# Fair termination of binary sessions - Artifact

This is the documentation for `FairCheck`, the artifact associated with POPL submission #30 (*Fair termination of binary sessions*). `FairCheck` is an implementation of the type system described in the paper. More specifically, `FairCheck` reads the specification of a program from a source file and makes sure that:

1. There exists a **typing derivation** for each definition in the program using the algorithmic version of the typing rules described in Appendix F.2.
2. Each process definition is **action bounded**, namely there exists a finite branch leading to termination (Section 5.1).
3. Each process definition is **session bounded**, namely the number of sessions the process needs to create in order to terminate is bounded (Section 5.2).
4. Each process definition is **cast bounded**, namely the number of casts the process needs to perform in order to terminate is bounded (Section 5.3).

## List of claims

Here is a list of claims made in the paper about the well- or ill-typing of some of the presented examples. Each claim is discussed in detail in the corresponding section below.

1. The *acquirer-business-carrier* program in Example 4.1 is well typed (Example 6.1)
2. The *random bit generator* program is well typed (Example 6.3)
3. In Eq. (3), the process $A$ is action bounded and $B$ is not (Section 5.1)
4. At the end of Section 5.1, $A$ is ill typed and $B$ would be well typed if action boundedness was not enforced
5. In Eq. (4) and Eq. (5), $A$ is session bounded whereas $B_1$ and $B_2$ are not (Section 5.2)
6. The process $C$ in Eq. (6) is well typed (Section 5.2)
7. The program in Eq. (7) would be well typed if action boundedness and cast boundedness were not enforced (Section 5.3)
8. The same program using the definitions in Eq. (8) would be well typed if cast boundedness was not enforced (Section 5.3)
9. The program in Eq. (9) is ill typed because it uses unfair subtyping (Section 5.3)

## Download, installation, and sanity-testing

The artifact is packaged in a VirtualBox image running Ubuntu Linux 20.04.3 LTS. Once the image has been downloaded and activated and the operating system has booted, open the terminal (grey icon on the left dock) and type

```
cd FairCheck
```

to enter the folder that contains the source code of `FairCheck` as well as the code of all of the examples that we are going to evaluate.

To make sure that the artifact compiles successfully, issue the commands

```
make clean && make
```

to clean up all the auxiliary files produced by the compiler and to regenerate the `FairCheck` executable. The compilation should take only a few seconds to complete.

`FairCheck` comes along a few examples of well- and ill-typed processes. To verify that they are correctly classified as such, issue the commands

```
make check_examples
make check_errors
```

Each of these commands presents a list of programs being analyzed along with the result of the analysis: a green `OK` followed by the time taken by type checking indicates that the program is well typed; a red `NO` followed by an error message indicates that the program is ill typed.

## Evaluation instructions

### Claim 1

The running example used throughout the paper models an *acquirer-business-carrier* distributed program and is described in Example 4.1. Its specification in the syntax accepted by `FairCheck` is contained in the script `acquirer_business_carrier.pi` and is shown below.

```
type T = !add.(!add.T ⊕ !pay.!end)
type S = ?add.S + ?pay.?end
type R = !add.R ⊕ !pay.!end

Acquirer(x : T) = x!add.x!{add: Acquirer⟨x⟩, pay: close x}

Business(x : S, y : !ship.!end) =
  x?{add: Business⟨x, y⟩, pay: wait x.y!ship.close y}

Carrier(y : ?ship.?end) = y?ship.wait y.done

Main = new (y : !ship.!end)
         new (x : R) [x : T] Acquirer⟨x⟩ in Business⟨x, y⟩
       in Carrier⟨y⟩
```

The script begins with three **session type declarations** defining the acquirer protocol `T`, the business protocol `S` and the dual of the business protocol `R`. Next are the process definitions corresponding to those of Example 4.1. Note that `FairCheck` implements a type checking algorithm, not a type reconstruction algorithm. Hence, **bound names** and **casts** must be **explicitly annotated** with session types. Also, for the sake of readability, **session restrictions** $(x)(P ~|~ Q)$ are denoted by the form `new (x : S) P in Q`.

Only the type `S` of the session endpoint used by `P` is provided. The endpoint used by `Q` is implicitly associated with the dual of `S`.

Example 6.1 claims that this program is well typed. To verify the claim we run `FairCheck` specifying the file that contains the program to type check. Hereafter, the line `----` separates the command being issued (above the line) from the output produced by `FairCheck` (below the line).

```
stack run artifact/acquirer_business_carrier.pi
----
OK
```

The `OK` output indicates that type checking was successful.

### Claim 2

The implementation of the *random bit generator* program in Example 6.3 is contained in the file `random_bit_generator.pi` and shown below.

```
type S = ?more.(!0.S ⊕ !1.S) + ?stop.!end
type U = !more.(?0.U + ?1.!stop.?end)
type V = ?more.(!0.V ⊕ !1.?stop.!end)

Server(x : S) = x?{more: x!{0: Server⟨x⟩, 1: Server⟨x⟩}, stop: close x}
Client(x : U) = x!more.x?{0: Client⟨x⟩, 1: x!stop.wait x.done}
Main          = new (x : V) [x : S] Server⟨x⟩ in Client⟨x⟩
```

Example 6.3 claims that this program is well typed:

```
stack run artifact/random_bit_generator.pi
----
OK
```

### Claim 3

The purpose of the processes defined in Eq. (3) is to illustrate the difference between **action-bounded** processes (which have a finite branch leading to termination) and **action-unbounded** processes (which have no such branch). The $A$ process in Eq. (3) is an example of action-bounded process. Its implementation is contained in the file `equation_3_A.pi` and is shown below.

```
A = A ⊕ done
```

This process may nondeterministically reduce to itself or to `done` and is claimed to be action bounded. In fact, it is also well typed.

```
stack run artifact/equation_3_A.pi
----
OK
```

The implementation of the process $B$ in the same Eq. (3) is contained in the file `equation_3_B.pi`.

```
B = B ⊕ B
```

This process is claimed to be action unbounded, which we now verify using `FairCheck`.

```
stack run artifact/equation_3_B.pi
----
NO: action unbounded process: B@1
```

The `NO` output indicates that the program is ill typed and the subsequent message provides details about (one of) the errors that have been found. In this case, the error indicates that the process `B` is action unbounded. The number after the `@` sign indicates, sometimes approximately, the line number where the error was detected.

### Claim 4

The purpose of the process definitions shown at the end of Section 5.1 is to illustrate how action boundedness helps detecting programs that claim to use certain session endpoints in a certain way, while in fact they never do so. To illustrate this situation, consider the implementation of the process $B$ described at the end of Section 5.1 and contained in `linearity_violation_B.pi`.

```
type T = !a.T

B(x : T, y : !end) = x!a.B⟨x, y⟩
```

This process claims to use `x` according to `T` (which is true) and `y` according to the session type `!end`, while in fact `y` is only passed as an argument in the recursive invocation of `B`. So, while the linearity of `y` is not violated, it is not true that `y` is actually used according to its session type. As claimed in the paper, a process like `B` is not action bounded and is therefore ruled out by the type system.

```
stack run artifact/linearity_violation_B.pi
----
NO: action unbounded process: B@3
```

A conventional session type system that does not enforce action boundedness is unable to realize that `y` is not actually used by `B`. We can verify this claim by passing the `-a` option to `FairCheck`, which disables the

enforcement of action boundedness.

```
stack run -- -a artifact/linearity_violation_B.pi
----
OK
```

The `--` symbols in the issued command are necessary to make it clear that the `-a` option is targeted to `FairCheck` and not to the `stack` tool we use to run it. In conclusion, without the requirement of action boundedness the process `B` would be well typed, despite the fact that it never really uses `y`.

The process $A$, also described at the end of Section 5.1 and contained in the file `linearity_violation_A.pi`, is a simple variation of `B` that is action bounded.

```
type S = !a.S ⊕ !b.!end

A(x : S, y : !end) = x!{a: A(x, y), b: close x}
```

Just like `B`, also `A` declares that `y` is used according to the session type `!end`. This process is claimed to be ill typed, which is expected since the `b`-labeled branch of the label output form does not actually use `y`.

```
stack run artifact/linearity_violation_A.pi
----
NO: linearity violation: y@3
```

## Claim 5

The process definitions in Eq. (4) and Eq. (5) illustrate the difference between **session-bounded** and **session-unbounded** processes. In a session-bounded process, the number of sessions the process needs to create in order to terminate is bounded.

The file `equation_4_A.pi` contains the implementation of the process $A$ in Eq. (4).

```
A = (new (x : !end) close x in wait x.A) ⊕ done
```

The process is claimed to be session bounded, because it does not need to create new sessions in order to terminate despite the fact that it *may* create a new session at each invocation. In fact, the program is well typed.

```
stack run artifact/equation_4_A.pi
----
OK
```

The file `equation_4_B.pi` contains the implementation of the process $B_1$ in Eq. (4).

```
B₁ = new (x : !end) close x in wait x.B₁
```

This process is claimed to be session unbounded. To verify this claim, as we have done for Claim 4, we need to use the `-a` option to disable action boundedness checking or else we would not be able to see the session unboundedness error.

```
stack run -- -a artifact/equation_4_B.pi
----
NO: session unbounded process: B₁@1 creates x@1
```

The tool reports not only the name `B₁` of the process definition that has been found to be session unbounded, but also the name `x` of the session that contributes to its session unboundedness.

Finally, the file `equation_5.pi` contains the implementation of the process $B_2$ in Eq. (5).

```
B₂ = new (x : !a.!end ⊕ !b.?end)
         x!{a: close x, b: wait x.B₂}
     in  x?{a: wait x.B₂, b: close x}
```

This process is claimed to be action bounded (each of the two processes in parallel has a non-recursive branch) but also session unbounded.

```
stack run artifact/equation_5.pi
----
NO: session unbounded process: B₂@1 creates x@1
```

## Claim 6

The file `equation_6.pi` contains the implementation of the program in Eq. (6). The purpose of this example is to show that a well-typed - hence session-bounded - process may still create an *unbounded* number of sessions.

```
C(x : !end) = (new (y : !end) C⟨y⟩ in wait y.close x) ⊕ close x
Main        = new (x : !end) C⟨x⟩ in wait x.done
```

We can run `FairCheck` with the option `--verbose` not only to verify the claim that the program is well typed, but also to show the **rank** (inferred by `FairCheck`) of the process definitions contained therein. The rank of a process is an upper bound to the number of sessions the process needs to create and to the number of casts it needs to perform in order to terminate.

```
stack run -- --verbose artifact/equation_6.pi
----
OK
process C has rank 0
process Main has rank 1
```

We see that the rank of `C` is 0, since `C` may reduce to `close x` without creating any new session. On the other hand, the rank of `Main` is 1, since `Main` may terminate only after the session `x` it creates has been completed.

### Claim 7

The file `equation_7.pi` contains the implementation of the program shown in Eq. (7).

```
type S = !add.S ⊕ !pay.!end
type T = ?add.T + ?pay.?end

A(x : S) = [x : !add.S] x!add.A⟨x⟩
B(x : T) = x?{add: B⟨x⟩, pay: wait x.done}
Main     = new (x : S) A⟨x⟩ in B⟨x⟩
```

The paper claims that this program would be well typed if action boundedness and cast boundedness were not enforced. To verify this claim, we run `FairCheck` with the options `-a` (to disable action boundedness checking) and `-b` (to disable both session and cast boundedness checking).

```
stack run -- -a -b artifact/equation_7.pi
----
OK
```

Note that the option `-b` disables *both* session boundedness and cast boundedness checking. Nonetheless, `FairCheck` detects the violation of each property independently. In particular, the program discussed in Claim 6 has been flagged as session unbounded, whereas the one discussed here is flagged as cast unbounded.

```
stack run -- -a artifact/equation_7.pi
----
NO: cast unbounded process: A@4 casts x@4
```

The error message provides information about the location of the cast that causes `A` to be cast unbounded.

### Claim 8

The purpose of Eq. (8) is to show that, if a program is allowed to perform an unbounded number of casts, it may not terminate even if it is action bounded. The file `equation_8.pi` contains the implementation of the program in Eq. (8).

```
type S  = !more.(?more.S + ?stop.?end) ⊕ !stop.!end
type T  = ?more.(!more.T ⊕ !stop.!end) + ?stop.?end
type SA = !more.(?more.S + ?stop.?end)

A(x : S) = [x : SA] x!more.x?{more: A⟨x⟩, stop: wait x.done}
B(x : T) = x?{more: [x : !more.T] x!more.B⟨x⟩, stop: wait x.done}
Main     = new (x : S) A⟨x⟩ in B⟨x⟩
```

The paper claims that this program is action bounded and cast unbounded. Indeed, each recursive process contains a non-recursive branch and yet it may need to perform an unbounded number of casts in order to terminate.

```
stack run -- artifact/equation_8.pi
----
NO: cast unbounded process: A@5 casts x@5
```

We can run `FairCheck` with the `-b` option to verify that the program is otherwise well typed, and in particular that all the performed casts are valid ones, in the sense that they use fair subtyping.

```
stack run -- -b artifact/equation_8.pi
----
OK
```

## Claim 9

The file `equation_9.pi` contains the implementation of the program shown in Eq. (9).

```
type S  = !more.(?more.S + ?stop.?end) ⊕ !stop.!end
type T  = ?more.(!more.T + !stop.!end) + ?stop.?end
type TA = !more.(?more.TA + ?stop.?end)
type TB = ?more.!more.TB + ?stop.?end

A(x : TA) = x!more.x?{more: A⟨x⟩, stop: wait x.done}
B(x : TB) = x?{more: x!more.B⟨x⟩, stop: wait x.done}
Main      = new (x : S) [x : TA] A⟨x⟩ in [x : TB] B⟨x⟩
```

This program is claimed to be action bounded, session bounded and cast bounded, but also ill typed because the two casts it performs are invalid.

```
stack run -- artifact/equation_9.pi
----
NO: invalid cast for x@8: rec X₄.!{ more: ?{ more: X₄, stop: ?end }, stop:
!end } is not a fair subtype of rec X₃.!more.?{ more: X₃, stop: ?end }
```

During type checking, session types are internally converted into regular trees, although the correspondence between session type trees (within `FairCheck`) and session type names (in the script) is not maintained. Nonetheless, it is relatively easy to see that the session type

```
rec X₄.!{ more: ?{ more: X₄, stop: ?end }, stop: !end }
```

is isomorphic to `S` in the script and that

```
rec X₃.!more.?{ more: X₃, stop: ?end }
```

is isomorphic to `TA`. Indeed, the error message refers to the leftmost cast in the definition of `Main` since `S` is not a fair subtype of `TA`.