Qafny Design and Bookkeeping

1 Typing

1.1 Typing with Sessions and Ranges

For example, let's say we have the full session

```
s == { x [0 .. 1]; y [0 .. 1]; z [0 .. 1] }
```

For a λ application statement, let's say

```
x [0 .. 1] *= λ (x => x + 1)
```

The typing procedure should locate `x[0 .. 1]` the corr. session `s` and identify that the session `s` is indeed of `CH` type. With the `CH` type ensured, we then should emit the term for the oracle operator that should *only* modify the range corr. to `x[0 .. 1]` instead of the entire session.

Therefore, a reasonable way to do this is to have following functions

```
-- | locate session with a range
findSession :: Range -> Transform Session

-- | locate a session with a (sub-)session
subSession :: Session -> Transform Session

-- | the typing of session (already done)
instance Typing Session QTy

-- | sub-range based codegen
augWithSession :: Session -> ? -> Transform ?
```

However, I think there might be other case where you need to compare if two sessions are exactly the same.

1.1.1 Sidenote

At the same time, by viewing a session as a row, I see the connection between the subtyping between sessions w/ or w/o ranges and the open and close rows in the row polymorphism.

1.2 Type Coercion, or Casting

I need to implement a new function to perform type coercion between `QTy`'s. When subtyping is satisfied, insert a cast to lift the subtype to its super type. This needs a support from `QPreludeUntyped` module.

2 `QPreludeUntyped` and Dafny 4.0

2.1 Ghost Functions

In the new standard, `function` is now `function method` by default, and `ghost function` represents `function` in Dafny 3.

2.2 `abstract module`

Dafny 4.0 introduced the notation `abstract module` which are modules not to be compiled. The upstream libraries seems to be adapted for that. (That's exactly what we need.) To conform to the standard, `Codegen` now generates a default abstract module wrapper to enable us to use those functions.

3 Compilation

3.1 `separates` Keyword and Body

I introduced `separates` keyword as a hint for the split-combine semantics for non-trivial guards.

Let's say, if we have a guard

```
- _ leading
if (f (x [0 .. n]))
  separates x <_ .. m>

- _ trailing
if (f (x [0 .. n]))
  separates x <m .. _>

- segment
if (f (x [0 .. n]))
  separates x <i .. j>
```

The `separates` side condition asserts that only the sequence of **states** in `x` between `i` and `j` actually satisfies `f (x [0 .. n])`. I will need to insert assertions when implementing assertion translations.

3.2 CH Biview

`CH` is now expected to have two views: `bitvector` versus `nat`.

I think it's unnecessary to track views by installing a new state field. A tentative solution is to make 2 `CH` types, `CHb` and `CHn`, to distinguish those two views.

3.3 Biview Preference

From previous implementations, the GHZ one favors `CHb` and the Shor's one favors `CHn`. I think there's a pattern: in GHZ, the guard is of `CH` type and the λ oracle is *flip* (a very bit-level operation), while in Shor's, the guard is `Had` type and the λ oracle is modulo multiplication which is on `nat`.

3.4 Placeholder for Phase Calculus

The following instance is responsible for mapping a session type to its emission type which should include the emitted type for kets as well as phases.

```
instance Typing QTy [Ty]
```

Currently, this list is always a singleton and is flatten by the `only1` combinator. Some lifting operation and bijective mapping will be expected when starting the phase calculus implementation.

4 Language Design

4.1 State Predicate

What would state predicates be like now?

5 Misc

5.1 Biview

Coincidentally, the choice between a sequence of bitvectors vs a sequence of nats is closely related to the idea in [Wadler 1987]

5.2 Point-free Translation of `buildOp2`

```
x <> ("&&" <!> y)
  = (<>) x  ("&&" <!> y)
  = (flip (<>)) ("&&" <!> y) $ x
  = (flip (<>)) (("&&" <!>) y) $ x
  = (flip (<>)) . ("&&" <!>) $ y x
```

References

[1] P. Wadler. "Views: a way for pattern matching to cohabit with data abstraction". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '87. New York, NY, USA: Association for Computing Machinery, Oct. 1987, pp. 307–313. ISBN: 978-0-89791-215-0. DOI: 10.1145/41625.41653. URL: https://doi.org/10.1145/41625.41653 (visited on 02/10/2023).