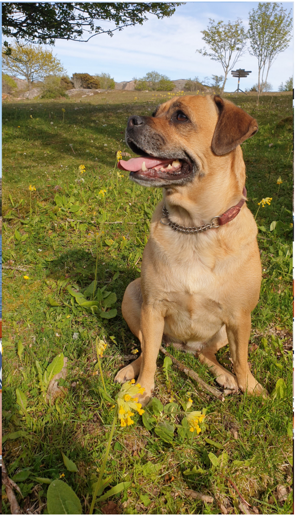
Signe

High-level functional quantum programming

Nicklas Botö   Fabian Forslund

December 1, 2021

Signe, the dog



Introduction

- Signe: A functional quantum programming language making use of quantum conditionals.

Introduction

- Signe: A functional quantum programming language making use of quantum conditionals.
- Compiles to a circuit representation.

Introduction

- Signe: A functional quantum programming language making use of quantum conditionals.
- Compiles to a circuit representation.
- Allows contraction (duplication*) and weakening (discarding).

Quantum programming languages

- Circuit languages: Used for describing quantum circuits, working directly with gates. Usually low abstraction.
  - OpenQASM
  - Qiskit
  - funQ

Quantum programming languages

- Circuit languages: Used for describing quantum circuits, working directly with gates. Usually low abstraction.
  - OpenQASM
  - Qiskit
  - funQ
- High-level languages: Features high levels of abstraction. Often implements constructs from classical programming.
  - Silq
  - QML
  - Signe

The Language Signe

Example Syntax

Signe

```
succ x : int -> int
  := x + 1

succ 0 -- 1
```

C

```
int succ(int x) {
  return (x + 1);
}

succ(0) // 1
```

Haskell

```
succ :: Int -> Int
succ x = x + 1

succ 0 -- 1
```

Hadamard

```
~+ : qubit := ~0 + ~1
~- : qubit := ~0 - ~1

H q : qubit -> qubit
    := if° q
         then ~-
         else ~+
```

CNOT

```
X q : qubit -> qubit
    := if° q
        then ~0
        else ~1

CX c t : qubit -> qubit -> qubit * qubit
    := if° c
        then (~1, X t)
        else (~0,   t)
```

S-gate

```
~i : qubit := i * ~1

S q : qubit -> qubit
    := if° q
        then ~i
        else ~0
```

T-gate

```
~e : qubit := e^(iπ/4) * ~1

T q : qubit -> qubit
    := if° q
        then ~e
        else ~0
```

Syntax

$$
\begin{array}{lrl}
Term & M, N, P & ::= x \mid \lambda x.M \mid M N \mid \text{if}^{\circ} P \text{ then } M \text{ else } N \\
 & & \mid \text{if } P \text{ then } M \text{ else } N \mid M + N \mid M - N \\
 & & \mid \kappa * M \mid \langle M, N \rangle \mid |0\rangle \mid |1\rangle \mid \text{let } x = M \text{ in } N
\end{array}
$$

$$
\begin{array}{lrl}
Mono & \tau, \varphi & ::= \alpha \mid \tau \to \varphi \mid \tau \otimes \varphi \mid \mathcal{Q}
\end{array}
$$

$$
\begin{array}{lrl}
Type & \sigma & ::= \tau \mid \forall \alpha . \sigma
\end{array}
$$

$$
\begin{array}{lrl}
Program & P & ::= f x : \sigma := M \lrcorner P \mid \mathbf{eof}
\end{array}
$$

$$
f, x, \alpha \in String \quad \kappa \in \mathbb{C} \quad \tau \in Mono \quad \sigma \in Type \quad M \in Term
$$

Compilation

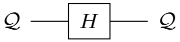
Signe

```
H q : qubit -> qubit
    := if° q
       then ~-
       else ~+
```

Internal

```
Ser [
 Par [Ser [Perm [0]]],
 Perm [0],
 Par [Perm [0]],
 Perm [0],
 Perm [0],
 Ser [Par [
   Gate (1/√2, 1/√2)
        (1/√2, −1/√2)
   ]
   , Perm [0]
  ]
]
```

Circuit



$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

Deutsch's algorithm

```
Deutsch oracle
  := let (x,_) = oracle (~+, ~-)
         h     = if° x then ~- else ~+
         (r,_) = (h,h)
           in r
```

Deutsch's algorithm

```
Deutsch oracle
  := let (x,_) = oracle (~+, ~-)
         h     = if° x then ~- else ~+
         (r,_) = (h,h)
           in r

Deutsch := measure ∘ H ∘ fst ∘ ap (~+, ~-)
```

Deutsch's algorithm

```
ap x f : ∀ a b . a -> (a -> b) -> b
  := f x

fst (x,y) : ∀ a b . (a * b) -> a
  := x

measure x : qubit -> qubit
  := let (r,_) = (x,x) in r
```

Type System

Overview

- The type system(s) ensures that the program follows a set of predefined rules.
- The compiler utilizes both a static (before code generation) and a dynamic (during code generation) system.
- The static type checker implements a Hindley-Milner type system.
- The dynamic type system implements a strict linear type system.

Hindley-Milner type checker

- Can derive the most general type for any expression.
- Allows types to be omitted from functions.
- Enables the use of polymorphic types (which give generic functions).

```
id x := x -- inferred type: ∀ a . a -> a
```

Hindley-Milner type checker

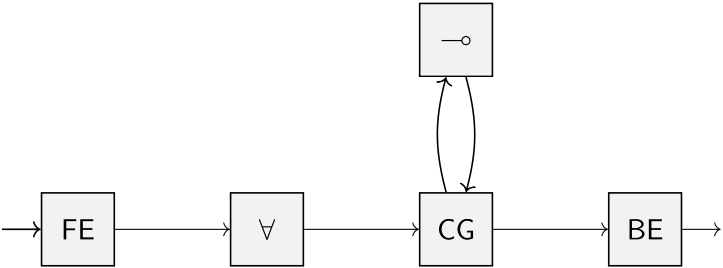
```
-- Polymorphic types
id x : ∀ a . a -> a := x

-- Monomorphic types
idQ x : qubit -> qubit := x
idQQ x : (qubit * qubit) -> qubit * qubit := x
idFQ x : (qubit -> qubit) -> qubit -> qubit := x
ifFQQ x : (qubit -> qubit * qubit) -> qubit -> qubit * qubit
 := x
```

$\vdots$

Linear type checker

- Imposes orthogonality constraints on certain expressions.
- Controls contraction and weakening and generates appropriate circuits according to the typing rules.
- Keeps track of the quantum register sizes.

Architecture



Finite Quantum Computations

The **FQC** category

- FQC is a category of finite quantum computations. Finite meaning we can't construct infinite types.

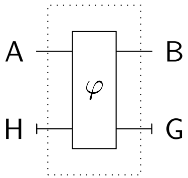
The **FQC** category

- FQC is a category of finite quantum computations. Finite meaning we can't construct infinite types.
- Signe programs are interpreted as FQC morphisms which serve as intermediate representations in compiling the language.

The **FQC** category

- FQC is a category of finite quantum computations. Finite meaning we can't construct infinite types.
- Signe programs are interpreted as FQC morphisms which serve as intermediate representations in compiling the language.
- The FQC category gives a convenient way of representing Signe programs (circuits).

The **FQC** category

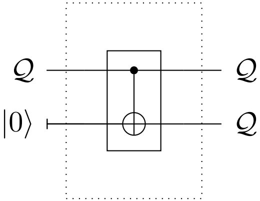


$$(H, G, \varphi) \in \mathbf{FQC}\ A\ B$$

$$|H| + |A| = |G| + |B|$$

$$\varphi \in A \otimes H \multimap B \otimes G$$

The **FQC** category



$$(\mathcal{Q}, \emptyset, \mathsf{CNOT}) \in \mathbf{FQC}\ \mathcal{Q}\ (\mathcal{Q} \otimes \mathcal{Q})$$

The **FQC** category

Canonical:

$$(H, G, \varphi) \in \mathbf{FQC}\ A\ B$$

The **FQC** category

Canonical:

$$(H, G, \varphi) \in \mathbf{FQC}\ A\ B$$

Strict:

$$(H, \emptyset, \varphi) \in \mathbf{FQC}^\circ\ A\ B$$

The **FQC** category

Canonical:

$$(H, G, \varphi) \in \mathbf{FQC}\ A\ B$$

Strict:

$$(H, \emptyset, \varphi) \in \mathbf{FQC}^{\circ}\ A\ B$$

Executable:

$$(H^{>0}, G, \varphi) \in \mathbf{FQC}^{\Downarrow}\ \emptyset\ B$$

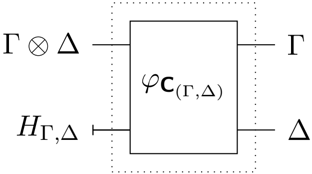
Contraction

- Practically means duplication of variables
- Is handled semantically not to violate the no-cloning property
- Allows for a *natural* programming style

```
copy x : ∀ a . a -> a * a
  := (x, x)
```

Operational semantics of contraction

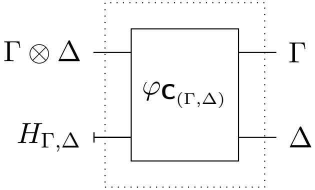
Contexts



$$\mathbf{C}_{(\Gamma,\Delta)} \in \mathbf{FQC}^{\circ} \left([\![\Gamma \otimes \Delta]\!] \right) \left([\![\Gamma]\!] \otimes [\![\Delta]\!] \right)$$

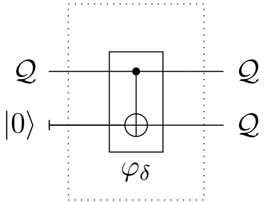
Operational semantics of contraction

Contexts



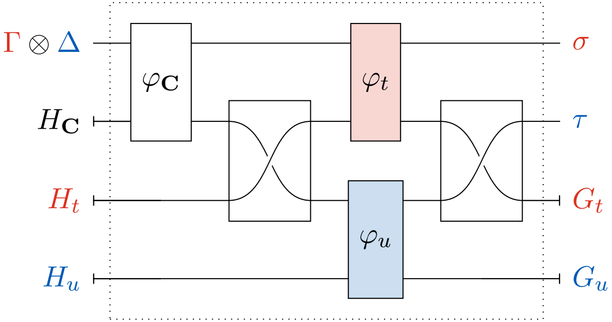
$$\mathbf{C}_{(\Gamma,\Delta)} \in \mathbf{FQC}^{\circ}\ (\llbracket \Gamma \otimes \Delta \rrbracket)\ (\llbracket \Gamma \rrbracket \otimes \llbracket \Delta \rrbracket)$$

Terms



$$\delta \in \mathbf{FQC}^{\circ}\ \mathcal{Q}\ (\mathcal{Q} \otimes \mathcal{Q})$$

Operational semantics of tuples



$$\langle t, u \rangle \in \mathbf{FQC}^{a \sqcap b} \; [\![\Gamma \otimes \Delta]\!] \; ([\![\sigma]\!] \otimes [\![\tau]\!])$$

$$t \in \mathbf{FQC}^{a} \; [\![\Gamma]\!] \; [\![\sigma]\!]$$

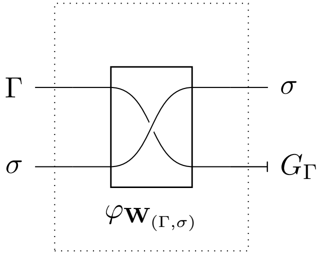
$$u \in \mathbf{FQC}^{b} \; [\![\Delta]\!] \; [\![\tau]\!]$$

Weakening

- Discarding of variables
- Carefully managed semantically by the compiler
- Greatly increases ease of use

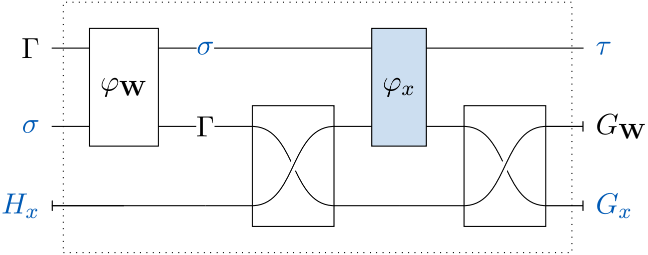
```
first (x,y) : ∀ a . a * a -> a
  := x
```

Operational semantics of weakening



$$\mathbf{W}_{(\Gamma,\sigma)} \in \mathbf{FQC}\left(\llbracket\Gamma\rrbracket \otimes \llbracket\sigma\rrbracket\right)\llbracket\sigma\rrbracket$$

Operational semantics of variables



$$x^{\mathbf{dom}\Gamma} \in \mathbf{FQC}\ (\llbracket \Gamma \rrbracket \otimes \llbracket \sigma \rrbracket)\ \llbracket \tau \rrbracket$$

$$x \in \mathbf{FQC}^a\ \llbracket \sigma \rrbracket\ \llbracket \tau \rrbracket$$

Comparison of conditionals

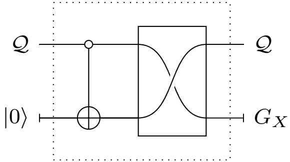
Code

Typing rule

Circuit

```
X q : qubit -> qubit
    := if q
       then ~0
       else ~1
```

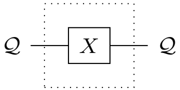
$$\frac{\Gamma \vdash c : \mathcal{Q} \qquad \Delta \vdash t, u : \sigma}{\Gamma \otimes \Delta \vdash \textsf{if } c \textsf{ then } t \textsf{ else } u : \sigma}$$



$X \in \mathbf{FQC}\ \mathcal{Q}\ \mathcal{Q}$

```
X° q : qubit -> qubit
     := if° q
        then ~0
        else ~1
```

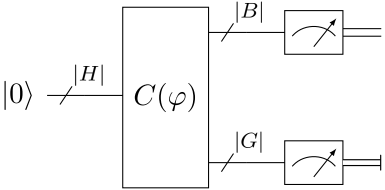
$$\frac{\begin{array}{c}\Gamma \vdash^{\circ} c : \mathcal{Q} \\ \Delta \vdash^{\circ} t, u : \sigma \qquad t \perp u\end{array}}{\Gamma \otimes \Delta \vdash^{\circ} \textsf{if}^{\circ} c \textsf{ then } t \textsf{ else } u : \sigma}$$



$X^{\circ} \in \mathbf{FQC}^{\circ}\ \mathcal{Q}\ \mathcal{Q}$

Circuit generation of executable morphisms

$$(H, G, \varphi) \in \mathbf{FQC}^{\Downarrow} \; B \quad \xRightarrow{\textit{generate}}$$



Summary

- Signe is a language making use of quantum conditionals that are circuit-compilable.

Summary

- Signe is a language making use of quantum conditionals that are circuit-compilable.
- Makes use of the FQC category as an intermediate representation of quantum programs.

Summary

- Signe is a language making use of quantum conditionals that are circuit-compilable.
- Makes use of the FQC category as an intermediate representation of quantum programs.
- Integrates both static and dynamic type checking to allow more powerful programming.

Summary

- Signe is a language making use of quantum conditionals that are circuit-compilable.
- Makes use of the FQC category as an intermediate representation of quantum programs.
- Integrates both static and dynamic type checking to allow more powerful programming.
- Duplication of qubits represented as contraction. Discarding represented as weakening. Both controlled semantically.

Future work

- Utilize the full power of polymorphic types in compilation.

Future work

- Utilize the full power of polymorphic types in compilation.
- Add support for (limited) quantum recursion.

Future work

- Utilize the full power of polymorphic types in compilation.
- Add support for (limited) quantum recursion.
- Adding classical datatypes (ints, lists, etc.).

Future work

- Utilize the full power of polymorphic types in compilation.
- Add support for (limited) quantum recursion.
- Adding classical datatypes (ints, lists, etc.).
- Add support for classical primitive recursion using countable datatypes (natural numbers).

Future work

- Utilize the full power of polymorphic types in compilation.
- Add support for (limited) quantum recursion.
- Adding classical datatypes (ints, lists, etc.).
- Add support for classical primitive recursion using countable datatypes (natural numbers).
- Translation to circuit-representing language (e.g QWIRE, SQIR).

Future work

- Utilize the full power of polymorphic types in compilation.
- Add support for (limited) quantum recursion.
- Adding classical datatypes (ints, lists, etc.).
- Add support for classical primitive recursion using countable datatypes (natural numbers).
- Translation to circuit-representing language (e.g QWIRE, SQIR).
- Use existing tools for circuit optimization and verification.

Future work

- Utilize the full power of polymorphic types in compilation.
- Add support for (limited) quantum recursion.
- Adding classical datatypes (ints, lists, etc.).
- Add support for classical primitive recursion using countable datatypes (natural numbers).
- Translation to circuit-representing language (e.g QWIRE, SQIR).
- Use existing tools for circuit optimization and verification.
- Run on actual quantum computer (perhaps IBM or Chalmers (hopefully!)).