A prototype implementation of the manifest contract system in "Manifest Contracts for Datatypes"

Yuki Nishida Taro Sekiyama Atsushi Igarashi

{nishida,t-sekiym,igarashi}@fos.kuis.kyoto-u.ac.jp

December 4, 2014

1 Introduction

This is instruction for a prototype implementation of the manifest contract system described in "Manifest Contracts for Datatypes". In the prototype, you can write programs in OCaml extended with casts by using a modified OCaml interpreter. The static type system is not implemented and the translation from refinement types to datatypes is not implemented, either. The syntax in the prototype is slightly different from that in paper. So please be cautious to use and read the following instructions. If you need, the prototype's source code can be downloaded from `http://rsworks.fam.cx/hg/Contra/`.

2 Files

- `mcc.vdi`: A VirtualBox Disk Image with Debian (64 bit) installed. The image is made by VirtualBox 4.3.10. The root user's password is `root`.
- `instructions.pdf`: This file.
- `paper.pdf`: Our accepted paper.

3 Quick start

1. Launch the virtual machine, and login as the user `mcc` (without password).
2. `$ cd mcc`
3. `$ ocaml`
4. Now, you can use an OCaml interpreter with extended syntax.

4 Examples

The following example casts an integer to a positive integer.

```
# <. {x:int|x>0} <= int > 5;;
- : int = 5
```

This cast succeeds, since `5` is a positive integer. You may notice that the returned value's type is `int` rather than `{x:int|x>0}`. Actually, the prototype only supports run-time checking and does not implement the type system in the paper.

If a cast fails, an exception is raised as follows.

```
# <. {x:int|x>0} <= int > 0;;
Exception: Failure "Cast failure: File \"\", line 1, ...
```

The cast above fails, since `0` is not a positive integer.

Another example is given as follows. This cast checks if a given string is non-empty.

```
# <. {s:string|String.length s > 0} <= string> "hello";;
- : string = "hello"
```

Of course this cast succeeds, since `"hello"` is non-empty.

You can load a program file in the interpreter with `#use` command. For example, if there is the following file `foo.ml`,

```
<. {x:int|x>0} <= int > 1;;
<. {s:string|String.length s > 0} <= string > "hello";;
<. {x:int|x>0} <= int > 0;;
```

then you can load and execute the file as follows.

```
# #use "foo.ml";;
- : int = 1
- : string = "hello"
Exception: Failure "Cast failure: File \"foo.ml\", line 3, ...
```

Though we say the static type checking is not implemented, the prototype checks *type compatibility* syntactically, i.e., compile-time. The following cast is not allowed because `int` and `bool` are incompatible.

```
# <. int <= bool >;;
Error: Failure: "incompatible type cast"
```

5 Extended syntax

In this section, we detail what types can be checked by casts. Our prototype allows not only usual OCaml types such as `int`, `float`, `int list`, etc. but also those concerning contracts in cast expressions. All types introduced here are available in our formal calculus.

There are three kinds of types that can deal with contracts.

- *Refinement type* `{ x : T | e }`, which intuitively denotes values of type `T` satisfying the Boolean expression `e`. Such values are represented by variable `x` of `T` in `e`. For example, as you see above, we can write a type for positive integers as `{ x : int | x > 0 }`.

- *Dependent function type* `x : T1 -> T2`, which means functions that take a value of `T1` and then return a value of the type obtained by replacing variable `x` in `T2` with the argument. For example, `x:int -> {y:int|x < y}` means functions that return an integer which is greater than the argument.

- *Dependent pair type* `x : T1 * T2`, which denotes pairs such that the second component is dependent to the first one. For example, `x:int * {y:int|x <> y}` means pairs of integers where the first component is unequal to the second.

For convenience, we can abbreviate types concerning contracts with keyword `type*` as follows: `type* t = {x:int|x > 0}`—in what follows, we can refer to type `t` as `{x:int|x > 0}`.

Another kind of types concerning contracts is *parameterized datatypes*, which are variant types with contracts. Our first example of parameterized datatypes is given as follows:

```
type* 'a sorted_list [n] || 'a list =
  | SNil  || []
  | SCons || :: of x:{x:'a|n<=x} * 'a sorted_list [x];;
```

This form declares a new datatype `sorted_list`, a variant of the datatype which appears in our paper's introduction, that denotes sorted lists all elements of which are greater than or equal to `n`. This datatype is parameterized in two senses: element type `'a` and the lower bound `n` of lists. The first line says that the new datatype `sorted_list` is parameterized over type variable `'a` and value variable `n` and that its constructors are compatible with those of `list` with keyword `||`. The second and third lines declare constructors `SNil` and `SCons` of `sorted_list`. The forms `|| []` and `|| ::` that follow constructor names says that these constructors are compatible with nil and cons constructors of list, respectively. The type `'a sorted_list [n]` means the type obtained by applying `'a sorted_list` to `n`. More generally, we write `T [e]` an application of type `T` to expression `e`. We will see how dynamic checking for `sorted_list` works later.

As another example, parameterized datatypes can represent lists with element `n`, which appears in Section 2.2 of our paper.

```
type* 'a list_including [n] || 'a list =
  | TCons || :: of {x:'a|x=n}  * 'a list
  | FCons || :: of {x:'a|x<>n} * 'a list_including [n];;
```

This datatype does not have a constructor corresponding with `[]` and does two constructors corresponding with `::`.

Of course, we can declare parameterized datatypes with more general form:

```
type* ('a,'b,...) t [x] || T =
 | A1 || B1 of T1
 | A2 || B2 of T2
 | ...
```

where `A1,A2,...` and `B1,B2,...` are constructors of the new datatype `t` and datatype `T`, respectively. Unlike type variables, the current implementation allows only a single value parameter.

6 More examples

A cast between function types is as follows.

```
# let f = <. int->{x:int|x>0} <= int->int > (fun x -> abs x);;
val f : int -> int = <fun>
```

The cast above should fail because `abs` may return `0`, but that check runs only when the casted function `f` applies to an argument.

```
# f 2;;
- : int = 2
# f 0;;
Exception: Failure "Cast failure: File \"\", line 4, ...
```

The first application cannot detect the cast failure, since the returned value `2` is positive. We can detect the failure at the second application, since the returned value `0` is not positive.

A type abbreviation is defined as follows. Please ignore the response of the type definition, which does not make very much sense.

```
# type* pos = {x:int|x>0};;
type pos = int
val cccaml_prefix_closure4 : int -> bool = <fun>
# <. pos <= int > 2;;
- : int = 2
# <. pos <= int > 0;;
Exception: Failure "Cast failure: File \"\", line 4, ...
```

The following example is a variant parameterized datatype definition of sorted lists in the paper's introduction.

```
# type* 'a sorted_list [n] || 'a list =
  | SNil || []
  | SCons || :: of x:{x:'a|n<=x} * 'a sorted_list [x];;
type sorted_list = SNil | SCons of (int * sorted_list)
val cccaml_prefix_closure2 : 'a -> 'a -> bool = <fun>
```

Then you can cast an integer list into a sorted list as follows.

```
# <. int sorted_list[0] <= int list > [1;2;3];;
- : sorted_list = SCons (1, SCons (2, SCons (3, SNil)))
```

Please see `test.ml` in the `mcc` directory if you are interested in other examples. In the file, a cast surrounded by `TEST[...]` is a successful cast, and a cast surrounded by `FAIL_TEST[...]` is a failed cast.

7 The prototype's internals

This prototype is implemented with Camlp4 which is an extensible OCaml preprocessor. So you can see how the prototype works by using the prototype explicitly as a preprocessor. For instance, if you have the following `bar.ml` file in the `mcc` directory (it's the first example).

```
<. {x:int|x>0} <= int > 5;;
```

You can preprocess the file as follows.

```
$ camlp4o -printer pr_o.cmo ./pp.cma bar.ml
```

The command above is bit complicated, but the important arguments are `./pp.cma` (that is the only file the prototype actually needs) and `bar.ml` (that is the source code you want to preprocess). The output is as follows.

```
(fun (cccaml_prefix_v2 : int) ->
   let x = cccaml_prefix_v2
   in
     if x > 0
     then x
     else failwith "Cast failure: File "bar.ml", line 1, char...)
  5
```

The output is a bit redundant, but you can see the cast is represented by the trivial function which checks the predicate in the refinement type of the cast destination. More complex casts can be implemented by an application of this method.

The difficulty of an implementation is how to choose a constructor when the datatypes in a cast have multiple corresponding constructors. In the prototype, we choose a constructor by trying to cast the constructor arguments. If an attempt fails, program execution backtracks by exception mechanism and tries another constructor. Try to preprocess the following source code if you are interested. The output is complicated but you may find the keywords `try` and `with` which are keywords to deal with exceptions in OCaml. There is backtracking.

```
type* t1 = A of int
type* t2 || t1 =
    B || A of {x:int|x<0}
  | C || A of {x:int|x>0}
<. t2 <= t1 > (A 0)
```

In the prototype, we cannot choose a constructor correctly when the constructor's arguments have dependent function types. That is because a cast between dependent function types does not check the casted value just by applying the cast. So an exception is never occurred, e.g., the prototype always choose the first candidate, and that is barely correct.