OContract

Assertion-Based Contracts and Blame Passing for OCaml

A paper presented for **CS4215**

Rayson Koh (A0149884J),
Zhu Hanming (A0196737L)
School of Computing
National University of Singapore

March 2022

Contents

1 Introduction

Assertions as a programming language construct have great practical significance as it has been shown to be very useful in making robust software systems [1]. Also, as functional programming languages becomes increasingly more popular, there is a greater need for such languages to support assertion-based contracts in a higher-order world.

As such, we will be discussing about the design and implementation of OContract, a spin-off of OCaml that supports higher-order contract-based assertions. We hope that OContract would serve as a proof-of-concept of the ideas as described in the landmark paper *Contracts for Higher-Order Functions* [2].

The first main section, Section 2, will largely be on user-level documentation. This would include high-level descriptions of OContract's features as well as some example programs. The second main section, Section 3, will be on developer documentation which would include detailed technical specifications and implementation details of OContract.

2 OContract Features

This section lists the various features that OContract offers. For a complete technical specification of OContract, please refer to the developer documentation in Section 3.

It is recommended that users run OContract programs on the dedicated online code editor at `https://murcia-cs4215.github.io/frontend/`. Alternatively, one can find the latest tagged release of OContract at `https://github.com/murcia-cs4215/ocontract/releases`, and follow the instructions in the README to setup OContract to run locally.

Note that the example programs in this section is displayed as if the program was executed one statement at a time using the interactive Read-Eval-Print-Loop(REPL) shell. For more information regarding how to set up and run OContract programs, please refer to the README of the OContract repo.

2.1 Primitive Types

OContract supports the following primitive types: `int`, `float`, `string`, `char` and `bool`.

A basic expression is simply a primitive value. We can turn such expressions into statements by appending two semicolons:

```
> 10;;
- : int = 10
> 10.5;;
- : float = 10.5
> "abc";;
- : string = "abc"
> '1';;
- : char = '1'
> true;;
- : bool = true
```

2.2 Basic Operators

OContract supports various basic operators:

- Unary `not`
- Arithmetic operators on `int`: `+`, `-`, `/`, `*`, `mod`
- Arithmetic operators on `float`: `+.`, `-.`, `/.`, `*.`, `**` (power)
- Logical operators: `&&`, `||`
- Binary comparison operators: `<`, `>`, `<=`, `>=`, `==`, `!=`, `=`, `<>`

- String concatenation operator: ^

This is an example of a valid OContract program that uses basic operators:

```
> (1 + 2) - 3 * 4;;
- : int = -9
> (1 == 1) && (true || false);;
- : bool = true
```

Note that `=` and `<>` check for structural equality, whereas `==` and `!=` check for physical equality. The latter describes the equality of the address of an object. This is best exemplified via the following example program:

```
> let x : string = "hello";;
val x : string = "hello"
> x = "hello";;
- : bool = true
> x == "hello";;
- : bool = false
> x == x;;
- : bool = true
```

2.3 If Statements

OContract supports if statements in the form of `if` ... `then` ... `else` ... .

An OContract program that uses if statements can be written like so:

```
> if (1 > 2) then "seems wrong..." else "seems correct...";;
- : string = "seems correct..."
```

2.4 Lambda Expressions

Similar to OCaml, OContract allows users to define lambda expressions using the `fun` keyword. Note that **all of the input types and output type have to be clearly specified** for the program to be valid.

To apply a lambda expression to some arguments, simply ensure that the arguments are whitespace-separated and comes after the lambda expression.

Here is an example of an OContract program using lambda expressions:

```
> (fun (x : int) : int -> x + 1) 1;;
- : int = 2
> (fun (x : int) (y : int) : bool -> x > y) 1 2;;
- : bool = false
```

2.5 Let Bindings

Similar to OCaml, OContract allows users to define bindings of the result of an expression to an identifier using the `let` keyword. Note that **the type of the identifier must be clearly defined**.

The binding can be defined to be valid locally via the use of the `let ... in ...` syntax.

Here is an example of some simple let bindings:

```
> let y : int = 10;;
val y : int = 10
> let x : int = 10 in x + 5;;
- : int = 15
> y;;
- : int = 10
> x;;
Uncaught Error: Line 1, Column 0: Unbound value x
```

In addition to binding values to identifiers, lambda expressions are also allowed to be bound to identifiers. This offers a convenient way of reusing the lambda expression by referring to the identifier that is bound to that lambda expression.

Let bindings also allows the user to define a function by defining symbols for the function arguments immediately after the identifier for the function.

For example, the functions `f` and `g` as shown in the two separate programs below are semantically equivalent:

```
> let f : int -> int = fun (x : int) : int -> x + 1;;
val f : int -> int = <fun>
> f 1;;
- : int = 2
> f 2;;
- : int = 3

> let g (x : int) : int = x + 1;;
val g : int -> int = <fun>
> g 1;;
- : int = 2
> g 2;;
- : int = 3
```

2.6 Recursive Functions

OContract supports defining recursive function via the `let rec f ... = ... f ...` syntax.

For example, the recursive factorial function can be defined as such:

```
> let rec fact (x : int) : int = if x == 0 then 1 else x * (fact (x - 1));;
val fact : int -> int = <fun>
> fact 5;;
- : int = 120
```

Note that without the `rec`, the function name will not be bound.

```
> let fact (x : int) : int = if x == 0 then 1 else x * (fact (x - 1));;
Uncaught Error: Line 1, Column 54: Unbound value fact
```

2.7 Function Currying

OContract supports the partial application of a function.

For example:

```
> let f (x : int) (y : int) : int = x + y;;
val f : int -> int -> int = <fun>
> let g : int -> int = f 1;;
val g : int -> int = <fun>
> g 100;;
- : int = 101
```

2.8 Higher-Order Functions

As a functional programming language, OContract allows the user to easily define and apply higher-order functions.

For example, the program below defines a higher-order function `f` that takes in a function `g` and an integer `x`.

```
> let f (g : int -> int) : int -> int = fun (x : int) : int -> g x;;
val f : (int -> int) -> int -> int = <fun>
> f (fun (x : int) : int -> x + 100) 10;;
- : int = 110
```

2.9 Contracts and Blame Assignment

The main feature of OContract is the ability for users to define assertion-based contracts and be notified of contract violations (if any) and the corresponding party to be blamed during runtime.

Not only does this aid in debugging, but it also gives user the reassurance that pre-conditions and post-conditions will be adhered to, regardless of how functions are passed around in the program.

2.9.1 Predicate Contracts

A predicate contract is a contract that is defined on an identifier that holds a primitive value.

Suppose that you wish to define a predicate contract on the identifier `x`, it can be done using the

```
contract x = E1;;
```

syntax, where `E1` denotes an expression that evaluates to a function that takes in the value of `x` as input and returns a `bool`. During runtime, if the returned value is `false`, it would indicate a contract violation and the runtime system will notify the user of that.

For example, the program below shows an example usage of predicate contracts. The contract on `x` only allows `x` to hold integers that are greater than 0. If `x` holds a value that is not greater than 0, a contract violation error will be thrown together with the party to be blamed (also termed the "blame party").

```
    > contract x = (fun (y : int) : bool -> y > 0);;
    val x : int = <contract>
    > let x : int = 1;;
    val x : int = 1
    > let x : int = 0;;
    Uncaught Error: Line 1, Column 0: Contract violation!
    Blame: main
    Contract at: Line 1, Column 0
```

Note that in a REPL context, each line would have a line number of 1, hence the above identical locations for both the violation and the violated contract.

2.9.2 Function Contracts

A function contract is a contract that is defined on an identifier that refers to a function.

To define a function contract on function `f` that takes in k arguments, it can be done using the following syntax:

```
contract f = E1 -> E2 -> ... -> Ek -> Er ;;
```

Each of the `Ei` ($1 \leq i \leq k$) is an expression that evaluates to a function which expects the i-th argument of `f` and returns a `bool`. Finally, the last term in the contract `Er` is an expression that evaluates to a function which expects the return value of fully evaluating the function `f` and returns a `bool`.

Note that if the function takes in k arguments, the contract that is defined on f must have exactly $k + 1$ expressions that are separated by "$\rightarrow$".

The following is an example of a function contract on a first-order function that takes in two integers and return their sum. For brevity, we excluded the definition of a helper function `gt0` that takes in an integer and returns a `bool` based on whether it is greater than 0 or not.

```
    > contract f = gt0 -> gt0 -> gt0;;
    val f : int -> int -> int = <contract>
    > let f (x : int) (y : int) : int = x + y;;
    val f : int -> int -> int = <fun>
    > f 1 1;;
    - : int = 2
    > f 0 1;;
    Uncaught Error: Line 1, Column 0: Contract violation!
    Blame: main
    Contract at: Line 1, Column 0
```

2.9.3 Contracts for Higher-Order Functions

Function contracts can also be used to define contracts for higher-order functions.

For example, the following program defines a contract for a higher-order function `compose` that takes in two functions as input and returns a function that is a composition of the two functions. Again, we assume for brevity that there exists a helper function `gt0` that checks whether an integer is greater than 0.

```
> contract compose = (gt0 -> gt0) -> (gt0 -> gt0) -> (gt0 -> gt0);;
val compose : (int -> int) -> (int -> int) -> int -> int = <contract>
> let compose (f : int -> int) (g : int -> int) : int -> int
              = fun (x : int) : int -> f (g x);;
val compose : (int -> int) -> (int -> int) -> int -> int = <fun>
> compose (fun (x : int) : int -> x + 1) (fun (y: int) : int -> y + 1) 0;;
Uncaught Error: Line 1, Column 61: Contract violation!
Blame: main
Contract at: Line 1, Column 0
```

OContract blames `main` in this case because a faulty value of 0 was supplied to `g` as `g` was expecting an input that is greater than 0.

2.9.4 Contract Set-Notation Syntax

For greater user flexibility, OContract allows a user to define a contract using an alternative Set-Notation syntax.

For instance, predicate contracts can be defined as follows:

```
> contract x = {a : int | a > 0};;
val x : int = <contract>
```

This means that `x` should hold an integer that is greater than 0.

More generally, each term on the right-hand-side of the contract declaration statement (i.e. after the "=" sign) contains two components separated by a "|" symbol. The first component is a symbol that is used to identify the argument, and the second component is an expression (which could possibly involve the use of the symbol defined in the first component) that returns a `bool`.

Note that it is also possible to use set notation syntax as well as the plain expressions interchangeably in the declaration of a contract. See the following example below:

```
> contract f = gt0 -> {y : int | y > 100};;
val f : int -> int = <contract>
> let f (x : int) : int = x + 100;;
val f : int -> int = <fun>
> f 0;;
Uncaught Error: Line 1, Column 0: Contract violation!
Blame: main
Contract at: Line 1, Column 0
```

2.9.5 Dependent Contracts

OContract allows the user to define dependent contracts, which refers having a predicate on parameters or the return value based on the values of the previous arguments.

Note that dependent contracts can only be defined using the set-notation contract syntax.

For instance, consider the following program that defines an adder function.

```
> contract adder = {x : int | x > 0} -> {y : int | y > 0} -> {z : int | z = x + y};;
val adder : int -> int -> int = <contract>
> let adder (x : int) (y : int) : int = x + 1;;
val adder : int -> int -> int = <fun>
```

```
> adder 1 2;;
Uncaught Error: Line 1, Column 0: Contract violation!
Blame: adder
Contract at: Line 1, Column 0
```

The contract of `adder` is violated due to the incorrect implementation of `adder`. Note that this contract violation error cannot be caught if one simply uses `gt0` as the last term in the contract of `adder`.

Hence, in a sense, dependent contracts allow users to define "stricter" contracts that provide stronger assertion guarantees.

2.10 Static Type Checker

OContract enforces types to be declared for all identifiers, arguments and return values.

The strong type safety of OContract results in the early detection of type errors and leads to faster development time. A more rigorous exploration on OContract's type safety will be done in the following Section 3.

For example, the following program shows how OContract is able to detect static type errors.

```
> let x : int = "not an integer";;
Uncaught:
Error: Line 1, Column 0: This expression has type string but an expression
was expected of type int
> let f (x : float) : int = x + 5;;
Uncaught:
Error: Line 1, Column 26: This expression has type float but an expression
was expected of type int
```

In addition, OContract also has type-checking on contracts and is able to detect type mismatch errors.

For example, the following program shows how OContract is able to detect type mismatch errors with respect to contracts.

```
> contract f = {x : int | x > 0};;
val f : int = <contract>
> let f : string = "not an int";;
Uncaught:
Error: Line 1, Column 0: This name has type string but its contract
was expecting type int
> contract g = {x : int | "not a bool"};;
Uncaught:
Error: Line 1, Column 13: This expression has type string but an expression
was expected of type bool
```

Although there is also runtime type checking occurring in OContract, it serves more as a safety net from a programming point of view. The runtime type checking would never find errors should the program pass the static type checker.

2.11 In-Built Functions and Predicates

To enhance the user experience, OContract comes packaged with a subset of OCaml's library functions and constants. These primarily come from OCaml's `Float` and `String` libraries.

Float			String	Others
infinity	neg_infinity	nan	empty	positive
pi	is_nan	of_int	make	negative
to_int	sqrt	cbrt	length	zero
exp	exp2	log	get	any
log10	log2	expm1	starts_with	to_string
log1p	cos	sin	ends_with	
tan	acos	asin	contains	
atan	atan2	hypot	substring	
cosh	sinh	tanh	uppercase	
acosh	asinh	atanh	lowercase	
round	ceil	floor	capitalize	

There are four predicates introduced to make writing contracts easier — `positive`, `negative`, `zero`, and `any`. The first three predicates do as their names suggest. `positive` checks that a value is positive, `negative` checks that a value is negative, and `zero` checks that a value is zero.

What is interesting about them, however, is that they work with a new type called `numeric`.

```
> positive;;
- : numeric -> bool = <fun>
> positive 10;;
- : bool = true
> positive (-0.5);;
- : bool = false
```

In other words, these three functions are able to take in both `int` and `float` values. But do note that this new type `numeric` is only for internal usage, and cannot be used explicitly for type declarations.

Similarly, `any` uses a new internal type called `any`, that accepts all types. `any` thus takes in any value and returns `true`, serving as an effective "empty" predicate for contracts. The example below uses this property to avoid any checks on the return value of `f`.

```
> any;;
- : any -> bool = <fun>
> contract f = positive -> any;;
val f : numeric -> any = <contract>
> let f (x : int) : string = to_string x;;
val f : int -> string = <fun>
> f 20;;
- : string = "20"
```

Do note that, though convenient, excessive usage of `any` defeats the purpose of assertions as a whole.

3 OContract Formal Specification

In this section, we define the formal semantics of OContract and describe the implementation of our interpreter of OContract in detail.

For steps on running an OContract program, please refer to Section 2. Once again, you can find the latest tagged release of OContract at `https://github.com/murcia-cs4215/ocontract/releases`, and follow the instructions in the README to setup OContract to run locally.

We first begin by defining the syntax of OContract. We then bring in the type system and define it using static semantics. This is then synergised with how our interpreter behaves, described using the framework of denotational semantics.

3.1 Syntax of OContract

We divide the syntax of OContract into three categories, *expressions*, *contracts* and *statements*.

Expressions

The set of expressions E is the least set that satisfies the following rules, where x ranges over a set of names V, n ranges over the set of expressible integers, f ranges over the set of expressible floats, c ranges over the set of all characters and s ranges over the set of expressible strings.

We also define p_1 ranging over the set of unary primitive operations $P_1 = \{\texttt{not}\}$ and p_2 ranging over the set of binary primitive operations $P_2 = \{\texttt{||}, \texttt{\&\&}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{mod}, \texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}, \texttt{**}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}, \texttt{=}, \texttt{<>}, \texttt{\^{}}\}$.

Lastly, we define t to represent an arbitrary type. We will further formalise the set of types later.

$$\frac{}{x} \qquad \frac{}{n} \qquad \frac{}{f} \qquad \frac{}{c} \qquad \frac{}{s} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{E}{p_1[E]}\,[\text{UnaryOp}] \qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]}\,[\text{BinaryOp}] \qquad \frac{E \quad E_1 \quad E_2}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2}\,[\text{CondExp}] \qquad \frac{E \quad E_1}{E\ E_1}\,[\text{FuncApp}]$$

$$\frac{E_1 \quad E_2}{\texttt{let } x \texttt{ : } t \texttt{ = } E_1 \texttt{ in } E_2}\,[\text{LocalLet}] \qquad \frac{E_1 \quad E_2}{\texttt{let } x\ (x_1 \texttt{ : } t_1) \cdots (x_n \texttt{ : } t_n) \texttt{ : } t \texttt{ = } E_1 \texttt{ in } E_2}\,[\text{LocalLetFunc}]$$

$$\frac{E}{\texttt{fun } (x_1 \texttt{ : } t_1) \cdots (x_n \texttt{ : } t_n) \texttt{ : } t \texttt{ = } E}\,[\text{LambdaExp}]$$

It may seem counter-intuitive that a local let "statement" is actually an expression. We can see why this is the case using the following example:

```
    > 40 = (let n : int = 2 in n + n) * 10;;
    - : bool = true
```

Contracts

The set of contracts C is the least set that satisfies the following rules:

$$\frac{E}{E} \qquad \frac{E}{\{x \texttt{ : } t \mid E\}}\,[\text{SetNotation}] \qquad \frac{C_1 \quad \cdots \quad C_n}{(C_1 \texttt{ -> } \cdots \texttt{ -> } C_n)}\,[\text{FuncContract}]$$

Statements

Finally, the set of statements S is the least set that satisfies the following rules:

$$\frac{E}{E\texttt{;;}} \qquad \frac{S_1 \quad S_2}{S_1\ S_2}\,[\text{Sequence}] \qquad \frac{C}{\texttt{contract } x \texttt{ = } C\texttt{;;}}\,[\text{ContractDecl}] \qquad \frac{E}{\texttt{let } x \texttt{ : } t \texttt{ = } E\texttt{;;}}\,[\text{GlobalLet}]$$

$$\frac{E}{\texttt{let } x\ (x_1 \texttt{ : } t_1) \cdots (x_n \texttt{ : } t_n) \texttt{ : } t \texttt{ = } E\texttt{;;}}\,[\text{GlobalLetFunc}]$$

For completeness, let us formally define the set of types t as well.

$$\frac{}{\texttt{int}} \qquad \frac{}{\texttt{float}} \qquad \frac{}{\texttt{char}} \qquad \frac{}{\texttt{string}} \qquad \frac{}{\texttt{bool}} \qquad \frac{t_1 \quad t_2}{t_1 \texttt{ -> } t_2}\,[\text{FuncType}]$$

3.2 Syntactic Conventions

We introduce the syntactic conventions to follow:

- We can use parentheses in order to group expressions and types together.
- We use the usual infix and prefix notation for operators.
- The type constructor `->` is right-associative, so that the type

  ```
  int -> int -> int
  ```

  is equivalent to:

  ```
  int -> (int -> int)
  ```

Thus, the function

```
> let f : int -> int -> int = fun (x : int) (y : int) : int -> x + y;;
val f : int -> int -> int = <fun>
```

takes in an integer `x` and returns a function (due to currying), whereas the function

```
> let h : (int -> int) -> int = fun (f : int -> int) : int -> f 10;;
val h : (int -> int) -> int = <fun>
```

takes a function `f` as an argument and returns an integer.

3.3 Static Semantics of OContract's Type System

Not all statements in OContract make sense. For example,

```
contract x = 5;;
```

does not make sense, because we expect some form of callable function for contract checking, while `5` is just an integer. We thus say that this statement is *ill-typed*, because a typing condition is not met. Statements that meet these conditions are what we call *well-typed* in OContract.

An expression `x + 5` within a statement may or may not be well-typed, depending on the type of `x`. For the following parts, we define Γ to denote the type environment. Γ effectively serves as a partial function from names to types, with which a name x is associated with type $\Gamma(x)$.

We further define a relation $\Gamma[x \leftarrow t]\Gamma'$, which constructs a new type environment where $\Gamma'(y)$ is t if $y = x$ and $\Gamma(y)$ otherwise. The set of names, on which a type environment Γ is defined is called the domain of Γ, denoted by $dom(\Gamma)$. We define the empty type environment $\Gamma = \emptyset$.

Lastly, to contract declaration, we will need to define a separate contract type environment Γ_c, which specifically stores contract types and otherwise works much like the normal environment. This is due to the name conflict when a name has both a contract type and its actual type.

The full environment is thus a pair (Γ, Γ_c).

3.3.1 Primitive Values and Symbols

We define the typing relation inductively with the following rules.

$$\frac{}{(\Gamma,\Gamma_c) \vdash x : \Gamma(x)}\,[\text{Var1T}] \qquad \frac{}{(\Gamma,\Gamma_c) \vdash_c x : \Gamma_c(x)}\,[\text{Var2T}] \qquad \frac{}{(\Gamma,\Gamma_c) \vdash n : \texttt{int}}\,[\text{IntT}]$$

$$\frac{}{(\Gamma,\Gamma_c) \vdash f : \texttt{float}}\,[\text{FloatT}] \qquad \frac{}{(\Gamma,\Gamma_c) \vdash c : \texttt{char}}\,[\text{CharT}] \qquad \frac{}{(\Gamma,\Gamma_c) \vdash s : \texttt{string}}\,[\text{StringT}]$$

$$\frac{}{(\Gamma,\Gamma_c) \vdash \texttt{true} : \texttt{bool}}\,[\text{TrueT}] \qquad \frac{}{(\Gamma,\Gamma_c) \vdash \texttt{false} : \texttt{bool}}\,[\text{FalseT}]$$

If $\Gamma(x)$ is not defined, then the first rule is not applicable. If $\Gamma_c(x)$ is not defined, then the second rule is not applicable. In this case, we say that there is no type for x derivable from the assumptions Γ.

3.3.2 Expressions

The following rules describes the set of expressions in OContract, where we define the type of `numeric` to denote `int` or `float` for convenience.

$$\frac{(\Gamma,\Gamma_c) \vdash E \ : \ \texttt{bool}}{(\Gamma,\Gamma_c) \vdash \texttt{not}\ E \ : \ \texttt{bool}}\,[\text{NotT}] \qquad \frac{(\Gamma,\Gamma_c) \vdash E \ : \ \texttt{numeric}}{(\Gamma,\Gamma_c) \vdash -E \ : \ \texttt{numeric}}\,[\text{NegativeT}]$$

For each binary primitive operation p_2, we have a rule of the following form:

$$\frac{(\Gamma,\Gamma_c) \vdash E_1 \ : \ t_1 \qquad E_2 \ : \ t_2}{(\Gamma,\Gamma_c) \vdash p_2[E_1, E_2] \ : \ t}\,[\text{BinOpT}]$$

where the types t_1, t_2, t are given by the following table.

p	t_1	t_2	t
`+,-,*,/,mod`	`int`	`int`	`int`
`+.,-.,*.,/.,**`	`float`	`float`	`float`
`&&,||`	`bool`	`bool`	`bool`
`<=,<,>=,>`	`int`	`int`	`bool`
`<=,<,>=,>`	`float`	`float`	`bool`
`==,!=,=,<>`	t'	t'	t'
`^`	`string`	`string`	`string`

For conditional expressions, we also enforce that the consequent and alternative expressions have the same type:

$$\frac{(\Gamma,\Gamma_c) \vdash E \ : \ \texttt{bool} \qquad (\Gamma,\Gamma_c) \vdash E_1 \ : \ t \qquad (\Gamma,\Gamma_c) \vdash E_2 \ : \ t}{(\Gamma,\Gamma_c) \vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \ : t}\,[\text{IfT}]$$

For local let bindings, the type of the expression is the type of the right hand side, after the type of the left hand side has been bound..

$$\frac{(\Gamma,\Gamma_c) \vdash E_1 \ : \ t_1 \qquad (\Gamma,\Gamma_c)[x \leftarrow t_1](\Gamma',\Gamma_c) \vdash E_2 \ : \ t_2}{(\Gamma,\Gamma_c) \vdash \texttt{let}\ x \ : \ t_1\ \texttt{=}\ E_1\ \texttt{in}\ E_2 \ : \ t_2}\,[\text{LocalLetT}]$$

$$\frac{(\Gamma,\Gamma_c)[x_1 \leftarrow t_1](\Gamma_1,\Gamma_c)\cdots(\Gamma_{n-1},\Gamma_c)[x_n \leftarrow t_n](\Gamma_n,\Gamma_c) \vdash E_1 \;:\; t \qquad (\Gamma,\Gamma_c)[f \leftarrow t_1 \texttt{ -> } \cdots \texttt{ -> } t_n\texttt{-> } t] \vdash E_2 \;:\; t_r}{(\Gamma,\Gamma_c) \vdash \texttt{let}\ f\ (x_1\ :\ t_1)\ \cdots\ (x_n\ :\ t_n)\ :\ t \texttt{ = } E_1 \texttt{ in } E_2 \;:\; t_r}$$

[LocalLetFuncT]

We have a similar rule for lambda expressions:

$$\frac{(\Gamma,\Gamma_c)[x_1 \leftarrow t_1](\Gamma_1,\Gamma_c)\cdots(\Gamma_{n-1},\Gamma_c)[x_n \leftarrow t_n](\Gamma_n,\Gamma_c) \vdash E \;:\; t}{(\Gamma,\Gamma_c) \vdash \texttt{fun}\ (x_1\ :\ t_1)\ \cdots\ (x_n\ :\ t_n)\ :\ t\ \to E\ :\ t_1 \to \cdots \to t_n \to t}$$

Lastly, for function application:

$$\frac{(\Gamma,\Gamma_c) \vdash E\ :\ t_1\ \to\ t_2 \qquad (\Gamma,\Gamma_c) \vdash E_1\ :\ t_1}{(\Gamma,\Gamma_c) \vdash E\ E_1\ :\ t_2}\text{[FuncAppT]}$$

Note that as OContract supports function currying out-of-the-box, the above rule can be extended to work for expressions such as `f 1 2 3`.

3.3.3 Contracts

There are two main types of contracts — predicate contracts (or flat contracts), which work on primitives, and function contracts, which work on functions.

There are two possible syntaxes for predicate contracts:

$$\frac{(\Gamma,\Gamma_c) \vdash E\ :\ t\ \to\ \texttt{bool}}{(\Gamma,\Gamma_c) \vdash E\ :\ \texttt{contract}\ t} \qquad \frac{(\Gamma,\Gamma_c)[x \leftarrow t](\Gamma',\Gamma_c) \vdash E\ :\ t\ \to\ \texttt{bool}}{(\Gamma,\Gamma_c) \vdash \{x\ :\ t \mid E\}\ :\ \texttt{contract}\ t}\text{[SetNotationT]}$$

where `contract` t is defined as the type of a contract that takes in type t.

For function contracts, we thus have:

$$\frac{(\Gamma,\Gamma_c) \vdash C_1\ :\ \texttt{contract}\ t_1\ \cdots\ (\Gamma,\Gamma_c) \vdash C_n\ :\ \texttt{contract}\ t_n}{(\Gamma,\Gamma_c) \vdash C_1 \texttt{ -> } \cdots \texttt{ -> } C_n\ :\ \texttt{contract}\ t_1 \texttt{ -> } \cdots \texttt{ -> } t_n}\text{[FuncContractT]}$$

3.3.4 Statements

The following rules applies for statements in OContract:

$$\frac{(\Gamma,\Gamma_c) \vdash E\ :\ t}{\Gamma \vdash E;;\ :\ t} \qquad \frac{(\Gamma,\Gamma_c)[x \leftarrow t](\Gamma',\Gamma_c) \quad (\Gamma',\Gamma_c) \vdash S\ :\ t'}{(\Gamma,\Gamma_c) \vdash \texttt{let}\ (x\ :\ t)\ =\ E;;\ S\ :\ t'}\text{[Sequence1T]}$$

$$\frac{(\Gamma,\Gamma_c) \vdash S_1\ :\ t_1 \quad (\Gamma,\Gamma_c) \vdash S_2\ :\ t_2}{(\Gamma,\Gamma_c) \vdash S_1\ S_2\ :\ t_2}\text{[Sequence2T]}$$ where S_1 is not a global let nor contract declaration statement.

Contract declarations, by themselves, more or less work like other statements:

$$\frac{(\Gamma, \Gamma_c) \vdash C \ : \ \texttt{contract}\ t}{(\Gamma, \Gamma_c) \vdash \texttt{contract}\ (x \ : \ t) \ = \ C;; \ : \ \texttt{unit}}\text{[ContractDeclT]}$$

where unit is a special type, which is used when the statement does not possess any type. Do note that the type here is not the same as the output you see when you do a contract declaration on the REPL, which is printed out more for usability purposes.

Contract declarations do have the following effect on sequences, however:

$$\frac{(\Gamma, \Gamma_c) \vdash C \ : \ \texttt{contract}\ t \quad (\Gamma, \Gamma_c)[x \leftarrow \texttt{contract}\ t](\Gamma, \Gamma_c') \quad (\Gamma, \Gamma_c') \vdash S \ : \ t'}{(\Gamma, \Gamma_c) \vdash \texttt{contract}\ x \ = \ C;; \ S \ : \ t'}\text{[Sequence3T]}$$

This mainly affects bindings, since type checking is performed when a binding is done to a name that already has a contract.

$$\frac{(\Gamma, \Gamma_c) \vdash E \ : \ t}{(\Gamma, \Gamma_c) \vdash \texttt{let}\ \ (x \ : \ t) \ \ \texttt{=}\ E;; \ : \ \texttt{unit}}\text{[GlobalLet1T] if } x \notin \mathit{dom}(\Gamma_c)$$

$$\frac{(\Gamma, \Gamma_c) \vdash E \ : \ t \quad (\Gamma, \Gamma_c) \vdash_c x \ : \ \texttt{contract}\ t}{(\Gamma, \Gamma_c) \vdash \texttt{let}\ \ (x \ : \ t) \ \ \texttt{=}\ E;; \ : \ \texttt{unit}}\text{[GlobalLet2T] if } x \in \mathit{dom}(\Gamma_c)$$

Similarly for functions:

$$\frac{(\Gamma, \Gamma_c)[x_1 \leftarrow t_1](\Gamma_1, \Gamma_c) \cdots (\Gamma_{n-1}, \Gamma_c)[x_n \leftarrow t_n](\Gamma_n, \Gamma_c) \vdash E \ : \ t}{(\Gamma, \Gamma_c) \vdash \texttt{let}\ f\ (x_1 \ : \ t_1) \ \cdots \ (x_n \ : \ t_n) \ : \ t \ \texttt{=}\ E;; \ : \ \texttt{unit}}\text{[GlobalLetFunc1T] if } f \notin \mathit{dom}(\Gamma_c)$$

$$\frac{(\Gamma, \Gamma_c)[x_1 \leftarrow t_1](\Gamma_1, \Gamma_c) \cdots (\Gamma_{n-1}, \Gamma_c)[x_n \leftarrow t_n](\Gamma_n, \Gamma_c) \vdash E \ : \ t \quad (\Gamma, \Gamma_c) \vdash_c f \ : \ \texttt{contract}\ t_1 \ \texttt{->} \ \cdots \ \texttt{->}\ t_n \texttt{->}\ t}{(\Gamma, \Gamma_c) \vdash \texttt{let}\ f\ (x_1 \ : \ t_1) \ \cdots \ (x_n \ : \ t_n) \ : \ t \ \texttt{=}\ E;; \ : \ \texttt{unit}}$$

[GlobalLetFunc1T] if $f \in \mathit{dom}(\Gamma_c)$

3.4 Semantic Rules

This section covers the denotational semantic rules that specifies the correctness of an OContract program. Note that this section will not cover rules regarding the contract semantics as that will be covered in the next section.

For brevity, types are omitted from certain rules if they are not essential for understanding those rules. In addition, we only consider well-typed OContract programs as per the typing rules in the previous section.

The semantic domains are as follows:

Semantic domain	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\mathbb{Z}$	set of all integers
Float	$\mathbb{R} \setminus \mathbb{Z}$	set of all real numbers excluding integers
Num	$\mathbb{R}$	set of all real numbers
Char	character	character
String	string	string
EV	**Bool + Num + Char + String + Fun**	expressible values
DV	**Bool + Num + Char + String + Fun**	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments
Fun	$\mathbf{DV} * \cdots * \mathbf{DV} \rightsquigarrow \mathbf{EV}$	function values

The semantic rules are as follows:

$$\frac{\Delta \Vdash E \rightarrowtail \texttt{true}}{\Delta \Vdash \texttt{not}\ E \rightarrowtail \texttt{false}} \qquad \frac{\Delta \Vdash E \rightarrowtail \texttt{false}}{\Delta \Vdash \texttt{not}\ E \rightarrowtail \texttt{true}} \qquad \frac{\Delta \Vdash E \rightarrowtail v\ :\ \texttt{numeric}}{\Delta \Vdash -E \rightarrowtail -v\ :\ \texttt{numeric}}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash p_2[E_1, E_2] \ \rightarrowtail v}[\text{BinOp}]$$

$$\frac{\Delta \Vdash E \rightarrowtail \texttt{true} \qquad \Delta \Vdash E_1 \rightarrowtail v}{\Delta \Vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \ \rightarrowtail v} \qquad \frac{\Delta \Vdash E \rightarrowtail \texttt{false} \qquad \Delta \Vdash E_2 \rightarrowtail v}{\Delta \Vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \ \rightarrowtail v}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta[x \leftarrow v_1] \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2 \ \rightarrowtail v_2}$$

$$\frac{\Delta \Vdash E \rightarrowtail (f, \Delta') \quad \Delta \Vdash E_1 \rightarrowtail v_1}{\Delta \Vdash E\ E_1 \ \rightarrowtail \Delta' \Vdash f\ v_1}[\text{FuncApp}]$$

For the Function Application rule above, (f, Δ') is a closure representing the lambda function f and environment Δ'.

$$\frac{}{\Delta \Vdash \texttt{fun}\ x_1 \cdots x_n \rightarrow E \rightarrowtail (f, \Delta)}$$

For the lambda function definition rule above, f is defined to be such that $f\ v_1 \cdots v_n$ yields a result that is equivalent to evaluating $\Delta[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n] \Vdash E$.

$$\frac{\Delta \Vdash E \rightarrowtail v \qquad \Delta[x \leftarrow v] \Vdash S \rightarrowtail v'}{\Delta \Vdash \texttt{let}\ x\ \texttt{=}\ E\ ;;\ S \rightarrowtail v'}[\text{GlobalLet}]$$

For the BinOp rule, the value v is given by the following table. The symbol "op" in the table refers to any of the binary operation in the same row.

Most of the operations should be self-explanatory, such as $v_1\ \texttt{op}_{\texttt{int}}\ v_2$, when $\texttt{op}_{\texttt{int}}$ is "+", then $v = v_1 + v_2$.

p	v
`+,-,*,/,mod`	v_1 $\texttt{op}_{\texttt{int}}$ v_2
`+.,-.,*.,/.,**`	v_1 $\texttt{op}_{\texttt{float}}$ v_2
`&&,||`	v_1 $\texttt{op}_{\texttt{bool}}$ v_2
`<=,<,>=,>`	v_1 $\texttt{op}_{\texttt{int\_comparison}}$ v_2
`<=,<,>=,>`	v_1 $\texttt{op}_{\texttt{float\_comparison}}$ v_2
`==,!=,=,<>`	v_1 $\texttt{op}_{\texttt{equality\_check}}$ v_2
`^`	v_1 $\texttt{op}_{\texttt{string\_concatenation}}$ v_2

3.5 Contract Semantics

There are two types of contracts:

1. Flat / Predicate contracts, which are predicates that can be applied to a primitive value and return boolean values.
2. Function contracts, which apply to functions and have contracts for each of their parameters and the return value.

Each contract comes attached with a **positive** and **negative** party. When a contract is violated, we want to blame the **positive** party.

We only need to set the positive and negative parties for identifiers in the code, since that is where contract checks and violations are needed.

Intuitively, we can think of it as such:

1. Positive party as the **Supplier**, which starts off as the identifier itself
2. Negative party as the **Consumer**, which starts off as the "scope" of the identifier, i.e. nearest global let, else "main".

The initial assignment of the positive and negative blame parties is done statically before the program is run by doing a Depth-First-Search of the AST and keeping track of the current scope at current node.

To capture the additional information of the contract as well as the positive and negative blame parties, we update the environment in which an expression E is evaluated as a tuple consisting of the contract, the positive blame party, the negative blame party, and the environment which stores a mapping of identifiers to values.

The following are the semantic rules for flat contracts. If the predicate on the contract evaluates to `false`, then the postiive blame party is blamed. Otherwise, we simply return the valid v.

$$\frac{\Delta \Vdash E \rightarrowtail v \quad \Delta \Vdash \texttt{con}v \rightarrowtail \texttt{false}}{(\texttt{con}, p, n, \Delta) \Vdash E \rightarrowtail \texttt{blame}\ p}$$

$$\frac{\Delta \Vdash E \rightarrowtail v \quad \Delta \Vdash \texttt{con}(v) \rightarrowtail \texttt{true}}{(\texttt{con}, p, n, \Delta) \Vdash E \rightarrowtail v}$$

The following are the semantic rules for function contracts. The key thing to note is then during function application, the context that is propagated to the argument consists of the argument-portion of the contract, and the positive and negative blame parties are swapped. This is because the consumer of the result of evaluating the argument is now the supplier of the function application expression, and the supplier of the result of evaluating the argument is now the consumer of the function application expression.

$$\frac{\Delta \Vdash E_1 \rightarrowtail (f, \Delta') \quad (\texttt{con1}, n, p, \Delta) \Vdash E_2 \rightarrowtail e_2}{(\texttt{con1} \rightarrow \texttt{con2}, p, n, \Delta) \Vdash E_1 E_2 \rightarrowtail e_2} \text{ if } e_2 \texttt{ is a blame}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail (f, \Delta') \quad (\texttt{con1}, n, p, \Delta) \Vdash E_2 \rightarrowtail e_2 \quad (\texttt{con2}, p, n, \Delta') \Vdash f\ e_2 \rightarrowtail v}{(\texttt{con1} \to \texttt{con2}, p, n, \Delta) \Vdash E_1\ E_2 \rightarrowtail v}\ \text{if } e_2\ \texttt{is not a blame}$$

3.6 Future Work

3.6.1 Type Inference

A good quality-of-life feature to have is to be able to do Type Inference to minimize the number of type annotations that a user needs to write.

3.6.2 Static Contract Checking

Certain classes of contracts may be checked during compile-time to capture contract violations even before the program is run. It could also be an optimization feature since there would be fewer contracts to check during runtime. This can be done via the use of a Theorem Solver such as the Z3 Theorem Prover.

3.6.3 Frontend - Visually highlighting blame location

It would be helpful for the online code editor to be able to visually highlight the exact location where a contract is violated.

3.6.4 Continue on Contract Violations

There may be certain use cases where it might not be ideal for a program to halt immediately when there is a contract violation. In this case, it might signal a warning on the console, or when the program runs to completion, OContract would gather all instances of contract violations and inform the user of them. This would be useful for mission-critical software.

References

[1] D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[2] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, sep 2002.