Dynamic Contract Checking for OCaml
– A Tutorial

Dana N. Xu
INRIA France
na.xu@inria.fr
January 22, 2014

Contents

1 Introduction 1

2 Contracts 3
  2.1 Predicate contract 3
  2.2 Dependent function contract 4
    2.2.1 Pure functions in contracts 4
    2.2.2 Specifying size properties 5
    2.2.3 Higher order functions can be called in contracts 6
    2.2.4 Higher order contracts 6
  2.3 Dependent tuple contract 7
  2.4 Local contract declaration 9

3 Divergent functions in contracts 10

4 Contract violations in contracts are reported 10

5 Modules 11

6 Command line instructions 12

7 Contract checking algorithm behind the scene 13

1 Introduction

A contract in a programming language is a formal and checkable interface specification that allows programmers to declare what a function assumes and what a function guarantees. In an OCaml program, you can specify a contract for a top-level function with a keyword contract. Consider a simple example:
(* basic.ml *)
contract f = {x | x > 0} -> {r | r > x}
let f x = if x > 0 then x + 1
    else failwith "f requires a positive number"

let t1 = f 2
let t2 = f 0

The contract for f specifies that an input to f must be greater than 0 (indicated by \( \{x | x > 0\} \)) and f should guarantee that after taking a non-negative input, its output is greater than its input (indicated by \( \{r | r > x\} \)).

We can see two calls of f in basic.ml:

- t1 calls f with 2, which satisfies f's precondition;
- t2 calls f with 0, which violates f's precondition.

If we compile basic.ml and run it, we get:

```
Fatal error: exception
  Contract_failure("basic.ml, line: 6, chars: 9-10,
  Blame t2, because t2 fails f's precondition")
```

The error message gives the precise information where a contract violation occurs. It tells the programmer that in the file basic.ml, the call of f (at location (line 6, chars 9-10)) violates the contract and the function t2 (i.e. the caller) is at fault.

Without the contract declaration, a call (f 0) gives the following run-time error message:

```
Fatal error: exception
  Failure("f requires a positive number")
```

which does not tell which caller of f invokes this failure.

Consider another example:

(* basic.ml *)
contract g = {x | x > 0} -> {r | r > x}
let g x = x - 1

let t3 = g 2

We would get the following run-time error message:

```
Fatal error: exception
  Contract_failure("basic.ml, line: 11, chars: 9-10,
  Blame g, because g's postcondition fails")
```
It says that the call of \( g \) (at location (line 11, chars 9-10)) causes a contract violation and the function \( g \) itself is at fault. From the definition of \( g \), we can see that the result of \( g \) is not greater than its input, thus, violates \( g \)'s postcondition.

From these examples, we can see that contracts give a way to reason about your program and help locating the function at fault precisely. This makes debugging more fun!

2 Contracts

Figure 1 gives the abstract syntax tree for contracts. We explain each of them in detail below. Note that the \( \rightarrow \) is right associative and the \( * \) binds stronger than \( \rightarrow \) without brackets.

\[
t ::= \{x \mid p\} \quad (* \text{predicate contract}*) \\
| x:t_1 \rightarrow t_2 \quad (* \text{dependent function contract}*) \\
| x:t_1 * t_2 \quad (* \text{dependent tuple contract}*)
\]

Figure 1: Abstract Syntax Tree for Contracts

2.1 Predicate contract

We use set notation \( \{x \mid p\} \) to mean \( \forall x.p \) holds, where the \( p \) is a boolean-expression in OCaml. Currently, we allow arbitrary pure functions not raising exceptions to be called in the boolean expression \( p \). For example, we may define a function \texttt{sorted}, which tests whether a list is sorted.

\[
\text{let rec sorted xs = match xs with} \\
| [] -> true \\
| [x] -> true \\
| x::y::l -> x <= y && sorted (y::l)
\]

We may have a function \texttt{insert}, which inserts an item to a sorted list and returns a sorted list.

\[
\text{contract insert = \{x \mid \text{true}\} \rightarrow \{y \mid \text{sorted y}\} \rightarrow \{r \mid \text{sorted r}\}} \\
\text{let rec insert item xs = match xs with} \\
| [] -> [item] \\
| x::l -> if item <= x then item::x::l else x::(insert item l)
\]

We can now define a function \texttt{insertion_sort} which always returns a sorted list.

\[
\text{contract insertion_sort = \{xs \mid \text{true}\} \rightarrow \{r \mid \text{sorted r}\}} \\
\text{let rec insertion_sort xs = match xs with} \\
| [] -> [] \\
| x::l -> insert x (insertion_sort l)
\]
2.2 Dependent function contract

Dependent function contracts allow us to specify a dependency between input and output in a function’s postcondition. Its full notation is \( x : t_1 \rightarrow t_2 \) as shown in Figure 1 where \( x \) denotes an element in the precondition \( t_1 \) and can be used in the postcondition \( t_2 \). A simple example of a dependent function contract is:

```ocaml
contract f = k:{x | x > 0} -> {r | r > k}
```

where \( k \) denotes an element in the precondition \( \{x | x > 0\} \) and we can refer to \( k \) in the scope of the postcondition \( \{r | \ldots \} \). If \( k \) denotes an element of a predicate contract, we can omit the \( k \), for example:

```ocaml
contract f = {x | x > 0} -> {r | r > x}
```

We allow such shorthand by assuming the \( x \) in the precondition \( \{x | \ldots \} \) scopes over the RHS of \( \rightarrow \). However, if \( k \) denotes a (dependent) function contract, we cannot omit the \( k \): For example,

```ocaml
contract g = k:({x | x > 0} -> {y | y > x}) -> {z | z > 0}
          -> {r | r > k z}
let g f x = f x + 1
```

when we refer to the function contract, we have to use \( k \) in \( \{r | r > k z\} \). Note that we cannot refer the \( x \) and the \( y \) outside the scope of the function contract, such as

```ocaml
contract g = k:({x | x > 0} -> {y | y > x}) -> {z | z = x}
          -> {r | r > k z}
```

2.2.1 Pure functions in contracts

Properties in contracts are all defined using pure expressions in OCaml itself. Logical operators are also pure functions. Note that the \( (\&\& \text{ }) \) and \( (|| \text{ }) \) in the stdlib are lazily defined and can be used directly. If you want strict versions, you can have the following.

```ocaml
let (&&) x y = if x then y else false
let (||) x y = if x then true else y
let (=>) x y = if x then true else y
let not x = if x then false else true
```

Let us see more examples of what can be specified in contracts.

```ocaml
let null xs = match xs with
  | [] -> true
  | _ -> false

contract hd = {xs | not (null xs)} -> {r | true}
```
let hd xs = match xs with
  | [] -> failwith "hd"
  | x::_ -> x

contract tl = {xs | not (null xs)} -> {r | true}
let tl xs = match xs with
  | [] -> failwith "tl"
  | _::l -> l

contract risers = {xs | true}
  -> {r | (null xs) ==> (null r)}
let rec risers xs = match xs with
  | [] -> []
  | [x] -> [[x]]
  | x::y::l -> let ss = risers (y::l) in
    if x <= y then (x::(hd ss))::(tl ss)
    else [x]::ss

We can see that preconditions of the functions hd and tl are always fulfilled.

2.2.2 Specifying size properties

In many cases, we want to specify size information of a function. For example:

contract length = {xs | true} -> {r | r >= 0}
let rec length xs = match xs with
  | [] -> 0
  | (x::l) -> 1 + length l

contract combine = {xs | true} -> {ys | length xs = length ys}
  -> {rs | length xs = length rs}
let rec combine xs ys = match (xs, ys) with
  | ([],[]) -> []
  | (x::w, y::v) -> (x, y) :: combine w v
  | (_,_) -> failwith "requires arguments of equal length"

Function combine requires both arguments to have the same length and the length of the resulting list is the same as that of the first argument.

For another example:

let min x y = if x <= y then x else y

contract zip = {x | true} -> {y | true}
  -> {r | length rs = min (length x) (length y)}
let rec zip = xs ys = match (xs, ys) with
  | ([],w) | (x,[]) -> []
  | (x::w, y::v) -> (x, y) :: zip w v
Function `zip` does the same as `combine` except that it stops combining two lists when the shorter list reaches its end.

### 2.2.3 Higher order functions can be called in contracts

We can define a higher-order function `filter` whose result is asserted with the help of another recursive higher-order function `for_all`.

```
(* val all : (a -> bool) -> [a] -> bool *)
let rec for_all f xs = match xs with
    | [] -> true
    | x::l -> f x && for_all f l

(* val filter : (a -> bool) -> [a] -> [a] *)
contract filter = {f | true} -> {x | true} -> {z | for_all f z}
let rec filter f xs = match xs with
  | [] -> []
  | x::l -> if f x then x :: filter f l
                                        else filter f l
```

### 2.2.4 Higher order contracts

A higher order contract allows `t_1` in `x : t_1 → t_2` to be a (dependent) function contract itself. For example:

```
(* ho.ml *)
(* val f1 : (int -> int) -> int *)
contract f1 = ({x | x > 0} -> {y | y >= x}) -> {z | z > 0}
let f1 g = (g 0) - 5
let t1 = f1 (fun x -> x + 1)
```

In this case, we get run-time error message:

```
Fatal error: exception
Contract_failure("ho.ml, line: 5, chars: 9-11, 
Blame f1, because f1’s postcondition fails or 
f1 violates its parameter’s contract")
```

It is because `f1` does not use `g` in a way that satisfies `g`'s precondition. But this is detected at the call site of `f1`, i.e. (line 6, chars 9-11).

For another example:

```
contract f2 = ({x | x > 0} -> {y | y >= x}) -> {z | z > 0}
let f2 g = (g 1) - 5
let t2 = f2 (fun x -> x + 1)
```

We get the following error message:
Fatal error: exception

Contract_failure("ho.ml, line: 9, chars: 9-11,
Blame f2, because f2’s postcondition fails")

We can see that the argument of f2, (fun x -> x + 1), satisfies f2’s precondition. From the contract of f2, we can see that x > 0 and y >= x, so y > 0, that is, (g 1) > 0. However, (g 1) - 5 is not greater than 0, thus, f2 fails its postcondition \{z | z > 0\}. That is why the error message blames f2. The location (line 9, chars 9-11) refers to the call site of f2.

Moreover, if we have:

let t3 = f2 (fun x -> x - 1)

we have error message:

Fatal error: exception

Contract_failure("ho.ml, line: 11, chars: 9-11,
Blame t3, because t3 fails f2's precondition")

The location (line 11, chars 9-11) is the call site of f2. Since t3 calls f2 with a function that fails f2's precondition, the function t3 is blamed.

### 2.3 Dependent tuple contract

Currently, we allow tuple contract of arbitrary length. For example, we may have:

(* tuple.ml *)

let fst (a,b) = a
let snd (a,b) = b
(* val f : int array -> int -> int *)

contract f = k:({arr | true} * {i | 0 <= i && i < Array.length arr})
  -> {r | r = snd k}

let f (arr, i) = arr.(i)

let a = Array.init 5 (fun x -> x)
let t1 = f (a, 2)
let t2 = f (a, 8)

Function f takes a tuple of an array arr and an index i and access the i-th element in the array. The contract of f requires the index to be within the bounds of the array, i.e. greater or equal to 0 and less than the length of the array. It also specifies that the result of f should be the same as the second component of the argument (i.e. the index). We can see that the call in t1 satisfies f's precondition while the call in t2 violates f's precondition and gives the following run-time error message:

Fatal error: exception

Contract_failure("tuple.ml, line: 9, chars: 9-10,
Blame t2, because t2 fails f's precondition")
Given a dependent tuple contract \( x : t_1 * t_2 \), if \( t_1 \) denotes a predicate contract, we can use shorthand, for example,

\[
\{\text{arr} \mid \text{true}\} * \{i \mid 0 \leq i && i < \text{Array.length arr}\}
\]

by assuming the variable \text{arr} scopes over the second component. However, if \( t_1 \) is a (dependent) function contract, we cannot omit the \( x \) in \( x : t_1 * t_2 \). For example:

\[
k : (\{x \mid x > 0\} \to \{y \mid y > x\}) * \{z \mid z > k \ 0\}
\]

we cannot omit the \( k \) if we want to use it in \( \{z \mid \ldots \} \).

Moreover, in the contract of \( f \), we can see that the variable \text{arr} and \( i \) do not scope over the RHS of \( \to \). In order to refer to the subcomponents of the argument, we have to define projection function that projects on the subcomponent such as \text{fst} and \text{snd} and use them instead.

Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]

Although the second element of the tuple is not used in the definition of \text{fstPos}.

Due to strict evaluation, a call \text{fst} (5, -1) fails \text{fst}'s precondition. Consider another example:

\[
\text{contract f1} = \{x \mid x > 0\} * \{y \mid y > 0\} \to \{z \mid z > 0\} ; ;
\]

\[
\text{let f1} \ (a,b) = a
\]

\[
\text{let t3} = f1 \ (5, -1)
\]
2.4 Local contract declaration

It is possible to give a contract to a local let-binding. For example:

```
contract bubblesort = {xs | true} -> {r | sorted r}
let rec bubblesort xs =
  let rec bsorthelper sat {xs | true} ->
    {r | (not (snd r)) ==> (sorted (fst r))}
    = function
    | [] -> ([], false)
    | [a] -> ([a], false)
    | x::xs -> let (y::ys, changed) = bsorthelper xs in
      if x <= y then (x ::(y::ys), changed)
      else (y ::(x::ys), true)
  in
  let (result, changed) = bsorthelper xs in
  if changed then bubblesort result
  else result

let test_bubblesort = bubblesort [2,1,5]
```

We give a contract to the local function `bsorthelper` in a format

```
< function name > sat < contract > = < function body >
```

which is analogic to the local function type declaration

```
< function name > : < type > = < function body >
```

In the above example, the result of `bubblesort` is sorted and thus satisfies its postcondition. Let us see another example:

```
contract f = {x | x >= 0} -> {y | y > x}
let f v = let rec h sat {x | x > 5} -> {y | y > x}
    = fun x -> x + v
    and k sat {x | x > 3} -> {y | y > 3}
    = fun x -> h x + 1 in
  k v + h 6

let t1 = f 0
let t2 = f 4
```

The contract of the local function `h` requires its result to be greater than its input. The call `(f 0)` in the definition of `t1` produces `x+0` which is not greater than `x`. Thus, the postcondition of `h` is not satisfied and we get run-time error message:

```
Fatal error: exception
  Contract_failure("local.ml, line: 6, chars: 17-18,
  Blame h, because h's postcondition fails or
  h violates its parameter's contract")
```
where the location (line 6, chars 17-18) refers to the call (h 6).

The local function \( k \) requires its input to be greater than 3, the call \((f 4)\) in the definition of \( t2 \) satisfies this precondition. However, the function \( k \) calls the local function \( h \) with its argument and fails \( h \)'s precondition \( \{x | x > 5\} \). Thus, the call \( h \ x \) (line 5, chars 26-27), in the definition of \( k \), violates \( h \)'s precondition and the function \( k \) is to blame. So we get the following run-time error message:

```
Fatal error: exception
  Contract_failure("local.ml, line: 5, chars: 26-27,
  Blame k, because k fails h's precondition")
```

### 3 Divergent functions in contracts

We only have partial correctness for our contract framework. That is, a divergent expression satisfies any contract and a divergent expression is the only expression that satisfies \( \{x | p\} \) where \( p \) evaluates to false.

Given

```
(* loop.ml *)
contract loop = {x | x > 0} -> {y | false}
let rec loop x = loop x
let t1 = loop 0
let t2 = loop 5
```

The call \((loop 0)\) invokes error message:

```
Fatal error: exception
  Contract_failure("loop.ml, line: 4, chars: 9-13,
  Blame t1, because t1 fails loop's precondition")
```

while a call \((loop 5)\) diverges (i.e. causes stack overflow).

### 4 Contract violations in contracts are reported

As we allow arbitrary pure functions to be called in contracts, it is possible that a call to a function throws an exception in contracts. For example:

```
(* c.ml *)
contract f1 = {x | true} -> {y | hd x > y}
let f1 x = 5
let t1 = f1 []
```

As we gave \( hd \) a contract, we can get run-time error message:

```
Fatal error: exception
  Contract_failure("cntr.ml, line: 16, chars: 33-34,
  Blame f1, because f1 fails hd's precondition")
```
The location (line 16, chars 33-34) refers to the call site of \texttt{hd} in the contract \{y \mid \texttt{hd }x > y\}. Here, "\textit{blame: f1}" means blaming \texttt{f1}'s contract.

For another example:

\begin{verbatim}
contract f2 = k:{x \mid x > 0} -> {y \mid y > x} -> {z \mid k 0 < z}
let f2 g = g 5

let t2 = f2 (fun x -> x + 1)
\end{verbatim}

We have error message:

\texttt{Fatal error: exception Contract_failure("cntr.ml, line: 7, chars: 53-54, Blame f2, because f2 fails k’s precondition")}

We blame the location of the call (\texttt{k 0}), which is in the contract of \texttt{f2}. Note, in this case, the \texttt{f2 in f2 fails k’s precondition} refers to the contract of \texttt{f2}.

To use a partial function in contracts that may throw an exception, we recommend users to give sufficient precondition for the partial function used. For example:

\begin{verbatim}
contract f3 = {xs \mid not (null xs)} -> {z \mid z > \texttt{hd }xs}
let f3 xs = match xs with
  | [] -> failwith "f3"
  | x::xs -> x + 1

let t3 = f3 []
\end{verbatim}

We get error message:

\texttt{Fatal error: exception Contract_failure("cntr.ml, line: 18, chars: 9-11, Blame t3, because t3 fails f3’s precondition")}

We can see that the precondition of \texttt{hd} is always satisfied when the call \texttt{hd xs} is reached.

\section{Modules}

We allow a contract to be defined directly next to its corresponding function in \texttt{.ml} files with a keyword \texttt{contract} mainly because programmers often want to specify contracts for functions that are not exported as well. Exported contracts are in \texttt{.mli} files where they should be syntactically the same as those in \texttt{.ml} files and declared after their type declaration. (That is, if a contract is specified for an exported function, programmers should write this contract in both \texttt{.ml} and \texttt{.mli} files and make sure that they are syntactically the same.) For example:
val f : int -> int
contract f = {x | x >= 0} -> {y | y > x}

It is possible not to write a .mli file. In this case, all contracts declared in the .ml file are presumed to be exported.

Similar ideas apply to nested modules: exported contracts are written in a module signature; if no module signature is given, we assume all contracts declared in the module are exported.

6 Command line instructions

Source code can be obtained by:

svn checkout https://yquem.inria.fr/caml/svn/ocaml/branches/contracts

which is based on the source code of ocamlc-3.11.2. To build ocamlc from source, do:

$ ./configure
$ make ocamlc

Small examples can be downloaded at:

http://gallium.inria.fr/~naxu/research/testcontracts.tar.bz2

I put the folder testcontracts and the folder contracts side by side in the same directory. To release .tar.bz2, do:

$ tar -jxvf testcontracts.tar.bz2

To test one of the examples, e.g. basic.ml, you can do:

$ make basic
$ocamlrun ./basic

To compile all examples:

$ make all

If you have installed the newly built ocamlrun and ocamlc and you do not want to use a Makefile, you can compile your own example as usual:

$ocamlrun ocamlc -I stdlib -c basic.ml
$ocamlrun ocamlc -o basic -I stdlib basic.cmo
$ocamlrun ./basic

If you want to install the newly built ocamlc locally, you need do:

$ make distclean
$.configure --prefix='pwd'/_install
$ make ocamlc
$ make install
This will install everything at local directory \_install. For example, you can see ocamlc, ocamlrun at \_install/bin/ and the standard library is at \_install/lib/ocaml/. You need to give the full path to invoke them.

If you want to switch off the contract checking, you can do:

$ ocamlrun ocamlc -I stdlib -nocontract -c basic.ml

You can look up the flag -nocontract with ocamlc -help.

7 Contract checking algorithm behind the scene

The beauty of contract checking is that it is simple! Given

\[
\text{contract } f = t \\
\text{let } f = e
\]

where the \( t \) refers to the contract of the function \( f \) and the \( e \) refers to the definition of \( f \). The contract checking algorithm is:

\[
\begin{align*}
\ell_1 \ell_2 \{ x | p \} & = \text{let } x = e \text{ in if } p \text{ then } x \text{ else blame } l_1 \\
\ell_1 \ell_2 \text{x:} t_1 \rightarrow t_2 & = \text{fun } v \rightarrow (e (v \ell_2 l_1 t_1)) \ell_1 l_2 t_2 [v \ell_2 l_1 t_1] / x \\
\ell_1 \ell_2 \text{x:} t_1 \ast t_2 & = \text{match } e \text{ with} \\
& \left( x_1, x_2 \right) \rightarrow (x_1 \ell_1 l_2 t_1, x_2 \ell_1 l_2 t_2 [x_1 \ell_1 l_2 t_1] / x)
\end{align*}
\]

The operator \( \ell_1 \ell_2 \) is called a wrapper, which wraps an expression with its contract. If the contract is satisfied, the original expression is returned; otherwise, a contract violation is reported. Here the \( l_i \) contains information about source location and function name of a callee (or a caller). The algorithm makes blaming of a callee or a caller precise. For more information related to contracts, you can read [1, 3, 2].

References
