Fair Termination of Binary Sessions - Artifact

FairCheck is an implementation of the type system described in the paper *Fair Termination of Binary Sessions* submitted to POPL 2022 (submission #30). A draft of the paper that also includes the algorithmic version of the type system on which FairCheck is based is available here (external link). FairCheck parses a distributed program modeled in a session-oriented variant of the π-calculus and verifies that:

1. There exists a **typing derivation** for each definition in the program using the algorithmic version of the type system (Section 6 of the paper and Appendix F.1 of the supplement).
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 the key examples presented in the paper. Each claim will be discussed and checked against the implementation of FairCheck 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 available on Zenodo (external link) as a VirtualBox image `FairCheck.ova` as well as a source code archive `FairCheck-master.zip`. The next sub-sections describe the steps to be taken to compile and test the artifact in each case.

Using `FairCheck.ova`

The virtual image runs Ubuntu Linux 20.04 LTS and contains all that is necessary in order to compile the source code of `FairCheck`; it has been tested using VirtualBox 6.1 on MacOS 11.6. 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 directory that contains the source code of `FairCheck` as well as the code of all of the examples that we are going to evaluate. This directory is in fact a clone of `FairCheck` GitHub public repository (external link).

Note that the present document with all working hyperlinks can also be visualized on GitHub (external link) or from within the virtual machine by clicking on the FireFox icon in the dock on the left-hand side of the screen.

Using `FairCheck-master.zip`

These instructions assume the use of MacOS with the `homebrew` package manager (external link) and a terminal running the `bash` shell. First of all, make sure that the Haskell compiler and the Haskell Tool Stack are installed. If not, issuing the commands

```
brew install haskell-stack
```

will install these tools. Unpacking the `.zip` archive downloaded from Zenodo will create a directory `FairCheck-master`. From the terminal, type

```
cd FairCheck-master
```

to enter the directory that contains the source code of `FairCheck` as well as the code of all of the examples that we are going to evaluate. This directory is in fact a clone of `FairCheck` GitHub public repository (external link).

Using `FairCheck-master.zip` on an M1 Mac

At the time this artifact is being evaluated, support for the Haskell compiler and the Haskell Tool Stack on M1 Macs is not completely aligned with that of other architectures. In particular, it may be necessary to use a different configuration file for the Haskell Tool Stack to compile the artifact on an M1 Mac. To this aim, in addition to the installation instructions above, install the Haskell compiler globally with the command

```
brew install ghc
```

then edit the `Makefile` and change the topmost line

```
YAML = stack.yaml
```

to

```
YAML = stack_m1.yaml
```

Sanity-testing

To clean up all the auxiliary files produced by the compiler, to (re)generate and install the `FairCheck` executable, issue the command

```
make clean && make && make install
```

The compilation should take only a few seconds to complete. To verify that the `FairCheck` executable has been built and installed successfully, issue the command

```
faircheck
```

to print the synopsis of `FairCheck` and a summary of the options it accepts. We will illustrate the effect of some of these options in the next section. Note that the executable is installed into a hidden local directory `~/.local/bin` that is already included in the `PATH` variable for the terminal shell in the virtual image. In case `FairCheck` is compiled from the `.zip` archive, it may be necessary to add the installation directory of the `stack` tool to the `PATH` environment variable (run `stack path --local-bin` to obtain the full path of this directory).

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

```
make check
```

to print the 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. Depending on the size of the terminal window, it may be necessary to scroll the window up to see the whole list of analyzed programs, divided into those that are well typed and those that are not.

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

A(x : T)                    = x!add.x!{add: A⟨x⟩, pay: close x}
B(x : S, y : !ship.!end) = x?{add: B⟨x, y⟩, pay: wait x.y!ship.close y
}
C(y : ?ship.?end)           = y?ship.wait y.done
Main                        = new (y : !ship.!end)
                                new (x : R) [x : T] A⟨x⟩ in B⟨x, y⟩
                              in C⟨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 checker, not a type reconstruction algorithm. Hence, **bound names** and **casts** must be **explicitly annotated** with session types. For example, the declarations `x : T` in the definition of A and `y : !ship.!end` in the definition of B state that x and y have type T and `!ship.!end` respectively, in agreement with the global type assignments given in Example 6.1. 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 must be provided, whereas 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, $ represents the shell prompt and preceeds the command being issued, whereas any text in the subsequent lines is the output produced by FairCheck.

```
$ faircheck artifact/acquirer_business_carrier.pi
OK
```

The OK output indicates that the program is well typed.

Claim 2

The *random bit generator* program described in Example 6.3 is defined in the script random_bit_generator.pi and is shown below (in the submitted version of the paper, the B process also uses a session endpoint y which is omitted in the script so that the program is self contained).

```
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)

A(x : S) = x?{more: x!{0: A⟨x⟩, 1: A⟨x⟩}, stop: close x}
B(x : U) = x!more.x?{0: B⟨x⟩, 1: x!stop.wait x.done}
Main     = new (x : V) ⌈x : S⌉ A⟨x⟩ in B⟨x⟩
```

Example 6.3 claims that this program is well typed.

```
$ faircheck artifact/random_bit_generator.pi
OK
```

Claim 3

The purpose of the definitions 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 process A in Eq. (3) is defined in the script `equation_3_A.pi`.

```
A = A ⊕ done
```

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

```
$ faircheck artifact/equation_3_A.pi
OK
```

The process B in the same Eq. (3) is defined in the script `equation_3_B.pi`.

```
B = B ⊕ B
```

This process can only reduce to itself and is claimed to be action unbounded, because it has no branch leading to termination.

```
$ faircheck artifact/equation_3_B.pi
NO: action-unbounded process: B [line 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 confirms that B is action unbounded.

Claim 4

The purpose of the process definitions 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 process B shown at the end of Section 5.1 and defined in the script `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 and y according to `!end`. While x is indeed **used** as specified by T, y is only passed as an argument in the recursive invocation of B so that the linearity of y is not violated. As claimed in the paper, a process like B is not action bounded and is therefore ruled out by the type system.

```
$ faircheck artifact/linearity_violation_B.pi
NO: action-unbounded process: B [line 3]
```

A conventional session type system that does not enforce action boundedness may be 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.

```
$ faircheck -a artifact/linearity_violation_B.pi
OK
```

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 defined at the end of Section 5.1 and contained in the script `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 because the b-labeled branch of the label output form does not actually use y.

```
$ faircheck artifact/linearity_violation_A.pi
NO: linearity violation: y [line 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, there is an upper bound to the number of sessions the process needs to create in order to terminate.

The script `equation_4_A.pi` contains 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 any new session in order to terminate despite the fact that it *may* create a new session at each invocation. In fact, the program is well typed.

```
$ faircheck artifact/equation_4_A.pi
OK
```

The file `equation_4_B.pi` contains the definition 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. Since this process is also action unbounded and `FairCheck` verifies action boundedness *before* session boundedness, 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.

```
$ faircheck -a artifact/equation_4_B.pi
NO: session-unbounded process: B₁ [line 1] creates x [line 1]
```

`FairCheck` reports not only the name B_1 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 script `equation_5.pi` contains the definition 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.

```
$ faircheck artifact/equation_5.pi
NO: session-unbounded process: B₂ [line 1] creates x [line 1]
```

Claim 6

The script equation_6.pi contains the definitions of the program in Eq. (6), whose purpose is to show that a well-typed - hence session-bounded - process may still create an *unbounded* number of sessions. The process A discussed in the previous section is already such an example in which the created sessions are *chained* together, so that a new session may be created only after the previous ones have terminated. In this example we see that sessions may also be *nested*, so that a session terminates only after those created after it have terminated as well.

```
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 to verify the claim that the program is well typed and 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.

```
$ faircheck --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 script equation_7.pi contains the definitions of the program in Eq. (7), which illustrates one case where "infinitely many" applications of fair subtyping may have the same overall effect of a single application of unfair subtyping.

```
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).

```
$ faircheck -a -b artifact/equation_7.pi
OK
```

Note that the option -b disables *both* session boundedness and cast boundedness checking. Nonetheless, `FairCheck` is able to distinguish the violation of each property independently. For example, both B_1 and B_2 discussed in Claim 5 are flagged as session unbounded, whereas A discussed here is flagged as cast unbounded.

```
$ faircheck -a artifact/equation_7.pi
NO: cast-unbounded process: A [line 4] casts x [line 4]
```

The error message provides information about the location of the cast that makes A 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 script `equation_8.pi` contains the definitions 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.

```
$ faircheck artifact/equation_8.pi
NO: cast-unbounded process: A [line 5] casts x [line 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.

```
$ faircheck -b artifact/equation_8.pi
OK
```

Claim 9

The script `equation_9.pi` contains the definitions 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.

```
$ faircheck artifact/equation_9.pi
NO: invalid cast for x [line 8]: rec X₄.!{ more: ?{ more: X₄, stop: ?e
nd }, stop: !end } is not a fair subtype of rec X₃.!more.?{ more: X₃,
stop: ?end }
```

Since `FairCheck` internally represents session types as regular trees, the session types printed in error messages may look different from those occurring in the script. However, it is relatively easy to see that the recursive session type

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

in the error message is isomorphic to S in the script and that

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

is isomorphic to TA. So, the error message indicates that S is not a fair subtype of TA. We can verify that the program is well typed if **unfair subtyping** is used instead of fair subtyping by passing the –u option to `FairCheck`.

```
$ faircheck -u artifact/equation_9.pi
OK
```

Additional artifact description

The `FairCheck` directory is structured in this way:

- `src`: Haskell source code of `FairCheck`
- `examples`: some examples of well-typed programs, all of which have also been discussed

in the previous sections
- `errors`: exhaustive set of ill-typed programs aimed at testing all of the errors that can be detected by `FairCheck`. Some (but not all) of these programs have been discussed in the previous sections.
- `artifact`: all of the programs discussed in the previous sections. This is a mixed bag of well- and ill-typed programs.

Within `src`, the source code of `FairCheck` is structured into the following modules:

- `Common`: general-purpose functions not found in Haskell standard library
- `Atoms`: representation of **identifiers** and **polarities**
- `Exceptions`: `FairCheck`-specific syntax and typing **errors**
- `Type`: representation of **session types**
- `Process`: representation of **processes**
- `Lexer`: Alex specification of the **lexical analyzer**
- `Parser`: Happy specification of the **parser**
- `Resolver`: expansion of session types into closed recursive terms
- `Node` and `Tree`: **regular tree representation** of session types
- `Checker`: implementation of the **type checker**
- `Formula`: implementation of **model checker** for the **μ-calculus**
- `Predicate`: **μ-calculus formulas** used in the algorithm for fair subtyping
- `Relation`: implementation of **session type equality**, **unfair subtyping** and **fair subtyping** decision algorithms
- `Render`: **pretty printer** for session types and error messages
- `Main`: main module and handler of command-line options

The `FairCheck` parser accepts a syntax that is close to, but not exactly the same as, the one used in the paper. The table below shows the grammar of scripts. Square brackets enclose optional parts of the syntax.

Entity		Definition	Description
x, y		non-capitalized identifier (e.g. x, y, ...)	Channel name
l		non-capitalized identifier or number (e.g. a, add, `0`, ...)	Label
X		capitalized identifier (e.g. S, T, ...)	Type name
A		capitalized identifier (e.g. A, `Main`, ...)	Process name
π	::=	?	Input polarity
		!	Output polarity
Script	::=	TypeDef_1 ... TypeDef_m ProcessDef_1 ... ProcessDef_m	
TypeDef	::=	X = Type	Type definition

ProcessDef	::=	A [(x_1 : Type , … , x_n : Type)] = Process	Process definition
		A [(x_1 : Type , … , x_n : Type)] ;	Undefined process declaration
Process	::=	`done`	Terminated process
		`close` x	Signal output
		`wait` x . Process	Signal input
		x π (y) . Process	Channel input/output
		x π { l_1 : Process , … , l_n : Process }	Label input/output
		x ! l . Process	Shortcut for label output
		`new` (x : Type) Process `in` Process	New session
		Process ⊕ Process	Non-deterministic choice
		⌈ x : Type ⌉ Process	Cast
		A [⟨ x_1 , … , x_n ⟩]	Invocation
		(Process)	Bracketed process
Type	::=	π `end`	Terminated session
		π Type . Type	Channel input/output
		π { l_1 : Type , … , l_n : Type }	Label input/output
		π l . Type	Shortcur for label input/output
		Type + Type	Shortcut for external choice (label input)
		Type ⊕ Type	Shortcur for internal choice (label output)
		X	Type name
		`rec` X . Type	Recursive type
		(Type)	Bracketed type