quilc
The Rigetti Quil compiler

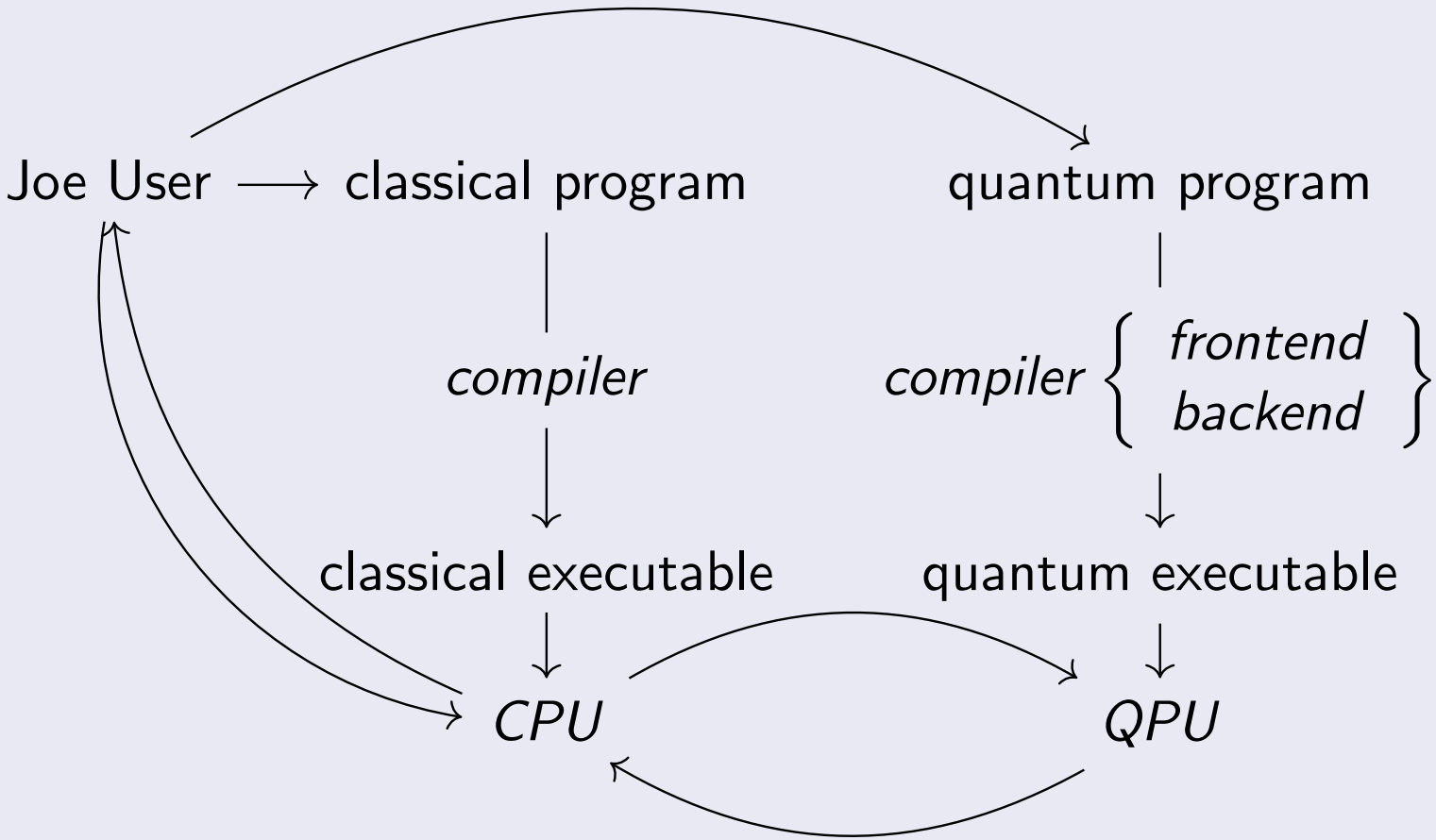
Eric Peterson

Rigetti Quantum Computing

August 16, 2019

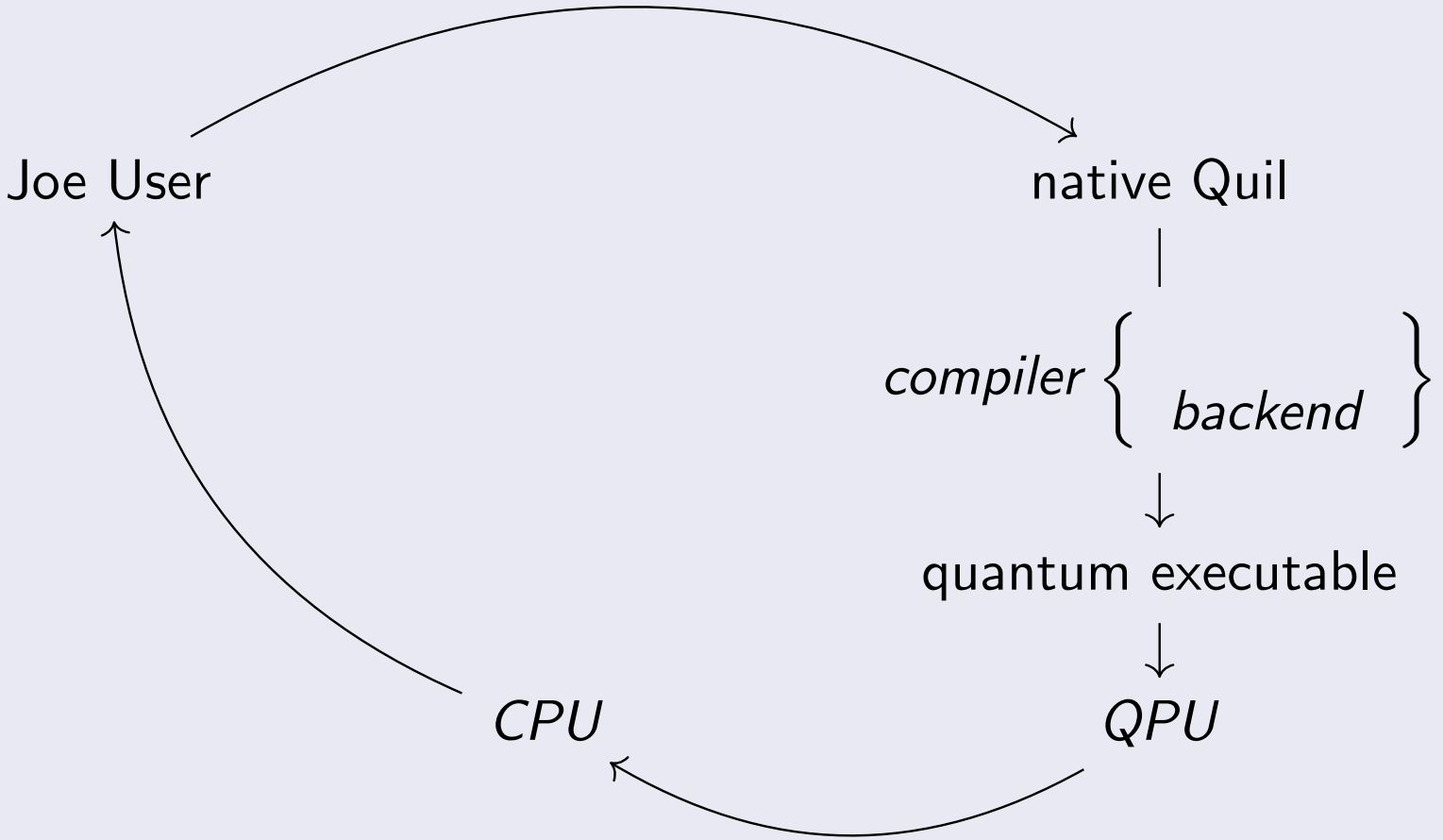
Rigetti's execution stack

The stack: Ideal



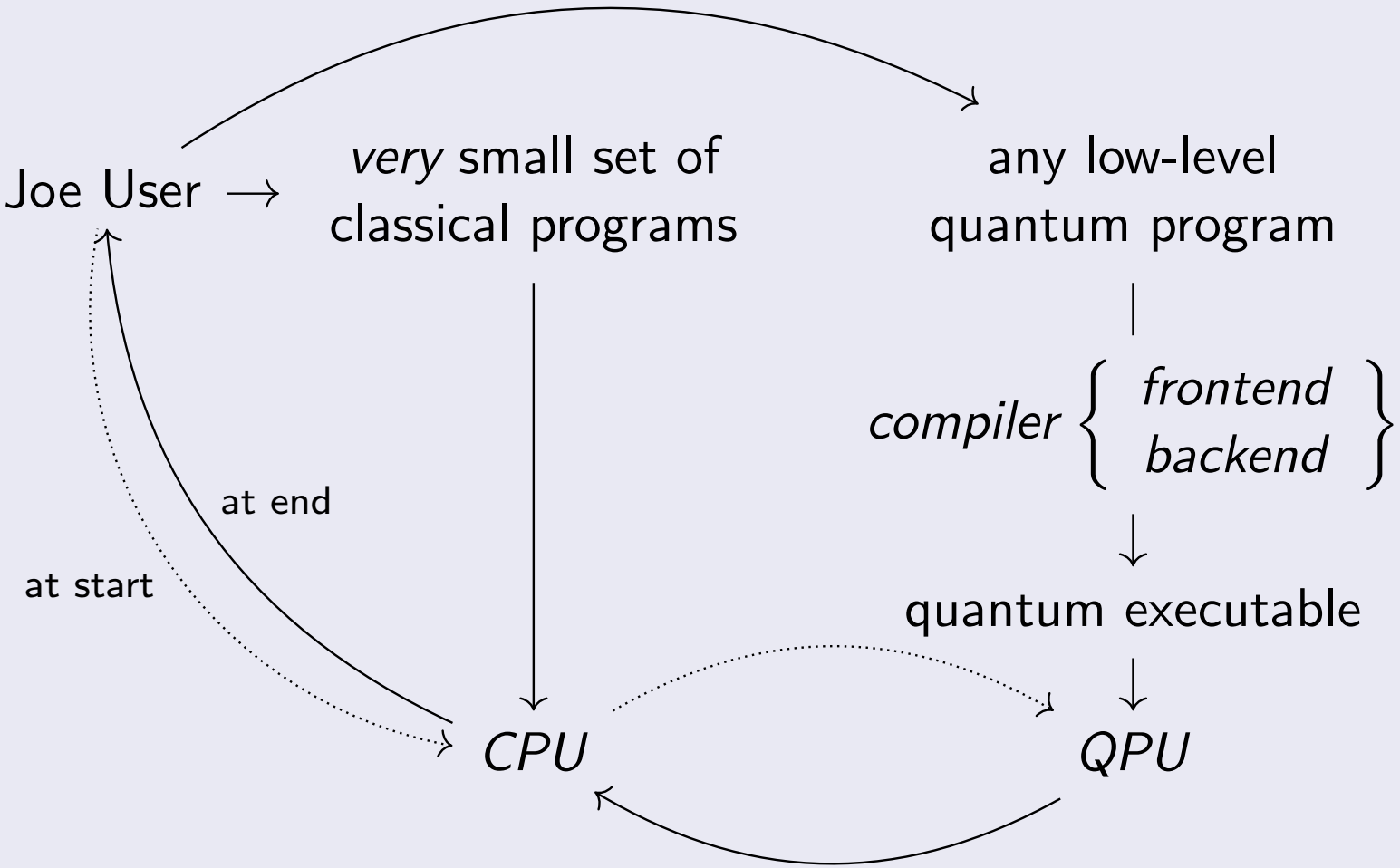
Rigetti's execution stack

The stack: Summer 2017



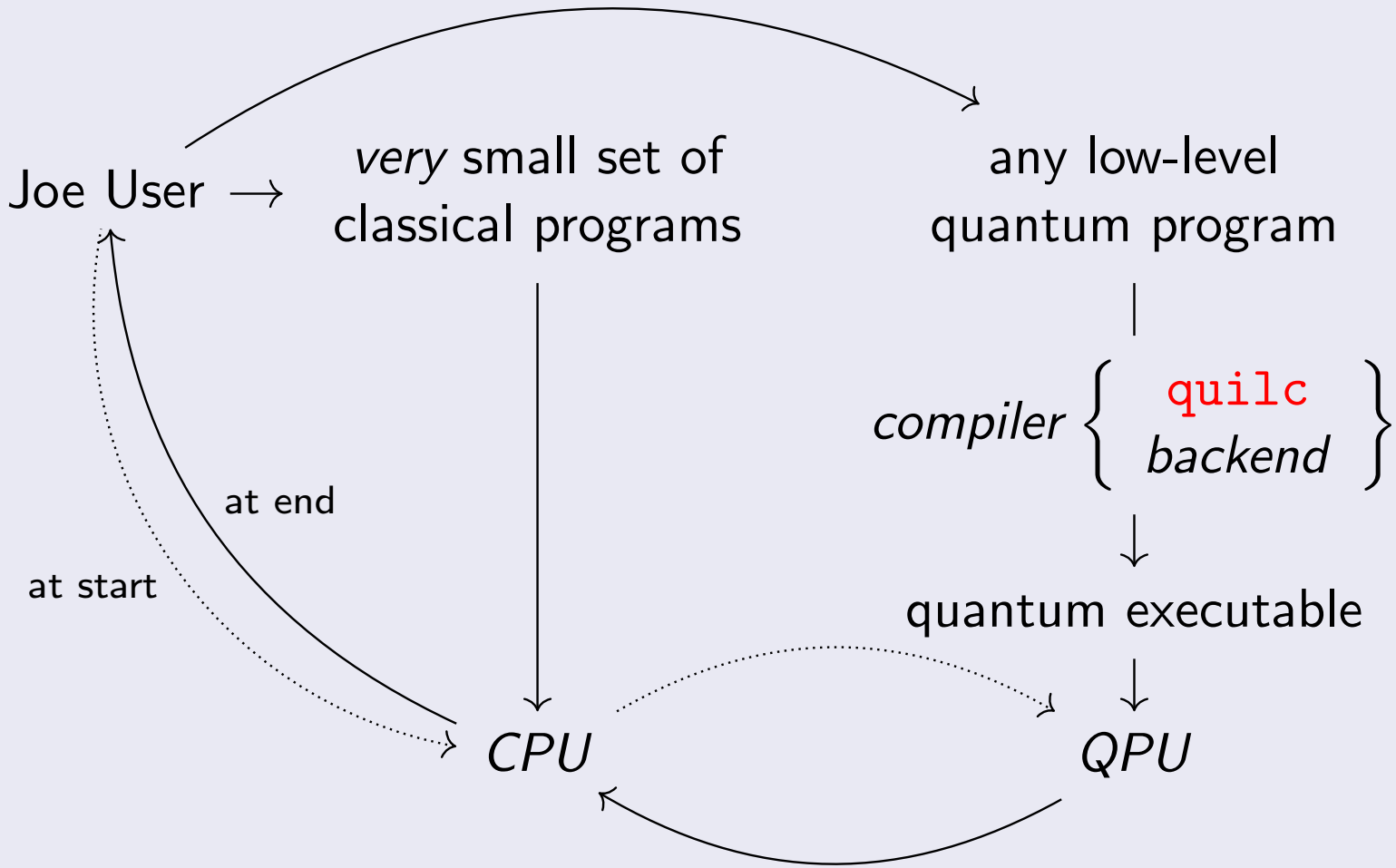
Rigetti's execution stack

The stack: Today



Rigetti's execution stack

The stack: Today



Quil, the language

High-level

Assembly-like. Small set of instructions, qubit "registers".

```
DECLARE    ro BIT[32]
CNOT       0  1
RX(pi/2)   4
MEASURE    4  ro[4]
```

Quil, the language

High-level

Assembly-like. Small set of instructions, qubit "registers".

```
DECLARE    ro BIT[32]
CNOT       0  1
RX(pi/2)   4
MEASURE    4  ro[4]
```

Salient features

Portability Gate-based architecture assumed, not much else.

Hybrid prim. Language-level support for:

1. `yield`-type functionality,
2. simple classical reasoning,
3. memory shared between the coprocessors.

Simplicity Simple grammar, simple semantics, minimal functionality.

quilc: A typical run

Main tasks

- Complex, high-level, device non-specific operations become simple, low-level, device-specific operations.
- Identify clever decompositions and deploy them.
  Identify unnecessary work and eliminate it.

quilc passes

Analyze CFG Segment "purely quantum" subprograms from intervening classical computation. Eliminate unnecessary logic.

Address Decompose complex operations and marry user-specified resources to target device resources.

Optimize Remove redundant gates.

Reconstitute Prepare program for transmission.

Each of these involves crawling the program in a distinct way.

quilc: Guiding principles

Primary use case

Execution of advantage-class algorithms on our devices.

Supporting principles

- Monolithic operation
- Retargetability
- Production-grade software

quilc: Guiding principles

Monolithic operation

Problem: Interplay of passes in compiler makes it hard to predict the full effect of any particular optimization.

Redress: Encourage new interactions among new passes, and make the default mode of use cash in on these interactions.

quilc: Guiding principles

Monolithic operation

Problem: Interplay of passes in compiler makes it hard to predict the full effect of any particular optimization.

Redress: Encourage new interactions among new passes, and make the default mode of use cash in on these interactions.

Case study: 2Q depth of random 4Q interaction on a line

gateset	old	QISKit challenge	all-to-all
CZ	230 (2.0x)	210 (1.82x)	115 (1.0x)
CZ + ISWAP	180 (1.56x)	130 (1.13x)	115 (1.0x)

quilc: Guiding principles

Monolithic operation

Problem: Interplay of passes in compiler makes it hard to predict the full effect of any particular optimization.

Redress: Encourage new interactions among new passes, and make the default mode of use cash in on these interactions.

Case study: 2Q depth of random 4Q interaction on a line

gateset	old	QISKit challenge	all-to-all
CZ	230 (2.0x)	210 (1.82x)	115 (1.0x)
CZ + ISWAP	180 (1.56x)	130 (1.13x)	115 (1.0x)
reweighted	230 (2.0x)	184 (1.57x)	

Guess: `CZ` and `ISWAP` are "mirror gates": $\texttt{SWAP} \equiv \texttt{ISWAP} \cdot \texttt{CZ}$.
Try: Weight `ISWAP` as twice `CZ`.

quilc: Guiding principles

Monolithic operation

Problem: Interplay of passes in compiler makes it hard to predict the full effect of any particular optimization.

Redress: Encourage new interactions among new passes, and make the default mode of use cash in on these interactions.

Case study: 2Q depth of random 4Q interaction on a line

gateset	old	QISKit challenge	all-to-all
CZ	230 (2.0x)	210 (1.82x)	115 (1.0x)
CZ + ISWAP	180 (1.56x)	130 (1.13x)	115 (1.0x)
reweighted	230 (2.0x)	184 (1.57x)	

Guess: `CZ` and `ISWAP` are "mirror gates": $\texttt{SWAP} \equiv \texttt{ISWAP} \cdot \texttt{CZ}$.
Try: Weight `ISWAP` as twice `CZ`.
Conclusion: Somehow, CZ + ISWAP + QISKit strategy actually does better than CZ + ISWAP or QISKit strategy on their own.

quilc: Guiding principles

Retargetability

Problem: Chips' intended design / actual features / operating points are in flux on timescales of months / weeks / hours.

Redress: Make it minimum-effort to target new devices.

Data

A "QPU graph" Σ: vertices are qubits, edges are qubit-qubit interactions. Each piece is tagged with native operations, their matrix encodings, execution properties (e.g., fidelity, duration), ...

Data $\mapsto$ Strategies

Address Fixed set of strategies available, each suitable for dealing with any (perhaps irregularly) shaped device.

Nativize/Optimize Starting from Σ's native gate set, backsolve for compilation subroutines that rewrite an arbitrary program into the desired gateset.

quilc: Retargetability, expanded

Tiny example Σ with defaults

```
prog = """H           0
          RY(pi/3)   0
          RZ(pi/8)   0
          RX(2*pi/5) 0"""

isa = {"1Q":
       {"0": None}}
```

```
RZ(-2.06171187047457) 0  # Entering rewiring: #(0)
RX(pi/2) 0
RZ(1.656541137935280) 0
RX(-pi/2) 0
RZ(1.088693765816577) 0  # Exiting rewiring: #(0)
```

quilc: Retargetability, expanded

Σ with free RXs, RZs

```
isa = {"1Q":
       {"0":
        [{"op": "RZ", "params": ["_"], "args": [0],
          "fidelity": 0.99, "duration": 80},
         {"op": "RX", "params": ["_"], "args": [0],
          "fidelity": 0.99, "duration": 80}]}}
```

```
RX(-0.8789605131516703) 0  # Entering rewiring: #(0)
RZ(-0.6433291804340883) 0
RX(-0.8994990281493116) 0  # Exiting rewiring: #(0)
```

quilc: Retargetability, expanded

Σ with free RXs, RZs; preferring RZs

```
isa = {"1Q":
       {"0":
        [{"op": "RZ", "params": ["_"], "args": [0],
          "fidelity": 0.999, "duration": 80},
         {"op": "RX", "params": ["_"], "args": [0],
          "fidelity": 0.99, "duration": 80}]}}
```

```
RZ(2.6506771099101 2) 0  # Entering rewiring: #(0)
RX(1.6565411379352 8) 0
RZ(2.6594900926114 7) 0  # Exiting rewiring: #(0)
```

quilc: Retargetability, expanded

Σ with free RXs, RZ($\mathbb{Z}\pi/2$)

```
isa = {"1Q":
       {"0":
        [{"op": "RZ", "params": [pi/2], "args": [0],
          "fidelity": 0.99, "duration": 80},
         # ... other multiples of pi/2 ...
         {"op": "RX", "params": ["_"], "args": [0],
          "fidelity": 0.999, "duration": 80}]}}
```

quilc: Retargetability, expanded

Σ with free RXs, RZ($\mathbb{Z}\pi/2$)

```
isa = {"1Q":
       {"0":
        [{"op": "RZ", "params": [pi/2], "args": [0],
          "fidelity": 0.99, "duration": 80},
         # ... other multiples of pi/2 ...
         {"op": "RX", "params": ["_"], "args": [0],
          "fidelity": 0.999, "duration": 80}]}}
```

```
Condition CL-QUIL::NO-COMPILER-PATH was signalled.

Backtrace:
  0: FIND-SHORTEST-COMPILER-PATH
  1: COMPUTE-APPLICABLE-COMPILERS
  2: WARM-HARDWARE-OBJECTS
  ...
```

quilc: Retargetability, expanded

Teach quilc a new trick

```
(define-compiler RZ-to-XZXZX
    ((rz-gate (RZ (alpha) q)))
  (inst RX '(  #.-pi/2)  q)
  (inst RZ '(   #.pi/2)  q)
  (inst RX `(,(- alpha)) q)
  (inst RZ '(  #.-pi/2)  q)
  (inst RX '(   #.pi/2)  q))
```

```
RX(-2.449756839946568) 0  # Entering rewiring: #(0)
RZ(pi/2) 0
RX(0.6433291804340882) 0
RZ(-pi/2) 0
RX(0.6712972986455852) 0  # Exiting rewiring: #(0)
```

quilc: Retargetability, expanded

Question

For native gates Γ, howcan a gate G be written as $G = \gamma_1 \cdots \gamma_n$?

quilc: Retargetability, expanded

Question

For native gates Γ, ~~how~~ when can a gate G be written as $G = \gamma_1 \cdots \gamma_n$?

quilc: Retargetability, expanded

Question

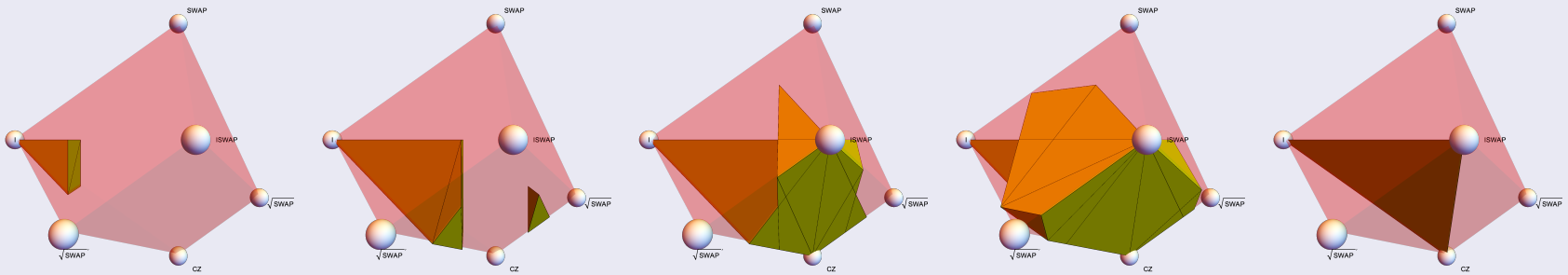
For native gates Γ, ~~how~~ when can a gate G be written as $G = \gamma_1 \cdots \gamma_n$?

A sort of answer (P.–Crooks–Smith)

For $\Gamma = \Gamma_{2Q} \cup \{\text{1Q gates}\}$, G decomposes iff certain numerical invariants of G belong to a certain finite union of convex polytopes.

Example: $G = \gamma_{1Q} \cdot \mathrm{XY}(k\pi/5) \cdot \gamma_{1Q} \cdot \mathrm{XY}(k\pi/5) \cdot \gamma_{1Q}$

Some polytopes indicating when G decomposes, $1 \leq k \leq 5$:



quilc: Guiding principles

Production-grade software

Problem: Compilers are *highly* complex pieces of software, prone to sprawl, slowness, tricky bugs, ...

Redress: Follow SWE best-practices: keep speed on the table, write highly extensible code, make using it feel like other compilers, build off of half a century of people thinking about compiler design.

quilc and Common Lisp

Common Lisp pros

Exploratory Easy to feel out a poorly understood problem space.

Fast Very speedy executables.

Stable Extremely stable language definition. Compilers from the '80s are still a rich source of information today.

Extensible CL extends to cover the problem at hand, easing your colleagues' work (e.g.: `define-compiler`).

Common Lisp cons

Unfamiliar Physicists are reluctant to learn it.
"My time is more valuably spent doing physics."

Insulated Interoperation with outside software requires attention: can be done, varies by implementation, isn't plug-and-play.

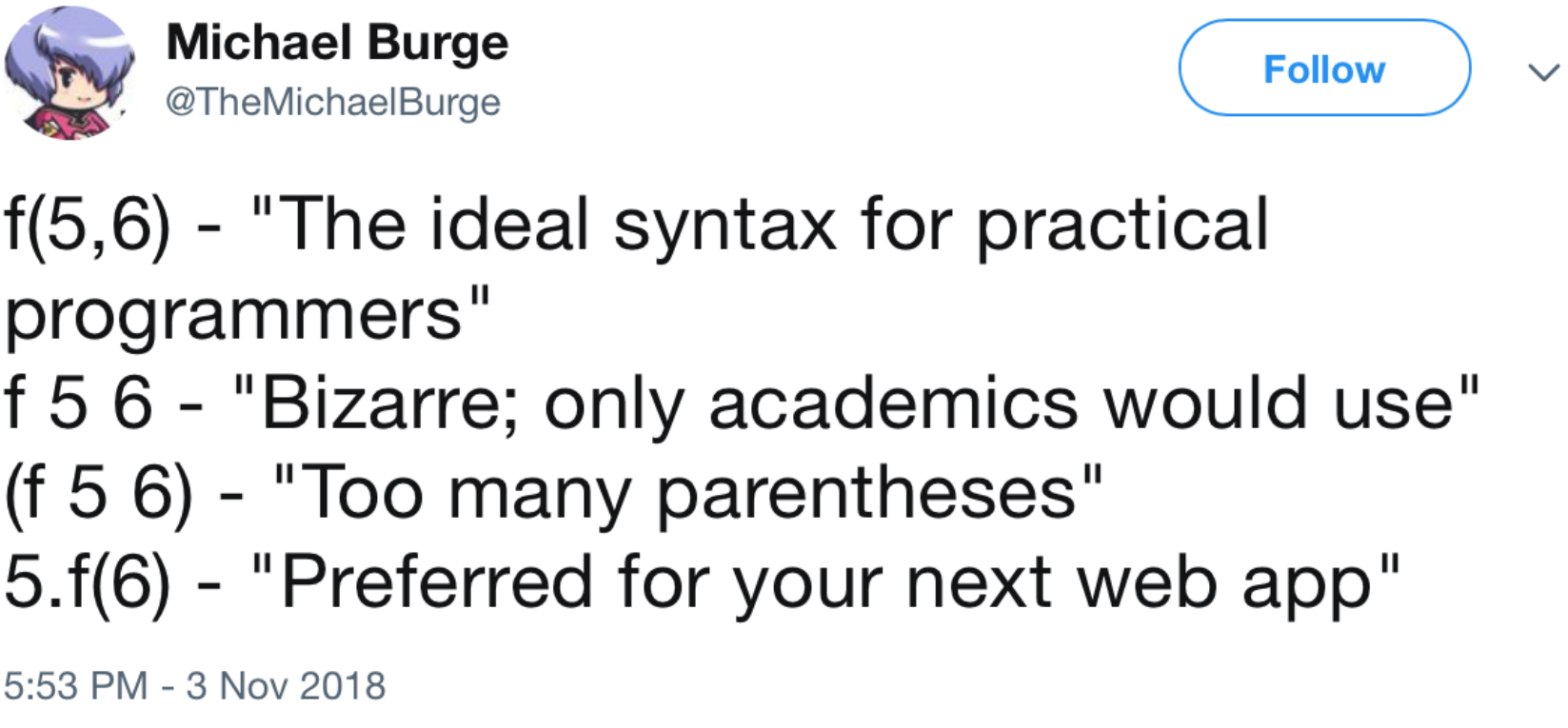
quilc and Common Lisp

Common Lisp: Personal experience

- No opportunity to be skeptical at the start.
- Easy language to learn, except for the idioms.
- Right tool for the right job.†

Common Lisp: Personal experience

- No opportunity to be skeptical at the start.
- Easy language to learn, except for the idioms.
- Right tool for the right job.†



quilc in the wild

Physicists

Likes	Running benchmarks.
Dislikes	Optimizing compilers / attempts to hide infidelity.
	Respecting abstraction boundaries.
	Using enough qubits to need an addresser.

quilc in the wild

Physicists

Likes	Running benchmarks.
Dislikes	Optimizing compilers / attempts to hide infidelity. Respecting abstraction boundaries. Using enough qubits to need an addresser.

Applications engineers

Likes	Hand-optimization. Maximum chip use.
Dislikes	Software trampling their careful optimization.

quilc in the wild

Newbies

Likes	Simply-described quantum programs. Wasting spatial resources.
Dislikes	Hardware limitations.

quilc in the wild

Newbies

Likes Simply-described quantum programs.
Wasting spatial resources.

Dislikes Hardware limitations.

(Academic) compiler authors

Likes Writing papers.
Doing A/B comparisons.

Dislikes Being locked into a feature.
Being locked into several features.

quilc in the wild

Newbies

Likes Simply-described quantum programs.
Wasting spatial resources.

Dislikes Hardware limitations.

(Academic) compiler authors

Likes Writing papers.
Doing A/B comparisons.

Dislikes Being locked into a feature.
Being locked into several features.

Important observation

None of these are the supposed default of "advantage-class algorithms running on advantage-class Rigetti hardware".

quilc in the future

Interoperability: ECL and C-style bindings.

Pulse-level control: quilc to extend downward: everything but the back-most piece of the compiler backend.

Higher-level constructs: Hamiltonians instead of circuits, projection operators instead of σ_z–basis MEASUREs, automatic mechanisms to trade more resources for increased connectivity / stability / ....

"Exotic" gatesets: $\mathrm{XY}(\theta)$, Google's goofy gate, ...

Thank you!!

http://github.com/rigetti/quilc
http://rigetti.com
http://chromotopy.org