Type system for automated generation of reversible circuits (abstract)

Benoît Valiron

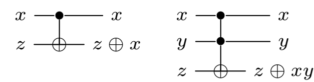
PPS, UMR 7126, Université Paris Diderot,
Sorbonne Paris Cité, F-75205 Paris, France,
benoit.valiron@monoidal.net

Summary of the abstract

Reversible circuits are boolean circuits made of reversible elementary gates. Such circuits are particularly useful in the context of quantum computation as quantum oracles. A quantum oracle is the circuit representation of a conventional, classical function. In this abstract, we aim at a simple method for generating reversible circuits out of a higher-order functional programming language. We expose the drawbacks of this approach and discuss how a type system could solve some of these problems.

Introduction. Reversible circuits are special kinds of boolean circuits, where wires are horizontal and parallel, and where elementary operations (called *gates*) are reversible. Wires host bits flowing from left to right, all at the same pace. When some bits meet a gate, the corresponding operation is applied on the wires attached to the gate. The overall circuit is reversible by making the bits flow backward.

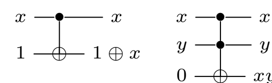
The gates we consider here are multi-controlled-not gates. A not gate is simply $-\oplus-$. The corresponding operation is the flip of the value flowing in the wire. The controlled-not gate, also denoted CNOT, is the first gate on the right, while the doubly-controlled gate, also denoted Toffoli is the second. A controlled-not flips the bit only when the controlling wires all have value "true" (or 1).



The main use we have in mind for reversible circuits in this abstract is quantum computation. In quantum computation, reversible circuits are mostly used as oracle: the *description* of the problem to solve. Usually, this description is given as a classical, conventional algorithm. This abstract is concerned with the design of reversible circuits derived from such algorithms, when given as programs in a PCF-like language: the language we aim at is minimal but expressive enough to encode most algorithms found in quantum oracles. It should feature recursion, pairs, booleans and lists, and easily be extended with additional structures if needed. We are therefore interested in the *compilation* of a program into a reversible circuit. In this abstract, we present the type system for a small subset of the language, nonetheless enough to discuss an example of term where optimization can happen.

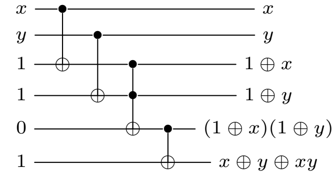
Reversible circuits from programs. Compiling a program into a reversible circuit is fundamentally different from compiling to a regular back-end: there is no notion of "loop", no real control flow, and all branches will be explored during the execution. In essence, a reversible circuit is the *trace* of all possible executions of a given program, and the compilation of a program essentially amounts to its evaluation.

As described by Landauer and Bennett, a conventional, classical algorithm computing a boolean function $f : \texttt{bit}^n \to \texttt{bit}^m$ can be mechanically transformed into a reversible circuit sending the triplet $(x, \vec{0}, \vec{0})$ to $(x, \mathrm{trace}, f(x))$. The input wires are not modified by the circuit, and the trace of all intermediate results are kept in garbage wires. For example, the Landauer embeddings for negation and conjunction are respectively shown in the right: they are simply CNOT and Toffoli gates, together with some auxiliary wires initialized to some constant boolean value 0 or 1.



Because of their particular structure, the Landauer embeddings of functions g and h can be composed to give a Landauer embedding of the composition $h \circ g$: the trace of the execution of both g and h are kept in garbage wires. For example, one can construct the embedding of the disjunction as the composition

of the embeddings for the negation and the conjunction, as shown on the right. The disjunction of x and y is $\neg(\neg x \wedge \neg y)$. The Landauer embeddings of negation and conjunction are composed in order to build all intermediary results, until we reach $x \oplus y \oplus xy$, which is indeed $x \vee y$. The two first CNOTs compute the inner negations, the Toffoli computes the conjunction, while the last CNOT makes the outer negation. As we said above, all intermediary results are retained.



One can use this technique to design the compilation of any lambda-calculus manipulating booleans into a reversible circuit. Therefore, if `not` and `and` have their obvious meanings, the two programs

$$x, y : \texttt{bit} \vdash \texttt{not}(\texttt{and}(\texttt{not}\ x)(\texttt{not}\ y)) : \texttt{bit} \tag{1}$$

$$x, y : \texttt{bit} \vdash (((\lambda z.\lambda t.\lambda s.s\,(\texttt{and}\ t\ z))(\texttt{not}\ x))(\texttt{not}\ y))\,\texttt{not} : \texttt{bit} \tag{2}$$

will generate the same circuit for disjunction shown above, regardless of the lambda abstractions involved in the term of Eq. (2).

Circuit optimization and type system. In the naive implementation of the disjunction in Eq. (1), the intermediate result of `and` cannot be reused later on as it is not stored in a variable. So instead of creating a new fresh variable, the outer `not` could simply reuse the wire of the intermediate result of `and` and update it with a $\oplus$ gate. In Eq. (1), this optimization can be realized at circuit-generation time by simple subterm inspection. In Eq. (2), it is not so clear: we claim that a specific type system is the right tool to untangle the terms and decide whether a particular wire is used several times. Terms can then be annotated with type information that is used at circuit-generation time to decide whether a fresh wire should be created or not.

Formally, the language consists of terms, types and sorts. For illustration, in this abstract we only give the subset needed to type Eqs (1) and (2). Terms are $M, N ::= x \mid \lambda x.M \mid MN \mid \texttt{and} \mid \texttt{not}$, types are $A, B ::= \alpha \mid A \to B$, while sorts are $\tau ::= 0 \mid 1 \mid +$. Sorts come with a transitive relation $<$ defined by $0 < 0 < 1 < + < +$. The type α ranges over an infinite alphabet, and it stands for a bit wire. So with respect to Eqs (1) and (2), we essentially have a type `bit` annotated with wires. A wire α of sort 0 is intuitively not used. A wire of sort 1 is used only once in the generated circuit while a wire of sort $+$ is used more than once. A typing judgement is of the form

$$\alpha_1 : \tau_1 \dots \alpha_n : \tau_n \mid x_1 : A_1 \dots x_m : A_m \vdash M : B,$$

and the typing rules enforce the property that wires can be shared amongst several variables but are themselves linear. If $|X|$ stands for the set of wires in X, then

$$\frac{|\Delta| = |A|}{\Delta \mid x : A \vdash x : A} \qquad \frac{\Delta \mid \Gamma, x : A \vdash M : B}{\Delta \mid \Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma_1 \mid \Delta \vdash N : A \quad \Gamma_2 \mid \Delta \vdash M : A \multimap B}{\Gamma_1 \cup \Gamma_2 \mid \Delta \vdash MN : B}$$

Finally, constants are typed with $\alpha : \tau_1, \beta : \tau_2 \vdash \texttt{not} : \alpha \to \beta$ when $\tau_1 \geq 1$, and $\alpha : \tau_1, \beta : \tau_2, \gamma : \tau_3 \vdash \texttt{and} : \alpha \to \beta \to \gamma$ when $\tau_1, \tau_2 \geq 1$. In the typing rule for application appears a union of contexts of sorts: $(\alpha_1 : \tau_1 \dots \alpha_n : \tau_n, \beta_1 : \sigma_1 \dots \beta_m : \sigma_m) \cup (\alpha_1 : \tau_1' \dots \alpha_n : \tau_n', \beta_{m+1} : \sigma_{m+1} \dots \beta_k : \sigma_k)$ is equal to $(\alpha_1 : \max(\tau_1, \tau_1') \dots \alpha_n : \max(\tau_n, \tau_n'), \beta_1 : \sigma_1 \dots \beta_k : \sigma_k)$ where $\max(\tau, \tau')$ is the smallest value σ such that $\tau, \tau' \leq \sigma$ and σ is strictly greater than one of τ and τ'. Since $(<)$ is reflexive on sort 0, if a wire α is neither "used" in M nor in N, then it is not used in MN. Again, $(<)$ is not reflexive on sort 1: a wire α used once both in M and N is used "many" times (aka "$+$") on MN.

Thus, during the compilation of a term, every `not` whose first argument is of sort 1 in the final term (i.e. the bottom of the typing derivation) can safely be compiled into a NOT gate, not generating a fresh wire. In the case of Eqs (1) and (2), one can type both terms with the pair of context and type $\dots, \alpha : 1, \beta : 1, \gamma : 0 \mid x : \alpha, y : \beta \vdash - : \gamma$ and the outer `not` has an argument of sort 1 in both cases, as desired.