Check out this one weird trick to make your type-checker look better

Pierre-Évariste Dagand   Jonathan Protzenko
INRIA
http://gallium.inria.fr/blog/

Abstract

The Mezzo programming language features an unusual type-checking discipline, in which several permissions may be available for the same variable. This means the type-checker must consider several possible types for one single variable. This precludes unification-based type-checking, and instead led us into implementing a backtracking procedure that performs forward type-checking.

The present paper extracts the essence of our derivations library. Thanks to a combination of monads, lazy streams and syntactic tricks, not only does the core of our type-checker look very close to the formal presentation of type-checking, but it also automatically builds a derivation, hence allowing the implementors to examine a successful or failed derivation.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords functional programming, testing, quickcheck

1. Introduction

We consider the simplest possible language.

$t ::=$		type
	$t \to t$	arrow
	$()$	unit
$e ::=$		expression
	$e\ e$	application
	$\lambda x.e$	abstraction
	x	variable
	$()$	unit

Figure 1. DumbML, the degenerated language we're considering

This leads to the following OCaml definitions:

```
type typ =
  | TArrow of typ * typ
  | TUnit
```

```
type expr =
  | ELambda of string * expr
  | EApp of expr * expr
  | EUnit
  | EVar of string
```

Not only do we take an excessively simple language, but we also take an excessively stupid algorithm which, instead of using a notion of polymorphism, rather, tries several solutions. That is, when type-checking x, instead of using a unification variable and generalizing (ML) or considering one possible type (simply-typed lambda-calculus), we are going to try several possible types for x, one after another, and see which of these types "work".

The reason why are taking such a surprising approach is that, as we mentioned earlier, in Mezzo, we cannot take a classic, unification-based approach. Mezzo is at the frontier between a program logic and a type system; therefore, the usual approaches that we use when writing a type-checker no longer work. Namely, we had to compromise on the following:

- the type-checker needs to backtrack; therefore, we deal with a stream (lazy list) of solution for each expression rather than one possible type, or zero type if the expression failed to be type-checked;
- error reporting also becomes harder; right now, the derivation library we built constructs derivations or presents the caller with an explanation for the failure.

2. The no-frills version

First, we explain how a streaming type-checker works.

```
type successful_derivation =
  typ MyStream.t (* Invariant: non-empty stream! *)

type outcome =
  successful_derivation option
```

A succesful derivation yields a series of types for a given expression. In order to make things efficient, this is represented using a stream, that is, a lazy list. The outcome of type-checking an expression is thus either a successful derivation, or nothing. At this stage, no actual derivation is being built, and the function just return list of types. Later on, we will build actual derivations.

The main function for type-checking an expression hence no longer returns a `typ` (or a `typ option`), but rather now returns a `result`.

```
let rec check_expr (env: Env.env) (expr: expr): outcome =
  match expr with
  | EApp (e1, e2) ->
      check_app env e1 e2
  | ELambda (x, e) ->
```

ICFP '14, September 1–3, 2014, Copenhagen, Denmark.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
      check_lambda env x e
  | EUnit ->
      check_unit
  | EVar x ->
      check_var env x
```

Similarly, the environment no longer maps variables to "their type", but to `outcome`'s.

```
module Env = struct

  type env = {
    env: outcome StringMap.t
  }

end
```

Our type-checker is dumb: whenever a new variable is introduced in scope, it will attempt to type-check the remainder of the scope with various possible types for the variable.

```
let base_types =
  Some (MyStream.of_list [TUnit; TArrow (TUnit, TUnit)])
```

For instance, when type-checking an abstraction:

```
and check_lambda env (x: string) (body: expr): outcome =
  let sd1 = base_types in
  let env = Env.add env x sd1 in
  check_expr env body
```

The set of base types that is explored for the variable `x` is arbitrary: we could try, beyond $()$ and $() \to ()$, several other types. This determines the fragment of the solution space that we wish to explore.

This means that, when type-checking $\lambda x.x$, we will yield as many types as there are in `base_types`.

The interesting case is that of the application $f\ x$: we take all possible types for f, all possible types for x, and must keep only those that allow for the typing derivation to succeed.

```
and check_app env (e1: expr) (e2: expr): outcome =
  match check_expr env e1, check_expr env e2 with with
  | None, _
  | _, None ->
      None
  | Some sd1, Some sd2 ->
      let sd1 = MyStream.filter is_arrow sd1 in
      let apply t1 t2 =
        match t1, t2 with
        | TArrow (u, v), u' when equal u u' -> Some v
        | _ -> None
      in
      let sd = MyStream.combine apply sd1 sd2 in
      let sd = MyStream.lift_option sd in
      sd
```

A first reason for failure is if the derivation for either f or x fails; in that case, this is an early exit and the application cannot be type-checked. If, however, no early failure occurs, we keep only arrows for f, then take all possible combinations of types for f and x respectively, and retain only those where the type for f is $u \to v$ and the type for x is u, meaning that the type in the conclusion is v.

Finally, it may be the case that none of the combinations work, meaning that the resulting stream contains only `None`. The call to `lift_option` maintains our invariant and makes sure that a stream of failures leads to `None`, while a stream with at least one successful derivation translates to `Some`.

References