Typed embedding of relational programming language

Dmitrii Kosarev (presenter), Dmitrii Boulytchev

Saint-Petersburg State University
Dmitrii.Kosarev@protonmail.ch

ML Workshop 2016
Nara, Japan

Example 1:

```
let typecheck: _ → Parsetree.expr → Types.t
```

Example 1:

```
let typecheck: _ → Parsetree.expr → Types.t

let inhabitance: _ → Types.t → Parsetree.expr
```

Example 1:

```
let typecheck: _ → Parsetree.expr → Types.t

let inhabitance: _ → Types.t → Parsetree.expr
```

Example 2:

```
let eval_match: Expr.t → rule list → Expr.t
```

Example 1:

let typecheck: _ → Parsetree.expr → Types.t

let inhabitance: _ → Types.t → Parsetree.expr

Example 2:

let eval_match: Expr.t → rule list → Expr.t

let check_exhaustiveness: Expr.t → rule list → Expr.t

```
...
check_exhaustiveness (ExceptionExpr Not_exhaustive) rules
```

Example 1:

```
let typecheck: _ → Parsetree.expr → Types.t

let inhabitance: _ → Types.t → Parsetree.expr
```

Example 2:

```
let eval_match: Expr.t → rule list → Expr.t

let check_exhaustiveness: Expr.t → rule list → Expr.t
...
check_exhaustiveness (ExceptionExpr Not_exhaustive) rules
```

Reverse execution.

3rd example (1)

```
let rec append xs ys = match xs with
| [] → ys
| h::tl → h :: (append tl ys)
```

3rd example (1)

```
let rec append xs ys = match xs with
| [] → ys
| h::tl → h :: (append tl ys)
```

We are going to emulate *reverse execution* using *relational programming*.

```
function: (xs,ys) -> result
relation: (xs, ys, result)
```

3rd example (2)

```
let rec append xs ys = match xs with
| [] -> ys
| h::tl -> h :: (append tl ys)
```

Going to rewrite as ‘mathy‘ formula:

3rd example (2)

```
let rec append xs ys = match xs with
| [] -> ys
| h::tl -> h :: (append tl ys)
```

Going to rewrite as ‘mathy‘ formula:

```
let rec appendo xs ys rez =
```

3rd example (2)

```
let rec append xs ys = match xs with
| [] -> ys
| h::tl -> h :: (append tl ys)
```

Going to rewrite as ‘mathy‘ formula:

```
let rec appendo xs ys rez =
```

$\vee$

3rd example (2)

```
let rec append xs ys = match xs with
| [] -> ys
| h::tl -> h :: (append tl ys)
```

Going to rewrite as ‘mathy‘ formula:

```
let rec appendo xs ys rez =
  ((xs ≡ []) ∧ (ys ≡ rez))
∨
```

3rd example (2)

```
let rec append xs ys = match xs with
| [] -> ys
| h::tl -> h :: (append tl ys)
```

Going to rewrite as ‘mathy‘ formula:

let rec appendo xs ys rez =

$((\mathtt{xs} \equiv [\,]) \wedge (\mathtt{ys} \equiv \mathtt{rez}))$

$\vee$

$((\mathtt{xs}\equiv \mathtt{h::tl}) \wedge (\mathtt{appendo\ tl\ ys\ tmp}) \wedge (\mathtt{rez}\equiv \mathtt{h::tmp}))$

3rd example (2)

```
let rec append xs ys = match xs with
| [] -> ys
| h::tl -> h :: (append tl ys)
```

Going to rewrite as ‘mathy‘ formula:

```
let rec appendo xs ys rez =
  ((xs ≡ []) ∧ (ys ≡ rez))
∨
  ∃ h,tl,tmp:
    ((xs≡ h::tl) ∧ (appendo tl ys tmp) ∧ (rez≡ h::tmp))
```

```
let rec appendo xs ys rez =
  ((xs === llist_nil) &&& (ys===rez))
  |||
  Fresh.three (fun h tl tmp →
     (xs === h % tl) &&& (appendo tl ys tmp) &&& (rez === x % tmp)
  )
```

```
let rec appendo xs ys rez =
  ((xs === llist_nil) &&& (ys===rez))
  |||
  Fresh.three (fun h tl tmp →
     (xs === h % tl) &&& (appendo tl ys tmp) &&& (rez === x % tmp)
  )
```

Let's try to execute it.

```
# ...give me q such that (appendo [1;2] [3;4] q)
[1;2;3;4]
```

```
let rec appendo xs ys rez =
  ((xs === llist_nil) &&& (ys===rez))
  |||
  Fresh.three (fun h tl tmp →
     (xs === h % tl) &&& (appendo tl ys tmp) &&& (rez === x % tmp)
  )
```

Let's try to execute it.

```
# ...give me q such that (appendo [1;2] [3;4] q)
[1;2;3;4]

# ...give me q such that (appendo q [3;4] [1;2;3;4])
[1;2]
```

```
let rec appendo xs ys rez =
  ((xs === llist_nil) &&& (ys===rez))
  |||
  Fresh.three (fun h tl tmp →
     (xs === h % tl) &&& (appendo tl ys tmp) &&& (rez === x % tmp)
  )
```

```
# ...give me q such that (appendo q [3;4] [1;2;3;4])
[1;2]

# ...give me (q,r) such that (appendo q r [1;2])
([], [1;2])
([1], [2])
([1;2], [])
```

```
let rec appendo xs ys rez =
  ((xs === llist_nil) &&& (ys===rez))
  |||
  Fresh.three (fun h tl tmp →
     (xs === h % tl) &&& (appendo tl ys tmp) &&& (rez === x % tmp)
  )
```

```
# ...give me (q,r) such that (appendo q r [1;2])
([], [1;2])
([1], [2])
([1;2], [])

# ...give me (r,q) such that (appendo r q q)
([], _.0)
```

Relational programming

- Relational programming $\equiv$ miniKanren
- `http://minikanren.org/`
- A family of many languages: $alpha$-Kanren, μKanren, constrained Kanren, etc.
- Implemented in many languages: various LISPS, OCaml, Haskell, F#, Scala, etc.

Contributions of this work

- 2nd implementation of miniKanren in OCaml (more type-safe than first one: `https://github.com/lightyang/minikanren-ocaml`.
- Less verbose programming than other approaches
  - No need for ad-hoc conversions to `term` type.
- No typeclasses used.
- We are reusing OCaml specific features in *unification* (algorithm behind $\equiv$).

About implementations with `term` type

- Special type for relation-related evaluations.
- ```
  type term = Var of int
            | ConstString of string
            | ConstInt of int
            | ListNil
            | ListCons of term * term
            | Atom of string
            | etc...
  ```
- Special convertion functions for every type `t`:

  ```
  val inject_t:  t → term
  val project_t: term → t
  ```
- And `project` is partial by two reasons:
  - Value of type `term` can contain free logic variables. ”
  - The value `ListCons(ConstInt, ListCons(ConstString "a"),ListNil))` cannot be converted neither to OCaml `int list` not to `string list`.
  ```