5) as the second argument of `lt` and use the result of unification to simplify the equation. The recursive call of `lt` in the right hand side of `(Eq.4)` can be treated similarly: we provide a concrete value (the Peano number 4) as the second argument of `lt` `(Eq.5)` and use the result of unification to simplify the equation `(Eq.6)`, which is then used to substitute for the recursive call of `lt` in `(Eq.4)`, as follows.

Replace `b` by `(S(S(S(S 0))))` and`a` by `n'` in `(Eq.1)` in a capture-avoiding manner, we get:

```
lt n' (S(S(S(S 0)))) = fresh n in (S(S(S(S 0)))) == S n
                     & { n' == 0 | fresh n'' in n' == S n'' & lt n'' n }
                                                                     (Eq.5)
```

Using the result of unification we can simplify `(Eq.5)` into:

```
lt n' (S(S(S(S 0)))) = n' == 0
                     | fresh n'' in n' == S n''
                     & lt n'' (S(S(S 0)))                        (Eq.6)
```

Now in `(Eq.4)` replace `lt n' (S(S(S(S 0))))` by the right hand side of `(Eq.6)`:

```
lt a (S(S(S(S(S O))))) = a == O
                       | fresh n' in a == S n'
                       & { n' == O
                         | fresh n'' in n' == S n''
                          & lt n'' (S(S(S O))) }          (Eq.7)
```

The right hand side of `(Eq.7)` produces another value of `a` which is `S O`, as follows. In `(Eq.7)`, distribute `a == S n'` we get:

```
lt a (S(S(S(S(S O))))) = a == O
                       | fresh n' in
                          a == S n' &  n' == O
                         | a == S n' & fresh n'' in n' == S n'' & lt n'' (S(S(S O)))

                                                    (Eq.8)
```

Replace `a == S n' & n' == O` by `a == S O` in `(Eq.8)`, we get:

```
lt a (S(S(S(S(S O))))) =  a == O
                       |  fresh n' in
                            a == S O
                          | a == S n' & fresh n'' in  n' == S n'' & lt n'' (S(S(S O)))

                                                                  (Eq.9)
```

In the right hand side of `(Eq.9)` move `a == S O` out of the scope of the `fresh n' in`, we have:

```
lt a (S(S(S(S(S O))))) =  a == O
                       |  a == S O
                       |  fresh n' in
                          a == S n' & fresh n'' in n' == S n'' & lt n'' (S(S(S O)))

                                                                  (Eq.10)
```

From `(Eq.1)` to `(Eq.10)` are steps of substitution, unification and simplification. Recursive calls are expanded and then reduced, and the initial formula `lt a (S(S(S(S(S O)))))` is gradually unfolded so that values of `a` are revealed one by one. Continue this way, the last but one equation would be:

```
lt a (S(S(S(S(S O))))) =  a == O
                       |  a == S O
                       |  a == S (S O)
                       |  a == S (S (S O))
                       |  a == S (S (S (S O)))
                       |  fresh n' in a == S n'
                       &  fresh n'' in n' == S n''
                       &  fresh n''' in n'' == S n'''
                       &  fresh n'''' in n''' == S n''''
                       &  fresh n''''' in n'''' == S n''''' & lt n''''' O
                                                                  (Eq.11)
```

Note that `lt n''''' O` expands to `fresh n in O == S n & ...` which is false, therefore the last equation is:

```
lt a (S(S(S(S(S O))))) =  a == O
                       |  a == S O
```

(continues on next page)

(continued from previous page)

```
                        |  a == S (S O)
                        |  a == S (S (S O))
                        |  a == S (S (S (S O)))                    (Eq.12)
```

From `(Eq.12)` we read off the answers to the query.

The derivation from `(Eq.1)` to `(Eq.12)`, combined with the operational semantics of OCanren in terms of stream manipulation, explains why we get the answer that `a` equals 0,1,2,3 or 4 from the goal `(G.1)`.

The reader may take an exercise to show that one plus one equals two by simplifying the formula `add (S O) (S O) c`.

1.4.7 Modifying the Search Behaviour

We compare two versions of the *simplify* relation, differing from each other only by a swap of conjuncts.

Both versions share the logic that the simplest form of `a/b` is `a'/b'` where `a'` (`b'`) is `a` (resp. `b`) divided by the greatest common divisor of `a` and `b`, provided `b` is non-zero. There is a short cut for the case where `a` is zero, then `b'` is set to one directly.

The difference is that:

- In one version we say, "*a* (*b*) divided by *c* equals *a'* (resp. *b'*), and *c* is the gcd of *a* and *b*."
- In the other version we say, "*c* is the gcd of *a* and *b*, and *a* (*b*) divided by *c* equals *a'* (resp. *b'*)."

In OCanren:

```
let simplify a b a' b' =
  ocanren {  fresh n in
    b == S n &
    { a == O & a' == O & b' == S O
    | fresh c, m in
        a == S m
      & div a c a' O             (* div first, then gcd *)
      & div b c b' O
      & gcd a b c } }

let simplify' a b a' b' =
  ocanren { fresh n in
     b == S n &
     { a == O & a' == O & b' == S O
     | fresh c, m in
         a == S m
       & gcd a b c                (* gcd first, then div *)
       & div a c a' O
       & div b c b' O  } }
```

The test file offers a comparison of these two versions over their forward and backward search behaviours. By *forward search* we mean that given a ratio *a/b* find its simplest form, e.g., 18/12 is simplified to 3/2. By *backward search* we mean given a ratio in the simplest form, find its equal ratios, e.g., 3/2 could be simplified from 6/4, 9/6, 12/8, etc. The test shows that both versions work well for forward search, but when it comes to backward search, `simplify` returns answers quickly but `simplify'` took ages without returning anything.

The ordering of the conjuncts, together with the state of the logic varaibles and the search behaviour of the sub-relations, results in apparently different operational meaning of the conjunctions in backward search, as follows:

1. *Variant 1*

```
div a c a' 0 &
div b c b' 0 &
gcd a b c
```

   Find *a* and *c* such that `a` divided by *c* equals *a'* exactly. Then find `b` such that `b` divided by *c* equals *b'* exactly. Now check that the gcd of *a* and *b* is *c*.

   Before the execution of the first conjunct, both *a* and *c* are unknowns. When the second conjunct is to be executed, *c* has already been found by the first conjunct, and only *b* is the unknown. Right before the execution of the thrid conjunct, all *a,b,c* have been found so only a check is due.

   This analysis requires knowledge of the search behaviour of *div arg1 arg2 arg3 arg4* in the following two cases:

   - Both *arg1, arg2* are unknowns, but *arg3, arg4* are known.
   - Only *arg1* is unknown, the other three are known.

1. *Variant 2*

```
gcd a b c &
div a c a' 0 &
div b c b' 0
```

   Find three unknowns`a,b,c` such that the relation *gcd a b c* holds, then check that *a* (*b*) is exactly dividable by *c* with quotient *a'* (resp. *b'*).

   Before the first conjunct is executed, all *a,b,c* are unknown, but by the time the second and third conjuncts are to be executed, the variables *a,b,c* are already computed by the first conjunct, therefore the last two conjuncts merely check the result.

   This analysis requires knowledge of the search behaviour of *gcd* when provided with three free logic variables for its three arguments.

The relevant search behaviours of the sub-relations mentioned in the table can be observed by running the test file or found in answers.txt. For instance, to know the search behaviour of `div` when only its first and second argument are unknown, we can make the specific query:

```
printf "\n What divided by what equals 3 with remainder 2 ? (give %d answers) \n\n" ans_
↪no;
ocrun2 ~n:ans_no (fun q r -> ocanren { div q r (S(S(S O))) (S(S O)) })
```

The answers are:

```
 What divided by what equals 3 with remainder 2 ? (give 20 answers)

(11, 3)
(14, 4)
(17, 5)
(20, 6)
(23, 7)
(26, 8)
(29, 9)
(32, 10)
(35, 11)
(38, 12)
```

(continues on next page)

(continued from previous page)

```
(41, 13)
(44, 14)
(47, 15)
(50, 16)
(53, 17)
(56, 18)
(59, 19)
(62, 20)
(65, 21)
(68, 22)
```

We could see that the `div` relation is enumerating all possible divisors in ascending order, starting with the least possible divisor which is 3 (the divisor must be greater than the remainder 2), together with the corresponding dividends.

In backward search, therefore, the `simplify` relation first finds a `c`-multiple of `a'` for some `c`, and then finds a `c` -multiple of `b'` for the same `c`. Its check of the gcd relation as the last step is starighforward if `a'/b'` is already in the simplest form. Note that the programmer provides `a'` and `b'` so practically `a'/b'` does not have to be in the simplest form, in which case the `gcd` check would fail. This all sounds like logical manners to find integral multiples of a ratio that is in the simplest form. However, the way in which `simplify'` approaches the problem is firstly guessing an arbitrary ratio together with the gcd of the numerator and the denominator, and then it checks if the ratio happens to reduce to `a'/b'`. This obviously has a bad chance to hit the target. That's why `simplify` works better than `simplify'` for backward search, and they only differ by a swap of conjuncts.

Note worthy is that the advantage of `simplify` over `simplify'` in backward search is at the cost of some efficiency in forward search, where `simplify'` smartly finds the gcd of the numerator and the denominator first and then divides to get the result, but `simplify` enumerates through all divisors of `a` to find the one that is also a divisor of `b` and the gcd of `a,b` — less efficient but still acceptable for small numbers.

As an exercise, the reader could experiment with reordering the conjuncts so that `gcd` is placed in between the two `div` 's. How would forward and backward search be influenced? A second question: what will happen and why, if we use `simplify` to find `a` and `b`, but give `a'` and `b'` as 4 and 2 respectively, i.e., a ratio not in the simplest form?

1.4.8 The Trick of Generate-and-test

When using the `gcd` relation to answer the question: "What and what have gcd 7 ?", the distribution of the answers does not look balanced: the second number is 7 most of the time, while the first number is growing. In comparison, the answers given by the `gcd'` relation has a more satisfactory distribution: the first number increases and for each possible first number, all possible second numbers are enumerated before the first number is further increased. The `gcd'` relation is defined using the famous technique known as *generate-and-test*, which we explain now.

Browsing the library source we could see that `gcd'` is defined in terms of `gcd` together with the addition relation and the Peano number predicate `isp` (read "is P" or "is a Peano Number").

Provided a free variable as the argument, `isp` enumerates all Peano numbers, in other words, we can obtain the following equation from the definition of `isp`, where the right hand side is an infinite formula:

```
isp n = n == O | n == S O | n == S (S O) | n == S (S (S O)) | ...
```

Therefore, `isp` could be a Peano number *generator* .

Moreover, when the third argument of `add` is concrete but the first and second argument are free variables, the `add` relation can find all ways to break up the third argument into two addends.

The sequence of `isp` and `add` in the body of `gcd'` can then be a Peano number pair generator, enumerating all possible pairs of Peano numbers (in the same way Georg Cantor shows that the set of rational numbers is enumerable). Now

the way `gcd'` works is clear: it enumerates through (i.e., *generates*) all possible pairs and then *tests* which pairs have the gcd 7. Since the pairs are generated systematically , the final answers are organized in the way we saw.

Another example of generate-and-test is the `simplify a b a' b'` relation. When `a,b` are given but `a',b'` are left unknown, the *first* ``div`` <peano.ml#L91>`__ generates all possible divisor-quotient pairs for `a`, and for each such pair the *second* ``div`` <peano.ml#L92>`__ tests if the divisor also divides `b` and if so generates the quotient. The sequence of two `div` 's then plays the role of a generator of all common divisors of `a,b` together with the corresponding pairs of numbers which are `a,b` divided by their common divisors. The `gcd` sub-relation peano.ml#L93 finally checks for the greatest common divisor, and the corresponding pair of quotients is the answer for `a',b'`.

In summary, generate-and-test is both a technique that the programmer applies to solve certain problems (e.g., the `gcd'` case), and a usual way in which relational programs search for answers even if the programmer does not intentionally apply it (e.g., the `simplify` case).

1.4.9 The Formula Parser

In the library implementation and the test file, we often see formulae enclosed by the `ocanren{}` quotation which takes care of, among others, precedence and associativity of the logic connectives. It is an OCanren-specific syntax extension which is described *OCanren-specific Camlp5 syntax extension*

1.4.10 Building a Library

Being essentially an OCaml library, an OCanren library shall have its interface `.mli` and implementation `.ml`. The interface typically contains type definitions (at the four levels), auxiliaries (such as injection functions, reifiers, etc.) and the relations. Note also the `@type` syntax in the interface.

Relations shall be systematically tested. This means all possible combinations of unknown arguments shall be queried wrt. each relation. The benefits of such systematic test includes reducing surprises when running the programs, and helping with debugging “bigger” relations that are defined using smaller relations. For example, understanding the search behaviour of `simplify` requires knowledge of search behaviours of `div` and `gcd` and in turn those of `lt`, `lte` and `add`.

1.5 The Range of a Fleet of Aircraft

List of symbols

- B : *abstract capacity*
- C : *fuel capacity*
- D : *distance* (between target and airbase)
- N : *fleet size*
- R : *fuel efficiency*

1.5.1 Background

There is a fleet of N identical aircraft, where every aircraft has a fuel capacity of C liters and fuel efficiency of R miles per liter of fuel. The fleet has a mission of reaching some target located at a distance D from the airbase, where CR < D < NCR. Once taking off, there is no more airbase along the way for the fleet to land and refuel. The fleet adopts such a strategy that at any stage, any one aircraft could be abandoned, whose fuel is simultaneously transferred to some fellow aircraft. The mission is considered as successful if at least one aircraft finally reaches the target. The typical behaviour of the fleet is like (read bottom-up):

```
            X   : Reaching the target
            ^
            |   : Action 1
          X X   : Action 2
          ^ ^
          | |   : Action 1
        X X X   : Action 2
        ^ ^ ^
        | | |   : Action 1
      X X X X   : Action 2
      ^ ^ ^ ^
      | | | |   : Action 1
    X X X X X   : Action 2
    ^ ^ ^ ^ ^
    | | | | |   : Action 1
  X X X X X X   : Action 2
  ^ ^ ^ ^ ^ ^
  | | | | | |   : Action 1
  X X X X X X   : Taking off from the airbase (initially six aircraft in the fleet).
-------------------------------------------------------------------------------------
Action 2 : Abandoning one aircraft, whose fuel is shared by the rest of the fleet.
Action 1 : Flying forward.
Symbol X : An aircraft
```

There are two problems of interest: 1. Given a fleet travel plan (which specifies how far the fleet shall fly together after taking off or after some aircraft is abondoned, and how to distribute the fuel of the abandoned aircraft among the rest of the fleet), how far can the fleet fly ? 1. Given a target distance, what shall the travel plan be for the fleet to reach the target, if possible?

The first problem is relatively easy. For instance, if we know that the plan is to let the fleet fly forward until they all run out of fuel, then they can fly as far as a single aircraft. We can even write a computer program to do the calculation for us, given the size of the fleet, the fuel capacity and efficiency of the aircraft model, and the travel plan.

However, the second problem is less straightforward, and it may require some trial-and-error and creativity to solve. We might do something like:

“Oh, I got a plan, let’s try it !”

“No it won’t work.”

“Then … what about this modified plan?”

“Sorry, this won’t work either !”

“Ok, let’s see … this one?”

” … “

The gift of OCanren is that we can use it to write a program to solve the first problem, but the same piece of program can also solve the second problem for us without any modification. It works likes this: we define a relation: *steps(pre,*

acts, post) where *pre* is a state of the fleet, *acts* is a travel plan, and *post* is the state of the fleet after executing the travel plan. To solve the first problem, we pose the query: *steps(initial_state, travel_plan, where?)* and to solve to the second problem we pose the query: *steps(initial_state, what_plan?, desired_state)*.

1.5.2 The Design of the Program

We adopt the following mathematical abstractions.

Discrete motion

For some arbitrary positive interger B, we take C/B liters as one unit of fuel and take (RC)/B miles as one unit of distance, and say that an aircraft has a maximum capacity of B units of fuel, and with which it can fly for B units of distance at the most. We call B the *abstract capacity* of an aircraft.

Moreover, we assume that an aircraft consumes fuel and covers distance in a discrete (or unit-by-unit) and propotional manner, where one unit of fuel is consumed at a time, resulting in one unit of distance flied.

Discrete fuel trasnfer

Transfer of fuel within the fleet is also discrete: only whole units are allowed. For example, if an aircraft has 3 units of fuel left in the tank that has a capacity of 5 units, then it can only refuel for 1 unit or 2 units.

Picking an abstract capacity

Although the value of the abstract capacity B is arbitrarily picked, we must set it to at least 2. If we set B = 1 then it would be impossible for the fleet to reach the target (now B < D < NB), given our *discrete motion* and *discrete fuel transfer* assumptions.

For instance, B = 1 implies that the fleet would move forward for 1 unit of distance, running out of fuel and all aircraft abandoned. If the fleet has two aircraft, and B = 2, then there are two possibilities:

1. The fleet flies for 2 units of distance, and then run out of fuel before reaching the target;
2. The fleet flies for 1 unit, then one aircraft is abandoned, transferring the fuel (1 unit) to the other, who then continues to fly for 2 units. Thus the fleet achieves the range of 3 units.

States of the fleet

A state of the fleet consists of the position of the fleet and a list of the amount of fuel available for each aircraft in the fleet, called the *fuel profile*. Implicitly the fuel profile shows how many aircraft are currently in the fleet: this is the length of the list.

Example. From the fuel profile `[5;5;5]` we can read that there are three aircraft in the fleet and each has five units of fuel. If this fleet fly together for 3 units of distance, the fuel profile would become `[2;2;2]`. Now if we abondon one aircraft (any one is ok, for the result is the same), and give its fuel to the rest of the fleet, the possible fuel profiles after the abandoning would be: `[2;4]`, `[3;3]` or `[4;2]`.

Fleet actions

There are two possible actions of the fleet: flying forward (together), and abandoning (with fuel sharing at the same time). A travel plan should be a list of actions as informative as possible, for example, showing how far the fleet flies, and how the fuel is shared. Therefore we define two action labels: `Forward(n)` and `Abandon([n1;...;nk])`. A initial state and a list of action labels would allow us to compute the state when the actions have been executed.

Example. Let `(0, [5;5])` be the initial state, and `[Forward (2); Abandon ([5]); Forward (5)]` be a list of actions. We could read that initially the fleet has two aircraft and is at position 0. They would fly forward for 2 units of distance, then the state would be (we are calculating by hand now) `(2, [3;3])`. The next action is `Abandon([5])`, which means that one aircraft is abandoned, and the new fuel profile of the fleet is the parameter of the `Abandon` label: `[5]`. The change of the fleet fuel profile from `[3;3]` (before abandoning an aircraft) to `[5]` (after abandoning the aircraft) implies that 2 units of fuel from the abandoned aircraft has been transferred to the remaining aircraft. Now the state is `(2, [5])`: we assume that abondoning and fuel transfer happen simultaneously and take no time. The final action is `Forward(5)` meaning the singleton fleet would fly forward for 5 units of distance. The last state is `(7, [0])`: there is one aircraft remaining in the fleet; it is 7 units of distance away from the airbase and it has no fuel. We have given an example of computing the final state given an initial state and a list of actions. It is interesting to note that the range (or maximum reach) of the two-aircraft fleet (each aircraft has an abstract capacity of 5 units) is 7 units.

Fleet state transition rules

A state transition rule relates a pre-state, a single action and the post-state. We also chain the state transition rules to obtain a multi-step state transition rule which relates a pre-state, a list of actions and the post-state. To avoid unnecessarily verbose travel plan, we require that a forward action must not be followed immediately by another forward action: if so, why not combine them into one? Subject to reasonable alternatives, we also require that after abandoning one aircraft, the fleet shall move forward before abandoning another. In the example above we have actually executed the multi-step state transition rule by hand.

1.5.3 OCanren at Work

So far we have discussed about the design of an algorithm to compute the post-state from a pre-state and a list of actions. We indicated that this algorithm, written in OCanren, can be run "backward": given a pre-state and a post-state, find the list of actions that bridges them. Below are some such results.

Let B = 5 and OCanren suggested the following solutions for fleets of various sizes to achieve certain ranges.It took about 10 mins to compute for the 6-aircraft fleet.

Fleet Size	Range	Moves
2	7	[Forward (2); Abandon ([5]); Forward (5)]
3	9	[Forward (2); Abandon ([4; 5]); Forward (2); Abandon ([5]); Forward (5)]
4	10	[Forward (2); Abandon ([5; 4; 3]); Forward (1); Abandon ([4; 5]); Forward (2); Abandon ([5]); Forward (5)]
5	11	[Forward (1); Abandon ([5; 5; 5; 4]); Forward (1); Abandon ([5; 5; 5]); Forward (2); Abandon ([4; 5]); Forward (2); Abandon ([5]); Forward (5)]
6	12	[Forward (1); Abandon ([5; 5; 5; 5; 4]); Forward (1); Abandon ([5; 5; 5; 4]); Forward (1); Abandon ([5; 5; 5]); Forward (2); Abandon ([4; 5]); Forward (2); Abandon ([5]); Forward (5)]

1.5.4 Reference

J. N. Franklin 1960 The Range of a Fleet of Aircraft Journal of the Society for Industrial and Applied Mathematics, 8(3), 541–548. (8 pages)

We begin with studying a “hello world” program, exposing various aspects of the technicalities of the language. After that, we go deep down into one of the most important: understanding and defining user types. Afterwards we will be ready to write non-trivial relational programs. we define a simple relational data base, followed by a library for Peano numbers.

In the last lesson we use OCanren to solve an arithmetic puzzle. The last three lessons are so ordered that the complexity of the examples increases gently.

CHAPTER

TWO

INSTALLATION

OCanren can be installed using opam (>= 2.1). First, install opam itself, initialize it and install right compiler version.

- `opam init --bare` if your opam has not been initialized before
- `opam switch create 4.14.0 --packages=ocaml-variants.4.14.0+options, ocaml-option-flambda` installes right compiler. (4.14.0 is a custom switch identifier)
- `eval $(opam env)` updates an environment. Opam should write invocation of this command to your `/.bashrc`
- check that `ocamlc -v` prints right compiler version

Then, install dependencies and `OCanren`:

- **Install GT (pick one command)**
  - `opam install GT -y` to install GT from opam repository.
  - `opam pin add GT https://github.com/JetBrains-Research/GT.git -n -y` for fresh GT.
- `git clone https://github.com/JetBrains-Research/OCanren.git`
- `cd OCanren`
- `opam install . --deps-only --yes`
- `make`
- `make tests`

Expected workflow:

- add new test to try something out,
- or use our template repository.

CHAPTER

THREE

CAMLP5 SYNTAX EXTENSIONS

A few syntax extensions are used in this project.

For testing we use the one from `logger-p5` opam package. It allows to convert OCaml expression to its string form. For example, it rewrites `let _ = REPR(1+2)` to

```
$ camlp5o `ocamlfind query logger`/pa_log.cmo pr_o.cmo a.ml
let _ = "1 + 2", 1 + 2
```

For OCanren itself we use syntax extension to simplify writing relational programs

```
$  cat a.ml
let _ = fresh (x) z
$  camlp5o _build/camlp5/pa_ocanren.cmo pr_o.cmo a.ml
let _ = OCanren.Fresh.one (fun x -> delay (fun () -> z))
```

3.1 OCanren vs. miniKanren

The correspondence between original miniKanren and OCanren constructs is shown below:

miniKanren	OCanren
`#u`	success
`#f`	failure
`((==) a b)`	`(a === b)`
`((=/=) a b)`	`(a =/= b)`
`(conde (a b ...) (c d ...) ...)`	`conde [a &&& b &&& ...; c &&& d &&& ...; ...]`
`(fresh (x y ...) a b ... )`	`fresh (x y ...) a b ...`

In addition, OCanren introduces explicit disjunction (`|||`) and conjunction (`&&&`) operators for goals.

3.2 OCanren-specific Camlp5 syntax extension

(Also known as regular extension)

This section is a part of Yue Li's tutorial on OCanren.

We take a look at the implementation of the `ocanren{}` quotation which we will call *the formula parser* in the rest of this lesson. Our terminology follows Camlp5 Reference Manual. We take a top-down approach, starting with an overview of the structure of the parser, then explain its individual parts.

3.2.1 The structure of the parser: an overview

We describe the formula parser as the reader (a Camlp5 novice) sees it, and then putting it in perspective, briefly explain how it works.

What we see

This code links with Camlp5 libraries, in particular the one that provide ways to extend syntax: *pa_extend*. Loading them amounts to extending the OCaml syntactic category expression with several sub-categories one of which is named *extend*:

```
expr = ... | extend ;
extend = "EXTEND", extend-body, "END" ;
```

An expression that belongs to the category "extend" would be called an *EXTEND statement*.

Our formula parser has only one EXTEND statement, whose extend-body starts with a global indicator followed by a semicolon separated list of *entries* (whose names are, exhaustively, long_ident, expr, ocanren_embedding, ocanren_expr, ocanren_term, ocanren_term' and ctyp). An *entry* is a (vertical bar separated) list of *levels* (with a pair of enclosing square brackets); a *level* is a (vertical bar separated) list of *rules* (with a pair of enclosing square brackets); a (non-empty) *rule* is a (semicolon separated) list of "psymbols" (collectively acting as a pattern) followed by an optional semantic action that produces an abstract syntax tree (or AST, of any string that matches the pattern specified by the list of psymbols). The details on the syntax and semantics of the "extend" category can be found in the Extensible Grammars section of the Camlp5 Manual.

Besides the EXTEND statement our formula parser has some auxiliary functions such as decapitalize, ctor and fix_term etc.

How it works

The syntax extension kit `pa_extend` is fundamental for the cascade of extensions described below. The entries `expr` and `ctyp` origin from the module Pcaml that is the core of Camlp5 and is opened by the formula parser. Pcaml initializes the (empty) grammar entries `expr` and `ctyp`. The standard OCaml parsing kit of Camlp5 then defines them by means of an EXTEND statement and accordng to the standard syntax of OCaml. Our EXTEND statement further extends these global entries with locally defined entries — entries other than `expr` and `ctyp` in our EXTEND statement are locally defined, such as `ocanren_embedding`, `ocanren_expr` and `ocanren_term` etc. Below you can find three stages of extension with links to documentation .. The following table summarizes the stages of extension, providing links to copies of relevant files from either OCanren ource or Camlp5 source, together with their documentations.

1. Initialization is documented in The Pcaml module and happend in file *pcaml.ml* : expr and ctyp.
2. Parsing Kit for Standard OCaml *pa_o.ml* : expr, ctyp
3. OCanren Formula Parser *pa_ocanren.ml*: expr, ctyp

As a preprocessing tool, Camlp5 defines its own parser `pa_o.ml` for standard OCaml, so that any standard OCaml code can be converted by it into an AST recongnizable by the OCaml compiler. Is `pa_o.ml` a redundant piece of work for we can just use the OCaml compiler to build the AST? Not exactly, because besides `pa_o.ml`, Camlp5 also provides EXTEND statments so that syntactic categories defined in `pa_o.ml` can be extended. The result is that using the combination of `pa_ocanren.ml` and `pa_o.ml` we can convert code that is not wholly in OCaml into a purely OCaml AST.

Conclusion

The OCanren formula parser has the EXTEND statement as its core, which consists of a list of entries, notably the global entries `expr` and `ctyp` that extend the corresponding predefined entries that conform to standard OCaml. Such extension is characterized by the locally defined entries such as `ocanren_embedding`.

We will next focus on the extension of `expr` and leave `ctyp`. As far as the semantics is concerned entries are parsers for syntactic categories. From now on we use the words "entry" and "parser" interchangeably.

3.2.2 The global entry: `expr`

This is the major entry of the OCanren formula parser, which starts like:

```
expr: LEVEL "expr1" [ ...
```

where we see the entry name *expr* and the position `LEVEL "expr1"`. We now use OCanren-expr to refer to the `expr` entry in the OCanren formula parser, and OCaml-expr to refer to the predefined entry `expr` in the Camlp5 parsing kit for standard OCaml. OCanren-expr extends OCaml-expr in the position `LEVEL "expr1"` : the first level of the OCanren-expr is merged with the level named "expr1" of the OCaml-expr, i.e., their rules are put together and grouped as a single level named "expr1"; other levels from OCanren-expr are inserted into OCaml-expr as new levels, right below the extended "expr1" level. There are three levels in the OCanren-expr, the third of which is:

```
[ e=ocanren_embedding -> e ]
```

This third level of OCanren-expr is inserted as a new level in OCaml-expr, and the entry `ocanren_embedding` directly corresponds to the `ocanren{}` quotations we see in the library implementation, so that we can mix `ocanren{}` quotations with standard OCaml expressions, and Camlp5 will take care to convert such mixture into standard OCaml AST. We now explain the local entry `ocanren_embedding`.

3.2.3 Local entries I: `ocanren_embedding` and `ocanren_expr`

The entry `ocanren_embedding` directly corresponds to the `ocanren{}` quotations we see in the library implementation, and it further calls the entry `ocanren_expr` to parse the content between the braces:

```
ocanren_embedding: [[ "ocanren"; "{"; e=ocanren_expr; "}" -> e ]];
```

The `ocanren_expr` entry has four levels which strongly reminds us of the recursive definition of a formula, i.e, a formula is either atomic, or a conjunction/ disjunction of two formulae, or an existential quantification over a formula, or an explicitly delimited formula (with braces).

1. The first level parses a disjunction:

```
"top" RIGHTA [ l=SELF; "|"; r=SELF -> <:expr< OCanren.disj $l$ $r$ >> ]
```

2. The second level parses a conjunction:

```
RIGHTA [ l=SELF; "&"; r=SELF -> <:expr< OCanren.conj $l$ $r$ >> ]
```

3. The third level parses a fresh variable introduction (i.e., existential quantification):

```
[ "fresh"; vars=LIST1 LIDENT SEP ","; "in"; b=ocanren_expr LEVEL "top" ->
   List.fold_right
     (fun x b ->
        let p = <:patt< $lid:x$ >> in
        <:expr< OCanren.call_fresh ( fun $p$ -> $b$ ) >>
     )
     vars
     b
]
```

4. The fourth level parses atomic, named and grouped formulae (and else):

```
"primary" [
    p=prefix; t=ocanren_term                        -> let p = <:expr< $lid:p$ >> in <:expr
↪< $p$ $t$ >>
  | l=ocanren_term; "==" ; r=ocanren_term           -> <:expr< OCanren.unify $l$ $r$ >>
  | l=ocanren_term; "=/="; r=ocanren_term           -> <:expr< OCanren.diseq $l$ $r$ >>
  | l=ocanren_term; op=operator; r=ocanren_term     -> let p = <:expr< $lid:op$ >> in
                                                       let a = <:expr< $p$ $l$ >> in
                                                       <:expr< $a$ $r$ >>
  | x=ocanren_term                                  -> x
  | "{"; e=ocanren_expr; "}"                        -> e
  (* other rules omitted *)
  ...
]
```

The order of the levels determines the precedence of the logic connectives: the parser first sees if the formula is a disjunction at the top level, if not, sees if it is conjunction, and so on, implying that disjunction has the least precedence, above which is conjunction, then existential quantification, and finally syntactic equality, disequality and braced groups (among others) enjoy the highest precedence. We can justly call a level: a "precedence level".

The first and second level also have the associativity indicator `RIGHTA`, requiring that the conjunction and disjunction connectives associate to the right.

The third level refers back to the first level (named "top") when parsing the <body> part of a formula of the form `fresh` <vars> `in` <body>, implying that the scope of `fresh` extends to the right as far as possible.

3.2.4 Quotations and antiquotations

In every rule above we could see at least one quotation:

```
quotation = "<:", quotation name, "<", quotation body, ">>"
```

Within a quotation body we may see an antiquotation:

```
antiquotation = "$", antiquotation body, "$"
```

If antiquotations are not allowed, then a quotation body is any expression in the revised syntax of OCaml. At parse time a quotation is expanded by the (loaded and predefined) quotation expander `q_MLast.cmo` into an AST of the quotation body. An antiquotaion body is usually a pattern variable bound to some other AST which is inserted into the the quotation body's AST.

3.2.5 Local entries II: ocanren_term and ocanren_term'

The values that we write in an ocanren{} quotation, such as "this is a string", 'c' (a single character), true (a boolean value), S (S O) (a constructor application), (O, S O) (a tuple), 15 (an integer), [1;2;3] (a list) and false :: [] (amending a list) etc., are converted into the injected level from the ground level where they seem to be. For example, the occurrence of S (S O) in the expression below is transformed into s (s (o ())) :

```
ocanren { fresh x in S (S O) == x }
```

Such conversion bridges the gap between the programmer's intuition of writing OCaml values and OCanren's internal representation of the same values, Inspecting the entries ocanren_term, ocanren_term' and their auxiliary functions help us know precisely how the conversion is performed.

Below is the definition of the entry ocanren_term :

```
ocanren_term: [[ t=ocanren_term' -> fix_term t ]];
```

where the ocanren_term' parser is called immediately to process expressions like S (S O) and the intermediate result (an AST) is bound to the pattern variable t and then passed to the auxiliary function fix_term. The AST returned by fix_term is returned by the parser ocanren_term.

The ocanren_term' parser has four levels, namely:

1. "app", for applications.

```
"app"  LEFTA  [ l=SELF; r=SELF -> <:expr< $l$ $r$ >> ]``
```

Applications are treated as being left associative as indicated by LEFTA. This level not yet converts constructor applications into injection function applications. Instead it simply builds the AST of the application in a straightforward manner, not distinguishing a constructor application from a function application.

2. "list" , for non-empty lists with :: as the top level constructor.

```
"list" RIGHTA [ l=SELF; "::"; r=SELF -> <:expr< OCanren.Std.List.cons $l$ $r$ >> ]
```

The constructor :: is replaced by the OCanren standard library function cons which is the injection function for the constructor OCanren.Std.List.Cons.

3. "primary", which has rules for:

   - anti-quotations

   ```
   "!"; "("; e=expr; ")" -> e
   ```

   So that the ocanren{} quotation would take any <value> from !(<value>) as is without further processing. In other words, the <value> will be parsed using the entry expr.

   - integers

   ```
   c=INT -> let n = <:expr< $int:c$ >> in <:expr< OCanren.Std.nat $n$ >>
   ```

   Thus, occurrences of integers like 15 within the ocanren{} quotation would be converted to values of the Peano number type that is provided by the OCanren standard library OCanren.Std.Nat.

   - characters and strings

   ```
   c=CHAR   -> let s = <:expr< $chr:c$ >> in <:expr< OCanren.inj (OCanren.lift $s
   ↪$) >>      | s=STRING -> let s = <:expr< $str:s$ >> in <:expr< OCanren.inj␣
   ↪(OCanren.lift $s$) >>
   ```

Characters and strings are injected using the primary injection function `!!` (see its signature and implementation).

- booleans

```
| true"   -> <:expr< OCanren.Std.Bool.truo >>
| "false"  -> <:expr< OCanren.Std.Bool.falso >>
```

Boolean values are converted into the corresponding injected values from the OCanren standard library LBool.

- lists delimited by `[]` and `;` <https://github.com/JetBrains-Research/OCanren/blob/master/camlp5/pa_ocanren.ml#L273>`__

```
"["; ts=LIST0 ocanren_term' SEP ";"; "]" ->
  ( match ts with
  | [] -> <:expr< OCanren.Std.nil () >>
  | _  ->
      List.fold_right (fun x l -> <:expr< OCanren.Std.List.cons $x$ $l$ >> )
      ts
      <:expr< OCanren.Std.nil () >>
  )
```

The entry *ocanren_term'* is recursively called to process the list members and the injection functions for list constructors are applied.

- operators (which are not qualified)

```
"("; op=operator_rparen -> <:expr< $lid:op$ >>
```

Operators are specified by the auxiliary function `` `is_operator <.https://github.com/JetBrains-Research/OCanren/blob/master/camlp5/pa_ocanren.ml#L87>`__ and extracted by another auxiliary function operator_rparen (the name of which reads "operator right parenthesis").

- (possibly empty) tuples

```
"("; ts=LIST0 ocanren_term' SEP ","; ")" ->
(match ts with
  | []  -> <:expr< OCanren.inj (OCanren.lift ()) >>
  | [t] -> t
  | _   -> <:expr< ( $list:ts$ ) >> )
```

There is a recursive call of the entry itself to process members of the tuple, and then the AST of the tuple is built.

1. The level for long identifiers.

```
[ long_ident ]
```

This level calls the entry long_ident to build AST's of (possibly qualified) upper / lower case identifiers and operators which are taken as is.

Therefore, given `S (S O)` the `ocanren_term'` parser would return a straightforward translation into an AST. The interesting thing is done by `fix_term` and its helper `ctor` (read "C-tour"). The latter tests the input: if it is a (possibly qualified) uppercase identifier then sets the initial letter to lowercase and wraps the whole thing by `Some`, e.g., `Mod1.Mod2.ABC` becomes (roughly) `Some Mod1.Mod2.aBC` ; if the input is not a (qualified) uppercase identifier then returns `None` :

```
let rec ctor e =
  let loc = MLast.loc_of_expr e in
  match e with
  | <:expr< $uid:u$ >>    -> Some (<:expr< $lid:decapitalize u$ >>)
  | <:expr< $m$ . $e$ >> -> (match ctor e with Some e -> Some (<:expr< $m$ . $e$ >>) | _␣
↪-> None)
  | _                    -> None
```

The fix_term function then recurses down the structure of applications to systematically replace uppercase identifiers with lowercase identifiers produced by `ctor`. After a constant constructor is changed to lowercase, it is provided with the unit value `()` as the argument, e.g., `0` becomes `o ()`. A non-constant constructor is not only set to lowercase, but also has its argument list transformed, e.g., `Cons(a,b)` becomes (roughly) `cons a b`. Tuples are also replaced by their OCanren standard library counterpart — logic tuples.

These lowercase identifiers converted from constructors are supposed to be injection functions, which must be defined by the programmer somewhere in the program, otherwise there would be some compile-time error like "unbound identifier". This explains why the injection function names are always differ from the corresponding constructor names by one letter: the initial letter.

CHAPTER

FOUR

SYNTAX EXTENSIONS

4.1 Writing OCanren without syntax extensions

OCanren and original miniKanren consist of many syntax extension. There we describe how to write relation in OCanren without them, to make values of the obvious.

This is how we write relation without syntax extensions. The top of the input is techinal stuff for loading right packages into OCaml toplevel.

```
$ ocaml -stdin  -impl - <<EOF
#use "topfind";;
#require "OCanren";;
#rectypes;;
open OCanren
open OCanren.Std

let rec appendo x y xy =
   conde [
      (x === Std.nil()) &&& (y === xy);
      Fresh.three (fun h tl tmp -> (x === h % tl) &&& (appendo tl y tmp) &&& (xy === h %␣
↪tmp))
   ]
EOF
```

This is how we use PPX exntesion to simplify code. It allows us

- not to think about count of fresh varaibles
- automatically insert &&& when creating fresh variables

```
$ ocaml -stdin  -impl - <<EOF                                                    ␣
↪                    4.13.1+flambda
#use "topfind";;
#require "OCanren";;
#rectypes;;
open OCanren
open OCanren.Std

#require "OCanren-ppx.ppx_fresh";;
let rec appendo x y xy =
      conde [
         fresh () (x === Std.nil()) (y === xy);
```

(continues on next page)

(continued from previous page)

```
        fresh (h tl tmp) (x === h % tl) (appendo tl y tmp) (xy === h % tmp)
    ]
EOF
```

There is also a camlp5 extension for simplifing relations described here.

4.2 PPX syntax extensions

PPX syntax extensions are not related to camlp5 and should be used, for example, if you want decent IDE support. Main extensions are compilable by `make ppx`

4.2.1 *ppx_repr*

An analogue for logger library is called `ppx_repr` (located at *OCanren-ppx.ppx_repr* package):

```
$ cat regression_ppx/test002repr.ml
let _ = REPR(1+2)
$ ./pp_repr.native regression_ppx/test002repr.ml
let _ = ("1 + 2", (1 + 2))
$ ./pp_repr.native -print-transformations
repr
```

4.2.2 *ppx_fresh*

An OCanren-specific syntax extension extension for creating fresh variables. It provides canonical miniKanren syntax from the Scheme

```
$ echo 'let _ = fresh (x) z' | dune exec ppx/pp_fresh.exe --  -impl -
let _ = Fresh.one (fun x -> delay (fun () -> z))
$ dune exec ppx/pp_fresh.exe --  -print-transformations
pa_minikanren
```

4.2.3 *ppx_distrib*

This extension is used to generate smart constructors and reifier from definition of our type. It optionally allows to decorate type definitions with deriving attributes which could be expanded later.

Below we use extension point with two type definitions. First one is nonrecursive fully abstract type. The extension with generate monadic fmap called *fmapt* for it. The second one is a specialization of previous type definition for our needs. it is uses to generate types for *ground*, *logic*, and *injected* values; reifier *reify* and exceptionful projection *prj_exn* from *injected* to *logic*/*ground* values; and smart constructor for creating values of *injected* type.

```
 dune exec ppx/pp_distrib.exe --  -impl - <<EOF
heredoc>[%%distrib
type nonrec 'a t =
   | Z
   | S of 'a
[@@deriving gt ~options:{ gmap; show }]
```

(continues on next page)

(continued from previous page)

```
type ground = ground t]
heredoc> EOF
include
   struct
      type nonrec 'a t =
         | Z
         | S of 'a [@@deriving gt ~options:{ gmap; show }]
      type ground = ground t[@@deriving gt ~options:{ gmap; show }]
      type logic = logic t OCanren.logic[@@deriving gt ~options:{ gmap; show }]
      type injected = injected t OCanren.ilogic
      let fmapt a subj__002_ =
         let open Env.Monad in
         ((Env.Monad.return (GT.gmap t)) <*> a) <*> subj__002_
      let (prj_exn : (_, ground t) Reifier.t) =
         let open Env.Monad in
         let open Env.Monad.Syntax in
            Reifier.fix (fun self -> OCanren.prj_exn <..> (chain (fmapt self)))
      let (reify : (_, logic t OCanren.logic) Reifier.t) =
         let open Env.Monad in
         let open Env.Monad.Syntax in
            Reifier.fix
               (fun self ->
                  OCanren.reify <..>
                  (chain (Reifier.zed (Reifier.rework ~fv:(fmapt self)))))
      let z () = OCanren.inji Z
      let s _x__001_ = OCanren.inji (S _x__001_)
   end
```

4.2.4 *ppx_deriving_reify*

Simplifies inline generation of reifiers for already known types.

```
$ dune exec ppx/pp_deriving_reify.exe --  -impl - <<EOF
let _ = [%reify: GT.int GT.list]
EOF
let _ = Std.List.reify OCanren.reify
```

CHAPTER

FIVE

PAPERS

- Typed Embedding of Relational Programming Language
- Relational Synthesis for Pattern Matching
- TODO: noCanren
- others…

OCanren is a strongly-typed embedding of miniKanren relational programming language into OCaml. Nowadays, implementation of OCanren strongly reminds faster-miniKanren. Previous implementation was based on microKanren with disequality constraints.

CHAPTER

SIX

OCANREN VS. MINIKANREN

The syntax between OCanren and vanilla miniKanren is a little bit different: *OCanren vs. miniKanren*. The correspondence between original miniKanren and OCanren constructs is shown below:

CHAPTER

SEVEN

INJECTING AND PROJECTING USER-TYPE DATA

To make it possible to work with OCanren, user-type data have to be *injected* into logic domain. In the simplest case (non-parametric, non-recursive) the function

```
val inj  : ('a, 'b) injected -> ('a, 'b logic) injected
val lift : 'a ->  ('a, 'a) injected
```

can be used for this purpose:

```
inj @@ lift 1
```

```
inj @@ lift true
```

```
inj @@ lift "abc"
```

If the type is a (possibly recursive) algebraic type definition, then, as a rule, it has to be abstracted from itself, and then we can write smart constructor for constructing injected values,

```
type tree = Leaf | Node of tree * tree
```

is converted into

```
module T = struct
  type 'self tree = Leaf | Node of 'self * 'self

  let fmap f = function
  | Leaf -> Leaf
  | Node (l, r) -> Node (f l, f r)
end
include T
module F =  Fmap2(T)
include F

let leaf    ()  = inj @@ distrib @@ T.Leaf
let node   b c  = inj @@ distrib @@ T.Node (b,c)
```

Using fully abstract type we can construct type of `ground` (without logic values) trees and type of `logic trees` – the trees that can contain logic variables inside.

Using this fully abstract type and a few OCanren builtins we can construct `reification` procedure which translates `('a, 'b) injected` into it's right counterpart.

```
type gtree = gtree T.t
type ltree = ltree X.t logic
type ftree = (rtree, ltree) injected
```

Using another function `reify` provided by the functor application we can translate `(_, 'b) injected` values to `'b` type.

```
val reify_tree : ftree -> ltree
let rec reify_tree eta = F.reify LNat.reify reify_tree eta
```

And using this function we can run query and get lazy stream of reified logic answers

```
let _: Tree.ltree OCanren.Stream.t =
  run q (fun q  -> q === leaf ())
        (fun qs -> qs#reify Tree.reify_tree)
```

CHAPTER

EIGHT

BOOL, NAT, LIST

There is some built-in support for a few basic types — booleans, natural numbers in Peano form, logical lists. See corresponding modules.

The following table summarizes the correspondence between some expressions on regular lists and their OCanren counterparts:

Regular lists	OCanren
`[]`	`nil`
`[x]`	`!< x`
`[x; y]`	`x %< y`
`[x; y; z]`	`x % (y %< z)`
`x::y::z::tl`	`x % (y % (z % tl))`
`x::xs`	`x % xs`

CHAPTER

NINE

SYNTAX EXTENSIONS

There are two constructs, implemented as syntax extensions: `fresh` and `defer`. The latter is used to eta-expand enclosed goal ("inverse-eta delay").

However, neither of them actually needed. Instead of `defer (g)` manual expansion can be used:

```
delay (fun () -> g)
```

To get rid of `fresh` one can use `Fresh` module, which introduces variadic function support by means of a few predefined numerals and a successor function. For example, instead of

```
fresh (x y z) g
```

one can write

```
Fresh.three (fun x y z -> g)
```

or even

```
(Fresh.succ Fresh.two) (fun x y z -> g)
```

CHAPTER

TEN

RUN

The top-level primitive in OCanren is `run`, which can be used in the following pattern:

```
run n (fun q1 q2 ... qn -> g) (fun a1 a2 ... an -> h)
```

Here `n` stands for *numeral* (some value, describing the number of arguments, `q1`, `q2`, ..., `qn` — free logic variables, `a1`, `a2`, ..., `an` — streams of answers for `q1`, `q2`, ..., `qn` respectively, `g` — some goal, `h` — a *handler* (some piece of code, presumable making use of `a1`, `a2`, ..., `an`).

There are a few predefined numerals (`q`, `qr`, `qrs`, `qrst` etc.) and a successor function, `succ`, which can be used to "manufacture" greater numerals from smaller ones.

CHAPTER

ELEVEN

SAMPLE

We consider here a complete example of OCanren specification (relational binary search tree):

```
open Printf
open GT
open OCanren
open OCanren.Std

module Tree = struct
  module X = struct
    (* Abstracted type for the tree *)
    @type ('a, 'self) t = Leaf | Node of 'a * 'self * 'self with gmap,show;;
    let fmap eta = GT.gmap t eta
  end
  include X
  include Fmap2(X)

  @type inttree = (int, inttree) X.t with show
  (* A shortcut for "ground" tree we're going to work with in "functional" code *)
  @type rtree = (LNat.ground, rtree) X.t with show

  (* Logic counterpart *)
  @type ltree = (LNat.logic, ltree) X.t logic with show

  type ftree = (rtree, ltree) injected

  let leaf    () : ftree = inj @@ distrib @@ X.Leaf
  let node a b c : ftree = inj @@ distrib @@ X.Node (a,b,c)

  (* Injection *)
  let rec inj_tree : inttree -> ftree = fun tree ->
     inj @@ distrib @@ GT.(gmap t nat inj_tree tree)

  (* Projection *)
  let rec prj_tree : rtree -> inttree =
    fun x -> GT.(gmap t) LNat.to_int prj_tree x

end

open Tree
```

(continues on next page)

(continued from previous page)

```
(* Relational insert into a search tree *)
let rec inserto a t' t'' = conde [
  (t' === leaf ()) &&& (t'' === node a (leaf ()) (leaf ()) );
  fresh (x l r l')
    (t' === node x l r)
    Nat.(conde [
      (t'' === t') &&& (a === x);
      (t'' === (node x l' r  )) &&& (a < x) &&& (inserto a l l');
      (t'' === (node x l  l' )) &&& (a > x) &&& (inserto a r l')
    ])
]

(* Top-level wrapper for insertion --- takes and returns non-logic data *)
let insert : int -> inttree -> inttree = fun a t ->
  prj_tree @@ RStream.hd @@
  run q (fun q  -> inserto (nat a) (inj_tree t) q)
        (fun qs -> qs#prj)

(* Top-level wrapper for "inverse" insertion --- returns an integer, which
   has to be inserted to convert t into t' *)
let insert' t t' =
  LNat.to_int @@ RStream.hd @@
  run q (fun q  -> inserto q (inj_tree t) (inj_tree t'))
        (fun qs -> qs#prj)

(* Entry point *)
let _ =
  let insert_list l =
    let rec inner t = function
    | []    -> t
    | x::xs ->
      let t' = insert x t in
      printf "Inserting %d into %s makes %s\n%!" x (show_inttree t) (show_inttree t');
      inner t' xs
    in
    inner Leaf l
  in
  ignore @@ insert_list [1; 2; 3; 4];
  let t  = insert_list [3; 2; 4; 1] in
  let t' = insert 8 t in
  Printf.printf "Inverse insert: %d\n" @@ insert' t t'
```