Quantenflussdiagramme und die Quantenprogrammiersprache QPL

Andreas Schroeder

10. Juni 2003

Zusammenfassung

Die entwickelten Algorithmen für Quantencomputer sind aufgrund der heute üblicherweise verwendeten Formulierung durch Quantenturingmaschinen wenig verständlich. Was in der Welt der Quantencomputer fehlt, sind verständliche, leserliche und übersichtliche Formalismen, wie sie aus der klassischen Welt bekannt sind. Zwei verständlichere Formalismen aus der klassischen Welt sind die Programmiersprachen und Flussdiagramme. Hier wird der von Peter Selinger entwickelte Formalismus für Berechnungen auf der Basis von Quantencomputern dargestellt, der sich sowohl in Flussdiagrammen wie auch in einer Programmiersprache ausdrücken lässt.

1 Einleitung

In der Welt der Quantencomputer existieren zwei gängige Formalismen für die Notation von Algorithmen. Das sind einerseits Quantenturingmaschinen und andererseits Quantenschaltkreise. Die für Programmierer verständlicheren Ansätze, die Programmiersprachen sowie Flussdiagramme, werden bisher für Quantencomputer nur wenig verwendet. Die hier geschilderten Ansätze sind ein Versuch, den Weg für natürlichere Programmierweisen in der Quantenwelt zu ebnen. Denn in der klassischen Welt ist das Programmieren von Turingmaschinen oder von Quantengattern weitaus komplizierter als das Verwenden von höheren Programmiersprachen. Quantengatter und Turingmaschinen sind entsprechtend nur für die theorie der Informatik eine brauchbare Abstraktion. Für Anwendungen und die Formulierung von Algorithmen sind Programmiersprachen das gängige Werkzeug. Die programmierbaren Quantencomputer, sollten sie einmal wirklich realisiert werden, werden mit Programmiersprachen arbeiten so wie es klassische Computer tun.

Die hier vorgestellten Sprachen (Flussdiagramme und die eigentliche Sprache) verstehen sich als funktional insofern, dass sie Eingabewerte zu Ausgabewerten überführen, und sie sind zwei syntaktische Ausprägungen eines Formalismus. In den Formalismus können Funktionsaufrufe und Rekursion, sowie höhere Datentypen wie integer, quanten-integer (fester Länge), Bäume und Listen eingebettet werden. Im letzten Kapitel wird die Frage aufgegriffen, inwiefern die hier vorgestellten Sprachen funktional in ihrer Modellierung sind.

Die Flussdiagramme basieren auf zwei Operationen: unitäre Transformationen und Messungen. Eine Besonderheit des hier vorgestellten Formalismus

im Gegensatz zu Quantenturingmaschinen ist, dass Messungen während der Berechnung erlaubt sind. Genauso ist hier, anders als bei Quantenturingmaschinen, die Steuerungskomponente klassisch, nur die Daten sind quantenmechanisch. Es lassen sich keine Überlagerungen (Superposition) des Programmablaufes erreichen, wie es bei Quantenturimgmaschinen möglich ist. Analysiert man allerdings die gängigen Algorithmen die für Quantencomputer exisitieren, dann scheint die mögliche Überlagerung von Programmflüssen auch wenig Verwendung zu finden.

Ein Flussdiagramm oder Programm beschreibt intuitiv eine Transformation von Eingabewerten zu Ausgabewerten, die einer Transformation durch einen sogenannten *Superoperator* entspricht. Durch den Superoperator eines Flussdiagramms ist also bereits die gesamte denotationelle Semantik gegeben. Auf die Semantik der Flussdiagramme wird in dieser Arbeit allerdings nicht zentral eingegangen. Die Arbeit von Peter Selinger, auf der diese Ausarbeitung basiert, behandelt die Semantik der vorgestellten Formalismen viel ausführlicher und zeigt sogar, dass die Formalismen in dem Sinne vollständig sind, dass sich jeder Superoperator durch ein Flussdiagramm dargestellen lässt. Das heisst wiederum, dass durch den (konstruktiven) Beweis gezeigt wurde, dass eine gewisse (wenn auch nicht praktische) Normalform für Flussdiagramme existiert.

Die genaue Formalisierung der Semantik der Flussdiagramme kann im Rahmen dieser Arbeit aus Platzgründen nicht wiedergegeben werden, und selbiges gilt leider auch für den Beweis der Vollständigkeit der hier vorgestellten Sprachen. Für Interessierte und zur Vollständigkeit der Arbeit sei hier auf den Artikel von Peter Selinger verwiesen [Sel02].

2 Grundlagen

Der Formalismus basiert auf linearer Algebra, so dass Vektoren und Matrizen über komplexe Zahlen der Dreh- und Angelpunkt dieser Arbeit sein werden. Für die Beschreibung der Flussdiagramme werden einige besondere Schreibweisen sowie Konventionen über Matrizen verwendet, so dass es notwendig wird, auf diese Besonderheiten zuvor einzugehen und einige grundlegende Sätze über Matrizen wieder ins Gedächtnis zu rufen.

Seien A, B, C, D quadratische Matrizen gleicher Dimension. Dann ist mit

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$$

die "vertikale und horizontale Konkatenation" von A, B, C, D bezeichnet. Konjugiert komplexe Matrizen von Matrizen $A = (a_{ij}) \in \mathbb{C}^{n\times m}$ werden als $A^* = (\overline{a}_{ji}) \in \mathbb{C}^{m\times n}$ notiert.

Definition 1 (Spur und Norm) *Die* Spur *einer Matrix* $A = (a_{ij}) \in \mathbb{C}^{n\times n}$ *ist* $\mathsf{tr}(A) = \sum_i a_{ii}$*, die* Norm $|A|^2 = \mathsf{tr}(A^*A)$

Definition 2 (Unitäre, hermitesche und positive Matrizen) *Eine quadratische Matrix* $A = (a_{ij}) \in \mathbb{C}^{n\times n}$ *ist*

1. *unitär gdw.* $S^*S = I \iff S^* = S^{-1}$
2. *hermitesch gdw.* $A^* = A$
3. *positiv gdw.* $\forall |v\rangle \in \mathbb{C}^n \ \langle v|A|v\rangle \geq 0$

Lemma 1 (Aussagen über Matrizen)

1. *Falls S unitär ist und $A = SBS^*$, dann ist $tr(A) = tr(B)$ und $|A| = |B|$.*
2. *Falls $A \in \mathbb{C}^{n\times n}$ hermitesch ist, dann ist für alle $|u\rangle \in \mathbb{C}^n \langle u|A|u\rangle$ reell.*
3. *Hermitesche Matrizen sind diagonalisierbar und haben reelle Eigenwerte.*
4. *Eine Matrix $A \in \mathbb{C}^{n\times n}$ ist* rein (pure) *wenn es ein $|u\rangle \in \mathbb{C}^n$ gibt, so dass $A = |u\rangle\langle u|$ ist.*

Im späteren Abschnitten wird auf die hier genannten Eigenschaften von Matrizen zurückgegriffen werden. Nach den Grundlagen der linearen Algebra kommen nun spezielle Normierungskonventionen und Schreibweisen für Quantenbits.

2.1 Quanten-Bits

Ein Quantenbit ist ein quantenmechanisches System mit zwei möglichen Messergebnissen, die im folgenden mit $|\mathbf{0}\rangle$ und $|\mathbf{1}\rangle$ bezeichnet werden. Der Zustand des Systems wird durch eine $\mathbb{C}$-Linearkombination $|\psi\rangle = \alpha|\mathbf{0}\rangle + \beta|\mathbf{1}\rangle$ beschrieben. Mit $|\mathbf{0}\rangle$ und $|\mathbf{1}\rangle$ als Basis lässt sich der Zustand schreiben als

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

Entsprechend lässt sich ein System aus n Quantenbits beschreiben als $\mathbb{C}^{2^n}$ - Vektor über der Basis $\{|x\rangle \mid x \in \{\mathbf{0},\mathbf{1}\}^n\}$ darstellen. Dabei wird die folgende Indexierungskonvention verwendet:

Definition 3 (Indexierung der Basisvektoren) *Sei*

$$|v\rangle = \begin{pmatrix} \alpha_0 \\ \dots \\ \alpha_{2^n-1} \end{pmatrix} \in \mathbb{C}^{2^n}.$$

wobei $i \in [0, 2^n - 1], \mathsf{bin}_n(i)$ die Binärkodierung der Länge n von i. Dann ist $\alpha_i \in \mathbb{C}$ die Wahrscheinlichkeitamplitude für den Basisvektor $|\,\mathsf{bin}_n(i)\rangle$.

Für ein System aus drei Quantenbits und einen Vektor $|v\rangle \in \mathbb{C}^8$ und $i = 6$ zum Beispiel bezeichnet α_6 die Amplitude des Zustandes $|\mathbf{110}\rangle$.

Ebenso wird eine Normierung der Wahrscheinlichkeitsamplituden vorgenommen, aber nicht ganz wie üblich. Warum dies so ist, und wie die Amplituden normiert werden, wird später erläutert. Bisher reicht es anzunehmen, die Vektoren seien so normiert, dass für $|v\rangle \in \mathbb{C}^n$ gilt, dass $\sum_{i=1}^{n} |v_i|^2 = 1$ ist.

2.2 Unitäre Transformationen

Eines der beiden wichtigen Elemente der Quantenflussdiagramme ist die unitäre Transformation eines Zustandes. Eine unitäre Transformation eines 1-QBit systems ist durch eine unitäre Matrix $S \in \mathbb{C}^{2\times 2}$ gegeben als

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \mapsto S \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Entsprechend ist eine unitäre Transformation eines n-QBit Systems durch eine unitäre Matrix aus $\mathbb{C}^{2^n \times 2^n}$. Eine unitäre Transformation lässt sich auch auf ein einzelnes QBit eines Mehr-QBitsystems anwenden, und diese Anwendung lässt sich recht einfach mit Hilfe des Tensorprodukts formulieren. Die Anwendung einer unitären Matrix A auf das dritte QBit eines 4-QBitsystems ist zum Beispiel gegeben durch $A' = I \otimes I \otimes A \otimes I$.

2.3 Messungen und gemischte Zustände

Das zweite wichtige Element der Quantenflussdiagramme sind Messungen. Es sei daran erinnert, dass die hier vorgestellte Sprache Messungen vor dem Ende der Berechnungen erlaubt. Bei Messungen kollabieren die quantenmechanischen Zustände je nach Zustand und Messung vollständig oder partiell. Bei einem 2-QBit-System im Zustand $|\psi\rangle = \alpha_0|\mathbf{00}\rangle + \alpha_1|\mathbf{01}\rangle + \alpha_2|\mathbf{10}\rangle + \alpha_3|\mathbf{11}\rangle$ zum Beispiel kollabiert der Zustand bei Messung des ersten Bit und Ergebnis **0** zu $|\psi\prime\rangle = \alpha_0|\mathbf{00}\rangle + \alpha_1|\mathbf{01}\rangle$ mit Wahrscheinlichkeit $|\alpha_0|^2 + |\alpha_1|^2$. Der Zustand bleibt aber "quantenmechanisch" - es existiert nach wie vor die Superposition der Zustände $|\mathbf{00}\rangle$ und $|\mathbf{01}\rangle$, je nach Wahrscheinlichkeitsamplituden.

Nun kann die Normierung der Zustände angegangen werden. In der Physik ist es üblich, dass nach der Messung die Zustände wieder normiert werden. Hier wird dies nicht so gehandhabt. Der Zustand wird zu Beginn der Berechnung normiert, und die Normierung wird auch nach Messungen so belassen. Später wird klar werden, dass dies eine deutliche Vereinfachung für die Definition der Semantik bedeutet (siehe Merge- und Verzweigungsprimitive).

Nach der Messung ist der Zustand des Systems wieder durch einen einzigen Vektor gegeben, falls das Ergebnis der Messung dem Beobachter bekannt ist. Falls dem Beobachter jedoch nur bekannt ist, dass eine Messung durchgeführt wurde, das Messergebnis aber unbekannt ist, kommt die klassische Statistik zum Tragen. Hier kann der Beobachter nur formulieren, dass ein bestimmtes Messergebnis mit der entsprechenden Wahrscheinlichkeit aufgetreten sein könnte, und die Beschreibung des Systems ist dem Beobachter nur noch durch eine Linearkombination von quantenmechanischen, "reinen" Zuständen gegeben. Für diesen Sachverhalt wird hier die Notation $\sum_{i=1}^{m} \lambda_i\{|u_i\rangle\}$ eingeführt. Dabei beschreiben die $|u_i\rangle$ Vektoren, die nach der Messung möglichen Systemzustände, wobei ein bestimmter Zustand $|u_i\rangle$ mit Wahrscheinlichkeit $\lambda_i \in \mathbb{R}$ aufgetreten ist. Gemischte Zustände treten also auf, falls ermittelbare Information über den Zustand verloren geht. Falls ein System nur durch eine solche Linearkombination reiner Zustände beschrieben werden kann, heisst der Zustand des Systems *gemischt*. Unitäre Transformationen lassen sich auch auf gemischte Zustände

anwenden. Hier wird die Transformation auf jede reine Komponente des gemischten Zustandes angewendet.

2.4 Dichtematrizen

Für die einheitliche Beschreibung von reinen und gemischten Systemzuständen lassen sich Dichtematrizen verwenden. Eine *Dichtematrix* für einen Zustand $|\psi\rangle$ ist gegeben durch die Matrix $|\psi\rangle\langle\psi|$. Zum Beispiel ist die Dichtematrix für den Zustand $|\psi\rangle = \frac{1}{\sqrt{2}}|\mathbf{0}\rangle - \frac{1}{\sqrt{2}}|\mathbf{1}\rangle$ die Matrix

$$|\psi\rangle\langle\psi| = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

Die Dichtematrix eines gemischten Zustandes ergibt sich aus der mit Wahrscheinlichkeiten gewichteten Summe der Dichtematrizen der einzelnen reinen Zustände.

$$\{|\psi\rangle\} = \sum_{i=1}^{m} \lambda_i \{|u_i\rangle\} \text{ hat die Dichtematrix } \sum_{i=1}^{m} \lambda_i |u_i\rangle\langle u_i|$$

Die Normierung von Dichtematrizen überträgt sich aus der Normierung der Zustände. Eine Dichtematrix ist normiert, falls der reine Zustand normiert war. Entsprechendes gilt auch für Dichtematrizen von gemischten Zuständen.

In den reellen Diagonalelementen lassen sich bei normierten Dichtematrizen die Wahrscheinlichkeiten ablesen, mit denen das System bei einer vollständigen Messung in einem Zustand vorgefunden werden kann. Die Elemente außerhalb der Diagonalen können komplex sein und beschreiben die "Verquickung" der messbaren Zustände untereinander.

Definition 4 (Dichtematrix) *Eine Dichtematrix ist eine positive hermitesche Matrix A mit* $\mathsf{tr}(A) \leq 1$.

Dass die Spur kleiner als eins sein kann, hat dieselben Gründe wie die etwas unübliche Normierung der Wahrscheinlichkeitsamplituden.

Die zwei Operationen, unitäre Transformation und Messung, lassen sich natürlich auch auf Dichtematrizen anwenden. Eine unitäre Transformation bildet den Zustand $|\psi\rangle$ auf $S|\psi\rangle$ ab. Entsprechend wird die Dichtematrix $|\psi\rangle\langle\psi|$ durch unitäre Transformation auf die Dichtematrix[1] $S|\psi\rangle\langle\psi|S^*$ abgebildet.

Die Messung auf Dichtematrizen erfolgt nicht durch einfache Matrixtransformationen. Angenommen, es liege ein System mit dem folgenden reinen Zustand vor:

$$|u\rangle = \left(\frac{v}{w}\right), \quad |u\rangle\langle u| = \left(\begin{array}{c|c} vv^* & vw^* \\ \hline wv^* & ww^* \end{array}\right)$$

Wenn nun eine Messung des ersten QBits vorgenommen wird, liegen nach der Messung die Zustände

$$S_1 = \left(\begin{array}{c|c} vv^* & 0 \\ \hline 0 & 0 \end{array}\right) \text{ oder } S_2 = \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & ww^* \end{array}\right)$$

[1] Durch die Lemmas über Matrizen lässt sich leicht zeigen, dass das Ergebnis der Abbildung wieder eine Dichtematrix ist.

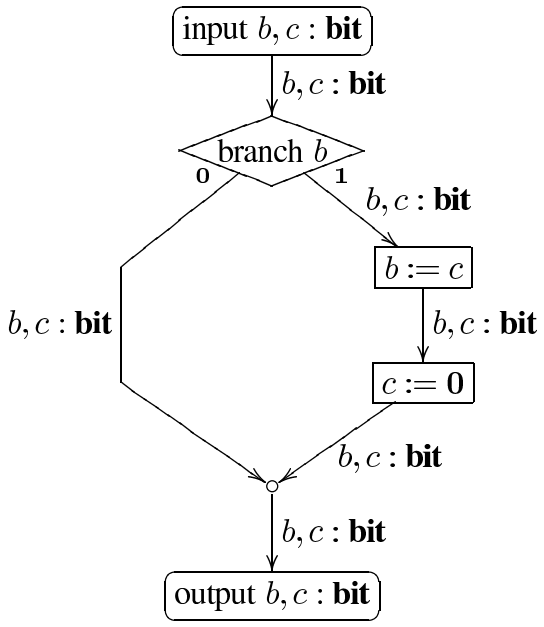


Abbildung 1: Beispiel

mit Wahrscheinlichkeit $|v|^2 = \mathsf{tr}(vv^*)$ bzw. $|w|^2 = \mathsf{tr}(ww^*)$ vor, falls die Matrix bzw. der Zustand u nach der Konvention normiert wurde. Der Vorgang der Messung erweitert sich, wie schon vorher die unitären Transformationen, linear auf Dichtematrizen gemischter Zustände. Falls nun das Ergebnis der Messung vergessen wird, können die Matrizen, da sie nicht umnormiert werden, einfach addiert werden:

$$S' = S_1 + S_2 = \left(\begin{array}{c|c} vv^* & 0 \\ \hline 0 & ww^* \end{array}\right)$$

Falls eine Messung des ersten QBits durchgeführt wird, ohne das Ergebnis abzulesen, ergibt sich also für einen beliebigen (gemischten oder reinen) Zustand

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right) \text{ der gemischte Zustand } \left(\begin{array}{c|c} A & 0 \\ \hline 0 & D \end{array}\right)$$

als Ergebiszustand. Man bemerke, dass das Messen und Vergessen anderer QBits außer des ersten QBits wegen der Indexierungskonvention schwieriger darzustellen ist, das Ergebnis aber immer dem Herauslöschen bestimmter Nichtdiagonalelemente entspricht.

Nach dieser theoretischen Einführung können nun die Flussdiagramme definiert werden, wobei zunächst die passende Formulierung klassischer Flussdiagramme untersucht wird, die es später erlauben wird, QBits einzubetten.

3 Klassische Flussdiagramme

Klassische Flussdiagramme beschreiben Abbildungen $\{0,1\}^n \to \{0,1\}^m$. Die Flussdiagramme selber sind zusammengesetzt aus primitiven Elementen, die

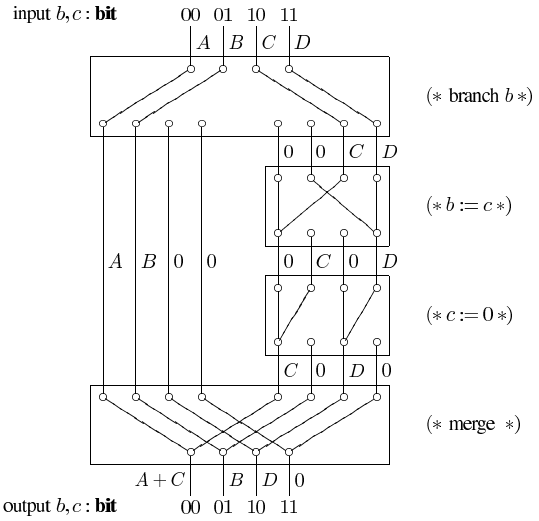


Abbildung 2: Variablenfreie Entfaltung

sich hintereinander schalten lassen. Ein Beispiel-Flussdiagramm ist in Abbildung 1 gegeben. Die Kanten, die primitive Elemente verbinden, sind mit getypten Variablenlisten annotiert. Die Typen garantieren, dass beim Zusammenführen von Zweigen die Typen aller Zweige übereinstimmen. Zunächst einmal wird nur ein einziger klassischer Typ **bit** eingeführt. Die Kanten werden also mit Tupeln von **bit**-Variablen annotiert.

Die Repräsentation der Flussdiagramme als Abbildungen deutet an, dass es möglich ist, ein klassisches $\{0,1\}^n \to \{0,1\}^m$ Flussdiagramm als 2^n unabhängige Kontrollflüsse zu expandieren: für jede mögliche Eingabe ein eigener Kontrollfluss. Durch die Möglichkeit, Variablen einzuführen und zu verwerfen, sowie im Flussdiagramm zu verzweigen oder Pfade zusammenzuführen, ist die Anzahl der Kontrollpfade nicht 2^n. Die Aussage ist genauer die folgende: Jede Kante, die mit n **bit**-getypten Variablen annotiert ist, kann zu 2^n Kanten expandiert werden, so dass für jede mögliche Variablenbelegung eine eigene Kante existiert. Die primitiven Elemente der Flussdiagramme permutieren die Kanten, je nachdem, welche Variablenbelegung durch die Ausführung des Elements auf eine vorherige Variablenbelegung entsteht. Es sei angemerkt, dass in einem derartig expandierten Flussdiagramm keine Variablen vorkommen, so dass diese Expansion der klassischen Flussdiagramme *variablenfreie Expansion* genannt wird. Abbildung 2 zeigt die Expansion des Flussdiagramms aus Abbildung 1.

Die variablenfreie Expansion der Flussdiagramme ist für die Semantik der Quantenflussdiagramme und für die Einbettung der QBits wichtig. Als Vorbereitung hierfür kann ein expandiertes Flussdiagramm probabilistisch gelesen werden: an jeder Eingangskante des expandierten Flussdiagramms sei eine Wahrscheinlichkeit gegeben. Dann kann die Wahrscheinlichkeit, dass eine Kante durchlaufen wird, für jede Kante durch die Permutierung und Zusammenführung der Kanten berechnet werden. In diesem Fall werden klassische Flussdiagramme zu Funktionen der Form $[0,1]^n \to [0,1]^m$ mit der zusätzlichen

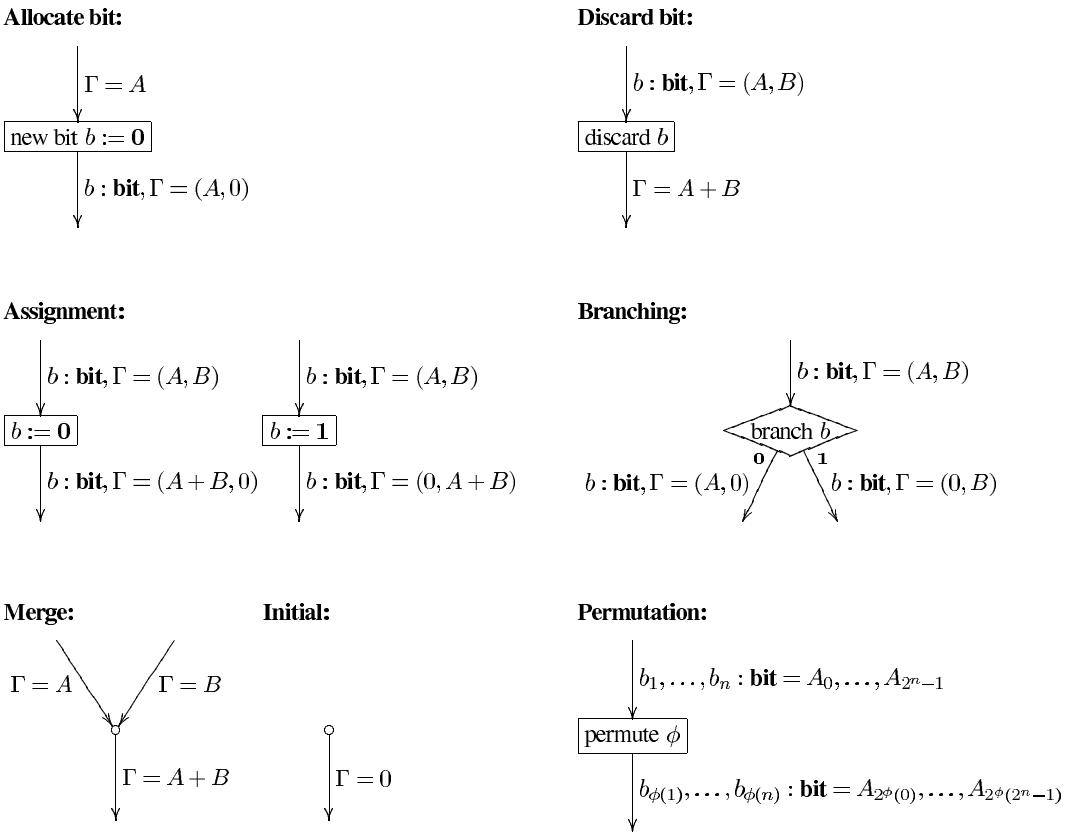


Abbildung 3: Elemente klassischer Flussdiagramme

Einschränkung, dass die Summe über die Eignabewerte kleiner oder gleich eins sein muss. Das Flussdiagramm aus Abbildung 2 berechnet also die Funktion $F(A, B, C, D) = F(A + C, B, D, 0)$. Aus diesen Überlegungen ist ersichtlich, wie im quantenmechanischen Fall die Wahrscheinlichkeiten durch ein klassisches Flussdiagramm propagiert werden, und dass eine Kante mit n klassischen Bits mit einem Tupel von 2^n Wahrscheinlichkeiten annotiert werden muss - für jede klassische Variablenbelegung eine quantenmechanische Wahrscheinlichkeit.

3.1 Primitive Elemente klassischer Flussdiagramme

Im Bild ist Γ ein beliebiger Typkontext. Die Beschriftung der Kanten besteht aus zwei Teilen: dem Label und der Annotation. Im Label wird der Typkontext geschrieben als Variablenliste mit Typen (abgekürzt in der Form a, b, c : **bit**). Die Annotation wird nach dem Gleichheitszeichen geschrieben und enthält die Wahrscheinlichkeit für das Durchlaufen der Kante für jede mögliche Belegung der klassischen Variablen. Über die Definition der Wirkung der primitiven Elemente auf Annotationen wird die Semantik von primitiven Elementen festgelegt. [2]

[2] Die Definitionen haben eine vereinfachte Form: es wird stets angenommen, dass das modifizierte bit das erste bit ist

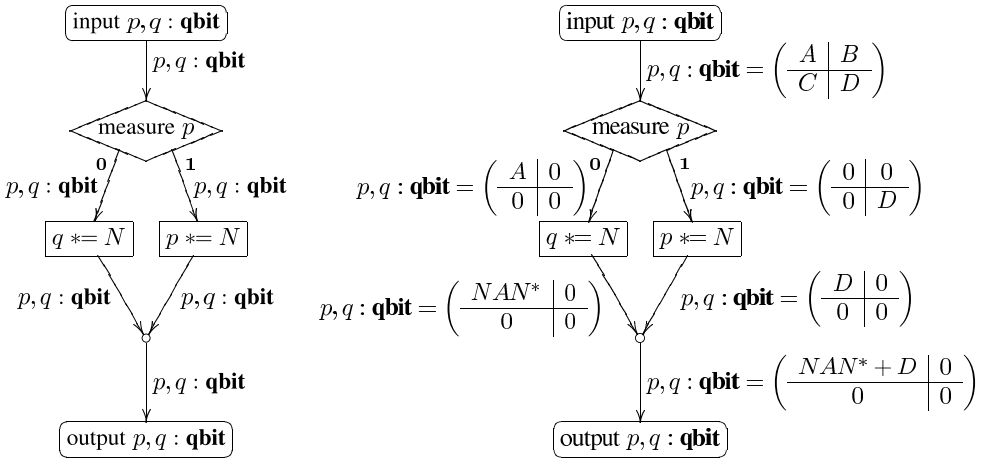


Abbildung 4: Beispiel eines Quantenflussdiagramms

Es gibt sieben primitive Elemente, die hier knapp beschrieben werden. “Allocate bit” reserviert ein neues Bit für das Programm und weist dieses bit einer Variablen zu. “Discard bit” gibt ein zugewiesenes bit wieder frei. “Assignment” weist einer Bitvariablen eines ihrer möglichen Werte zu. “Branching” verzweigt das Flussdiagramm je nach Belegung einer Bitvariablen. “Merge” führt mehrere Zweige zusammen (nicht notwendigerweise nur zwei) falls diese denselben Typ haben. “Initial” erzeugt eine neue, nicht erreichbare Kante. Dieses Konstrukt wird bei variablenfrei expandierten Flussdiagrammen benötigt. “Permutation” permutiert die Reihenfolge der Variablen im Typisierungskontext, ohne die Wertzuweisungen zu verändern.

Jedem primitiven Element wird hier direkt eine Semantik zugewiesen, die sie auch für Flussdiagramme mit qbits behalten werden. Die Definition über Tupel von Wahrscheinlichkeiten deutet hier schon auf zwei Dinge hin. Zum einen werden die Flussdiagramme für die Definition der Semantik zu variablenfreien Flussdiagrammen expandiert. Zum anderen werden wohl bald die einfachen Wahrscheinlichkeiten durch Dichtematrizen ersetzt.

4 Quanten-Flussdiagramme

Nun werden Flussdiagramme betrachtet, in welchen nur der Typ **qbit** erlaubt ist. In solchen Flussdiagrammen sind die beiden wichtigsten primitiven Elemente die unitäre Transfomration und die Messung. In dem Messprimitiv ist, um dem Messergebnis Rechnung zu tragen, eine Verzweigung eingebaut. Die Operation der unitären Transformation von qbits wird geschrieben als $p* = S$ für einstellige Gatter (z.B. N, not), $p, q \ *= \ S$ für zweistellige Gatter (z.B. N_c, controlled not) oder allgemein $p_i, \ldots, p_n \ *= \ S$ für n-stellige Gatter. Ein Beispiel-Flussdiagramm ist in Abbildung 4(a) gegeben, zusammen mit seiner Annotation. Analog zu einem klassischen Flussdiagramm kann ein quantenflussdiagramm als Funktion von Eingabe zu Ausgabe gesehen werden: Ganz

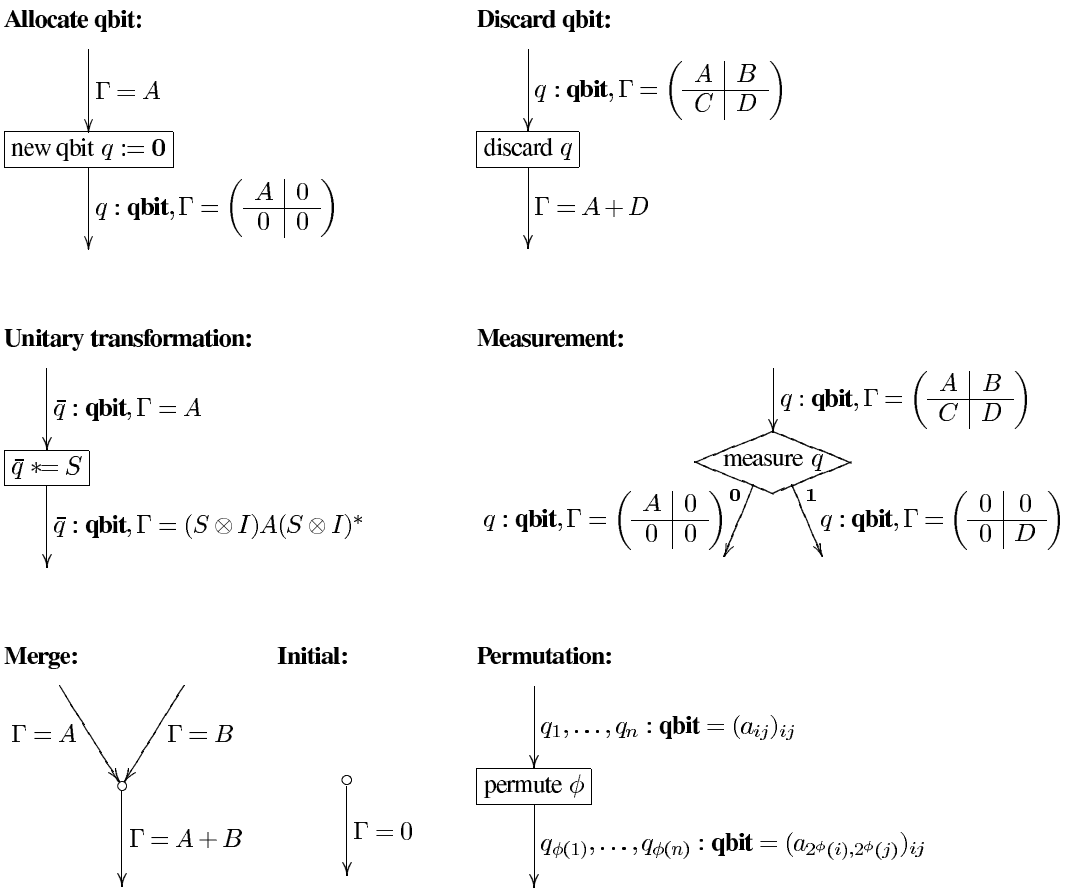


Abbildung 5: Elemente quantenmechanischer Flussdiagramme

allgemein kann die Eingabe ein gemischter Zustand sein, und die Ausgabe entsprechend auch ein gemischter Zustand. Das Flussdiagramm aus Abbildung 4 überführt die Eingabe $|\mathbf{00}\rangle$ (genauso wie die Eingabe $|\mathbf{11}\rangle$) zum reinen Zustand $|\mathbf{01}\rangle$. Die Eingabe des reinen Zustands $\frac{1}{\sqrt{2}}|\mathbf{00}\rangle + \frac{1}{\sqrt{2}}|\mathbf{10}\rangle$ führt aber zum gemischten Zustand $\frac{1}{2}\{|\mathbf{00}\rangle\} + \frac{1}{2}\{|\mathbf{01}\rangle\}$. Gemischte Zustände entstehen im Formalismus durch die Merge-Operationen. Diese Operationen entsprechen also dem Verlust von klassischer Information: dem Wissen darüber, welcher Zweig durchlaufen wurde. Das Messen und Vergessen eines QBits würde zum Beispiel in einem Flussdiagramm als Messung und anschließende Vereinigung der beiden Pfade dargestellt werden.

4.1 Primitive Elemente quantenmechanischer Flussdiagramme

Für **qbit**-getypte Variablen existieren im Formalismus wieder eine Reihe von Programmierprimitiven, deren eingehende und ausgehende Kanten wieder mit Typen und Wahrscheinlichkeiten annotiert werden. Die Semantik eines Elements ist gegeben durch seine Wirkung auf eine Dichtematrix, deren Größe von der Anzahl der dem Programm zugeteilten qbit-Speicherzellen abhängt.

Die Beschreibung der Semantik der einzelnen Elemente lässt sich aus den Beschreibungen der klassischen Programmprimitiven übernehmen. Messung und unitäre Transformation wurden zuvor ebenfalls beschrieben.

5 Zusammenführen quantenmechanischer Flussdiagramme mit klassischen

Die Kombination von quantenmechanischen und klassischen Flussdiagrammen wird so vorgenommen, wie es bereits nach der Behandlung der variablenfreien Expansion von klassischen Flussdiagrammen angedeutet wurde. Bei der Zusammenführung kann eine Kante, dessen Typ mit n bits und m qbits annotiert ist durch 2^n Kanten mit m qbits ersetzt werden. In der Semantik also wird eine einzelne Kante mit n bits und m qbits durch ein 2^n-Tupel von Dichtematrizen der Dimension $2^m \times 2^m$ $(A_0, \ldots, A_{2^n-1})$. Nun müssen also die Operationen der Quantenflussdiagramme auf Matrixtupel erweitert werden, da nun die eingehenden Kanten von Programmprimitiven nicht mehr mit einer einzigen Dichtematrix annotiert sind, sondern mit Tupeln von Dichtematrizen. Die Erweiterung wird als "lineare fortsetzung" der Definitionen vorgenommen. Spur, Konjugierte, Matrizenmultiplikation und Norm von Tupeln werden wie folgt definiert:

$$\begin{aligned}
\mathsf{tr}(A_0, \ldots, A_{2^n-1}) &:= \sum_{i=0}^{2^n-1} \mathsf{tr}(A_i) \\
(A_0, \ldots, A_{2^n-1})^* &:= (A_0^*, \ldots, A_{2^n-1}^*) \\
S(A_0, \ldots, A_{2^n-1})S^* &:= (SA_0S^*, \ldots, SA_{2^n-1}S^*) \\
|(A_0, \ldots, A_{2^n-1})|^2 &:= \sum_{i=0}^{2^n-1} |A_i|^2
\end{aligned}$$

Und die Komposition von Matrizen wird ebenfalls auf Tupel erweitert. Seien $A = (A_i)_i, B = (B_i)_i, C = (C_i)_i, D = (D_i)_i$ Tupel gleicher Länge von Matrizen gleicher Dimension. Dann wird mit

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$$

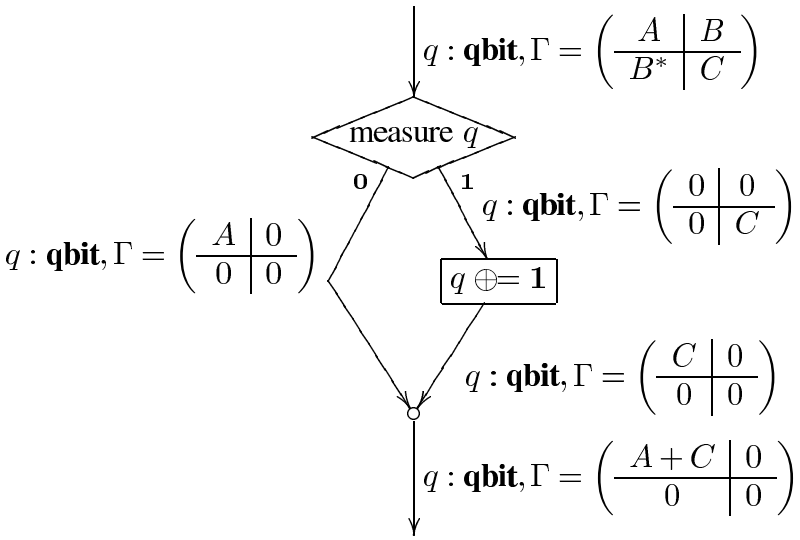
ein Tupel von Matrizen beschrieben, deren i-te Komponente die Matrix

$$\left(\begin{array}{c|c} A_i & B_i \\ \hline C_i & D_i \end{array}\right)$$

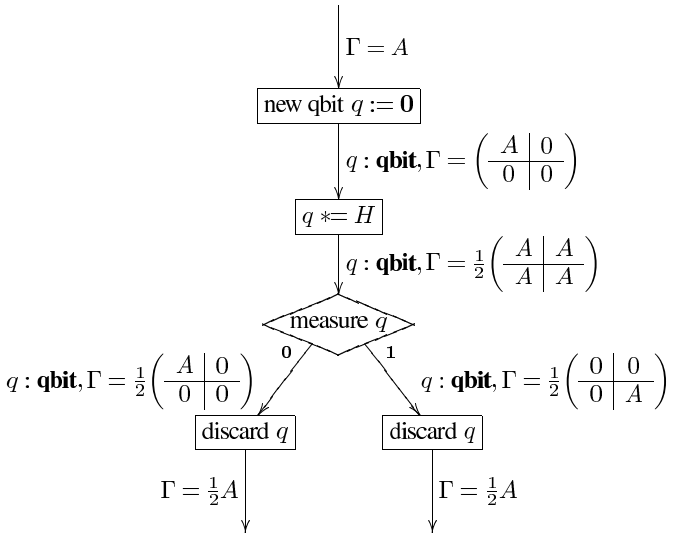
ist. Auch wird (A, B) als Tupelkonkatenation von A und B verstanden. Wird die Definition der Programmprimitiven mit diesen Erweiterungen gelesen, so kann die Semantik von kombinierten Flussdiagrammen von oben nach unten nach den gegebenen Regeln berechnet werden. Die Semantik ergibt sich eindeutig aus der Typannotation der eingehenden Kanten.

6 Beispiele

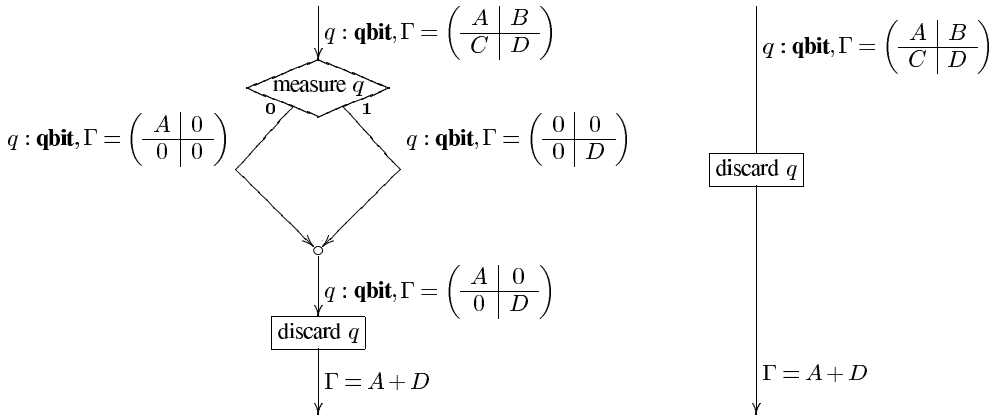
Zum besseren Verständnis der kombinierten Flussdiagramme eignen sich Beispiele am besten. Hier werden auch einige Abkürzungen für häufig benötigte Befehle gegeben.



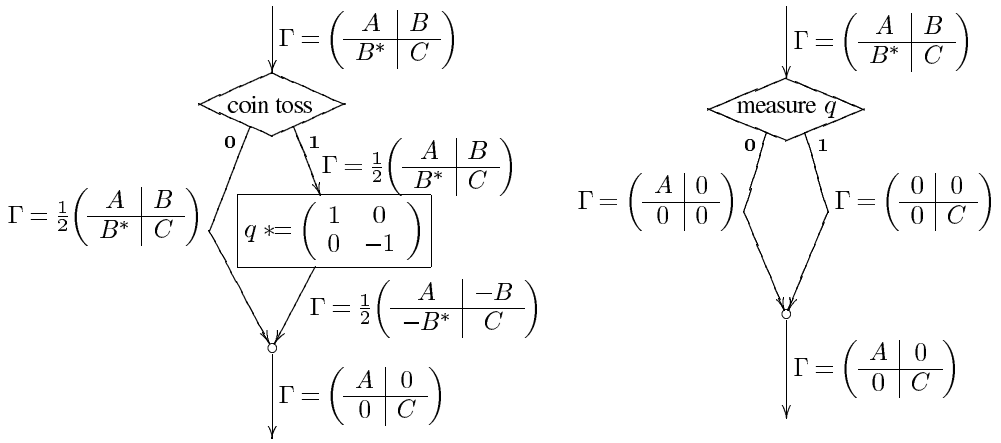
Dieses Beispiel zeigt, wie ein **qbit** auf 0 gesetzt werden kann. Dies wird hier durch eine Messung verwirklicht.



Das zweite Beispiel zeigt, wie ohne externen Zufallsgenerator ein Quantencomputer einen Münzwurf mit echtem Zufall durchführen kann. Bei klassischen Computern ist für solche Aufgaben stets ein externer Zufallsgenerator notwendig, oder eine Funktion, die Pseudozufallszahlen generiert.

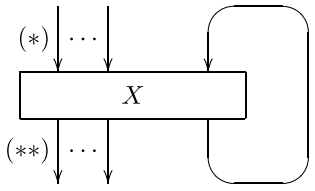


Im nächsten Beispiel sieht man, wie das Messen und anschließende Verwerfen des Quantenbits semantisch äquivalent zum direkten Verwerfen des Quantenbits ist; ein Verhalten, das sicher erwünscht ist. Dies ist auch der Fall, falls das Quantenbit mit anderen Quantenbits "verstrickt" (entangled) war.



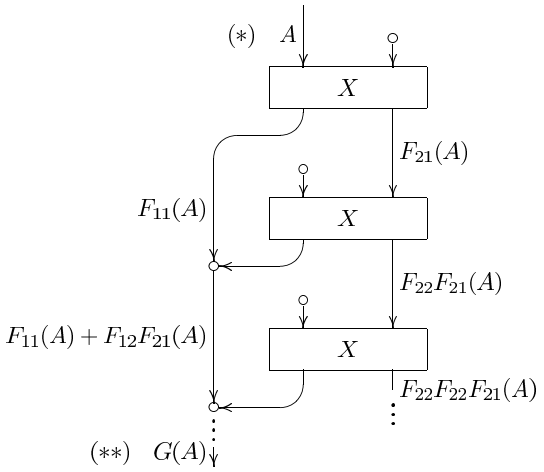
Das vierte Beispiel zeigt zwei unterschiedliche Programmfragmente, die semantisch äquivalent sind: Sie entsprechen dem Messen eines Qbits und Vergessen des Messergebnis. Beide Fragmente verhalten sich innerhalb eines größeren Programmkontext absolut identisch, was nicht unbeding der Fall ist, falls einem äußeren Beobachter das Ergebnis des Münzwurfs bekannt wird.

7 Schleifen



Nach den Beispielen kommen wir nun wieder zurück zu Erweiterungen des Formalismus. In klassischen Programmiersprachen gehört das Konzept der

Schleifen zu den grundlegenden Elementen, und in diesem Formalismus wird dieses Konstrukt ebenfalls eingeführt. Dies geschieht in einer allgemeineren Form als dies sonst üblich ist. Im Bild ist X ein beliebiges Programmfragment mit $n+1$ eingehenden und $m+1$ ausgehenden Kanten. Die Semantik des Schleifenkonstrukts ergibt sich auf andere Weise als im klassischen Fall. Dies hängt mit der probabilistischen Expansion der Flussdiagramme und mit den quantenmechanischen Zuständen zusammen. Bei einer klassischen Schleife wird der Rumpf der Schleife mehrere Male durchlaufen; sagen wir für einen bestimmten Programmlauf n Mal. Es macht hier keinen Sinn, in der Semantik mehr als n Durchläufe in irgendeiner Weise zu berücksichtigen. Anders ist es im quantenmechanischen Fall: Hier kann aufgrund der probabilistischen Natur der Quantenzustände keine feste Anzahl von Durchläufen einer Schleife angegeben werden. Es ist durchaus üblich, dass jeder der unendlich vielen Schleifendurchläufe einer Schleife zur Spur der ausgehenden Kanten einen Beitrag liefert. Da möglicherweise nie der gesamte Fluss wieder "abgeschöpft" wird, geht gegebenenfalls etwas vom Fluss verloren. Deshalb wurden Dichtematrizen mit einer Spur kleiner eins erlaubt.



Die Semantik einer Schleife wird also über die "unendliche Entfaltung" der Schleife definiert, die in obiger Abbildung angedeutet ist. Zur mathematischen Formalisierung lässt sich die durch das Flussdiagramm berechnete (lineare) Funktion $F(A,B) = (C,D)$ mit $A := (A_1,\ldots,A_n)$ und $C := (C_1,\ldots,C_m)$ schreiben als

$$\begin{aligned} F(A,B) &= F(A,0) + F(0,B) \\ &= F_1(A) + F_2(B) \\ &= (F_{11}(A) + F_{21}(B), F_{12}(A) + F_{22}(B)). \end{aligned}$$

Damit lässt sich die Semantik der unendlichen Entfaltung der Schleife, also die resultierenden Matrixtupel welche die Kanten (**) annotieren, schreiben als

$$G(A) = F_{11}(A) + \sum_{i=1}^{\infty} F_{12}(F_{22}^i(F_{21}(A))).$$

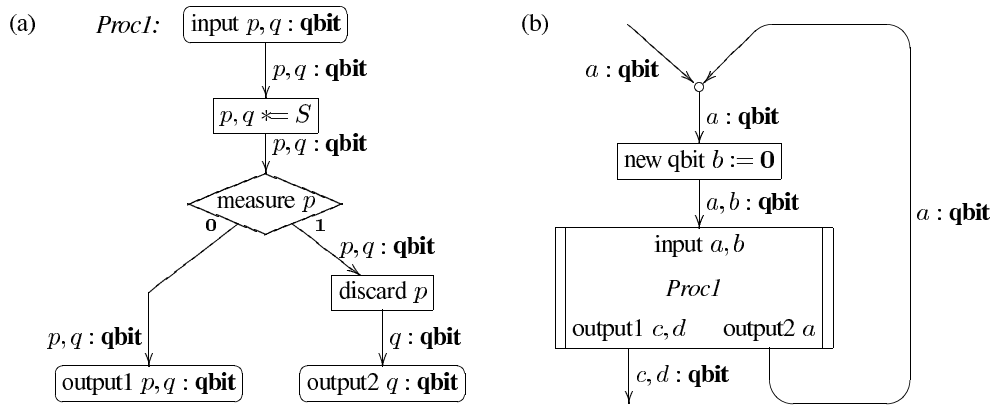


Abbildung 6: Prozedur und Prozeduraufruf

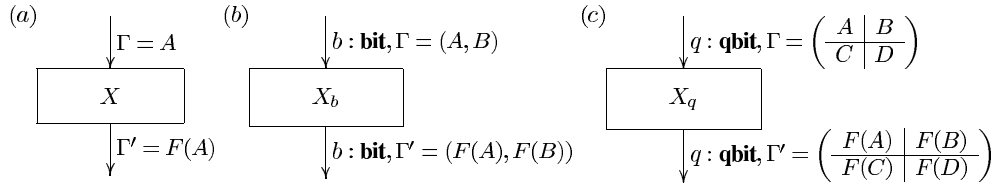


Abbildung 7: Weakening

8 Prozeduraufrufe und Weakening

Prozeduraufrufe sind im Formalismus einfach nur Flussdiagrammfragmente, die einen bestimmten Eingabetyp und Ausgabetyp. Um Prozeduraufrufe möglichst flexibel zu gestalten, wird es erlaubt, dass Prozeduren mehrere Einganskanten und Ausgangskanten besitzen. Die Eingabe- und Ausgabetypen einer Prozedur werden hier mit den Typkonstruktoren $\times$ und ; erzeugt, wobei durch ; die Typen unterschiedlicher Kanten getrennt werden. Zum Beispiel hat die Prozedur aus Abbildung 6 den Typ

$$Proc1 : \mathbf{qbit} \times \mathbf{qbit} \to \mathbf{qbit} \times \mathbf{qbit}; \mathbf{qbit}.$$

Bei genauerer Betrachtung der Bedeutung eines Prozeduraufrufs stellt sich die Frage: Was passiert, wenn der Typ der eingehenden Kanten um eine **bit**- oder **qbit**-Variable erweitert wird? Hat die Prozedur auf einen solchen Typkontext eine fundamental andere Semantik oder bleibt die Semantik gleich? Kann eine Prozedur aufgerufen werden, wenn zusätzliche (globale) Variablen erzeugt wurden?

Zur Beantwortung dieser Frage wird ein neues Konzept, das Weakening, eingeführt. Mit dessen Hilfe lässt sich die Semantik eines Programmfragments in einem größeren Typkontext als die vorgegebenen Typen eingehender und ausgehender Kanten festlegen. Klar ist, dass die Erweiterung des Typkontextes gleich für alle eingehenden und ausgehenden Kanten sein muss. Diese Situation ist in Abbildung 7 dargestellt. Abbildung 7(a) zeigt ein beliebiges Programmfragment

mit getypten Kanten. Nun lässt sich durch strukturelle Induktion über Flussdiagramme zeigen, dass falls $F : \overline{\Gamma} \to \overline{\Gamma'}$ die Semantik für das Programmfragment X ist, so ist die Semantik des Programmfragments bei Einführung einer **bit**-Variablen wie in Abbilduung 7(b) gegeben durch $G(A, B) = (F(A), F(B))$. Selbiges gilt für die Einführung einer **qbit**-Variablen (Abbildung 7(c)). Die Semantik des modifizierten Flussdiagramms ergibt sich hier als

$$G\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right) = \left(\begin{array}{c|c} F(A) & F(B) \\ \hline F(C) & F(D) \end{array}\right).$$

Die Semantik des Programmfragments lässt sich also linear auf beliebige, größere Typkontexte erweitern und ist eindeutig gegeben, falls F bekannt ist.

9 Rekursion

Das Weakening hat also gezeigt, dass es auch möglich ist, eine Prozedur in einem größeren Typkontext aufzurufen. Dadurch sind auch rekursive Prozeduraufrufe möglich, die Variablen allokieren und während des rekursiven Aufrufs allokiert halten. Abbildung 8 zeigt ein Beispiel eines rekursiven Prozeduraufrufs, wobei H_c die "controlled Hadamard" Matrix ist, und $q \oplus = r \;:=\; r, q * = N_c$, wobei N_c die "controlled not" Matrix ist.

Nun muss wieder die Semantik des rekursiven Prozeduraufrufs festgelegt werden. Für diese Berechnung wird wie folgt vorgegangen. Angenommen die Prozedur X sei rekursiv. Dann ist die berechnete Semantik für einen Durchlauf von X abhängig vom rekursiven Prozeduraufruf. Sei τ die berechnete "Semantikfunktion" für X. Falls die Semantik (eine lineare Abbildung von Tupeln von Dichtematrizen) des Prozeduraufrufs mit A gegeben wäre, dann ließe sich die Semantik von X schreiben als $\Phi(A)$. Nun ist das Verfahren zur Berechnung recht direkt gegeben, denn wie ließe sich die Semantik des Prozeduraufrufs von X berechnen als durch die Semantik von X selbst, Φ. Es wird also eine rekursive Folge definiert als

$$F_0 = 0, \;\; F_{i+1} = \Phi(F_i)$$

Schließlich lässt sich die Semantik des Prozeduraufrufs definieren als der Grenzwert dieser Folge,

$$G = \lim_{i \to \infty} F_i.$$

10 Die Programmiersprache QPL

Die in den letzten Abschnitten vorgestellten Flussdiagramme eignen sich zwar für eine übersichtliche graphische Darstellung; für das tatsächliche Programmieren sind sie allerdings nicht gut handhabbar. Für das alltägliche Programmieren wurde deshalb die Programmiersprache QPL erdacht, die eine textuelle Syntax anbietet. Diese wird im Folgenden beschrieben.

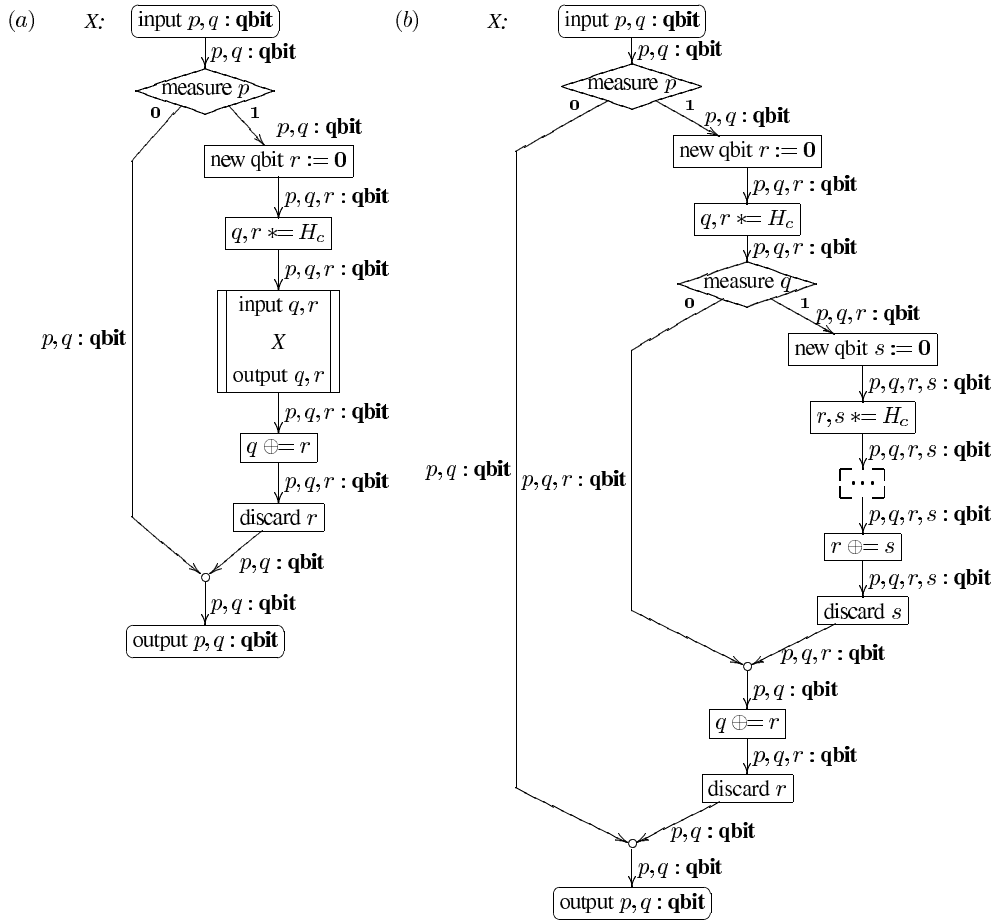


Abbildung 8: Rekursive Prozedur

10.1 Syntax von QPL

Zunächst einmal wird eine abzählbar unendlich große Menge an Variablen benötigt. Variablen werden mit eventuell indizierten Kleinbuchstaben notiert $x, y_4, z, p, q, \ldots$. Darüber hinaus wird eine ebenfalls abzählbar unendliche Menge an Prozedurvariablen benötigt; sie werden mit eventuel indizierten Großbuchstaben notiert $X, Y_2, \ldots$.

Die Typen von QPL werden aus den zwei Basistypen, **bit** und **qbit** zusammengesetzt. Hierbei gibt es drei Typsorten: den Typkontext, den Prozedurtyp, und den Prozedurkontext. Typkontexte definieren die Typen von allokierten Variablen und werden als $x_1 : t_1, \ldots, x_n : t_n$ geschrieben. Ein Prozedurtyp beschreibt den Typ einer Prozedur und wird als $t_1, \ldots, t_n \to s_1, \ldots, s_m$ geschrieben, wobei $s_i, t_j \in \{\mathbf{bit}, \mathbf{qbit}\}$. Ein Prozedurkontext beschreibt die Menge der definierten Prozeduren und wird als $X_1 : T_1, \ldots, X_n : T_n$ geschrieben, wobei T_i Prozedurtypen sind. Für Typkontexte wird im Folgenden stets Γ als Bezeichner verwendet, für Prozedurkontexte Π. Typerweiterungen von gegebenen Typen werden dann geschrieben als $x : t, \Gamma$ und $X : T, \Pi$ in der Annahme, dass x in Γ nicht vorkommt und X in Π ebenfalls nicht vorkommt, was bedeutet, dass der neue Typkontext bzw. Prozedurkontext wohldefiniert ist.

Die QPL-Terme, was soviel ist wie die abstrakte Syntax der erlaubten Programme, lassen sich durch folgende EBNF-Regeln beschreiben:

$$
\begin{aligned}
P ::= \quad & \textbf{new bit } b := \mathbf{0} \mid \textbf{new qbit } q := \mathbf{0} \mid \textbf{discard } x \\
& \mid b := \mathbf{0} \mid b := \mathbf{1} \mid q_1, \ldots, q_n * = S \\
& \mid \textbf{skip} \mid P_1 ; P_2 \\
& \mid \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \mid \textbf{measure } q \textbf{ then } P_1 \textbf{ else } P_2 \mid \textbf{while } b \textbf{ do } P \\
& \mid \textbf{proc } X : \Gamma \to \Gamma' \{P_1\} \textbf{ in } P_2 \mid y_1, \ldots, y_m = X(x_1, \ldots, x_n)
\end{aligned}
$$

proc $X : \Gamma \to \Gamma' \{P_1\}$**in** P_2 definiert eine neue Prozedur, deren Rumpf P_1 ist, und deren Skopus P_1 und P_2 ist. Darüber hinaus ist es sinnvoll, abkürzende Schreibweisen zu definieren, wie zum Beispiel $b := c$.

Bei genauerer Betrachtung der in QPL erlaubten Programme stellt man fest, dass die Programmiersprache nicht die Flexibilität der Flussdiagramme besitzt. QPL weist gegenüber den Flussdiagrammen drei Einschränkungen auf:

- Verzweigungen und Schleifen müssen korrekt verschachtelt werden, es ist z.B. nicht möglich, aus einer Schleife heraus zu verzweigen
- Mergeoperationen können nur im Zusammenhang mit Schleifen oder Verzweigungen auftreten
- Prozeduren haben nur eine eingehende und eine ausgehende Kante, wenn sie in den Flussdiagramm-Formalismus rücküberführt werden.

Dennoch ist die Programmiersprache QPL genauso mächtig wie die Flussdiagramme.

10.2 Typisierungsregeln

Da QPL eine streng typisierte Sprache ist, wird ein Werkzeug benötigt, um zur Kompilierzeit den Typ und die Gültigkeit eines Programms festzustellen.

Hierfür werden Typisierungsregeln verwendet, die in einem Typisierungskalkül angewendet werden. Die Regeln basieren auf Typaussagen, die von der folgenden Form sind:

$$\Pi \vdash \langle \Gamma \rangle P \langle \Gamma' \rangle$$

Die Bedeutung einer Typaussage ist, dass unter dem Prozedurkontext Π das Programm P ein wohlgeformtes Programm ist, das den Typkontext Γ in den Typkontext Γ' überführt.

Alle Typisierungsregeln für QPL sind in Abbildung 9 zu sehen. Die Typisierungsregeln erzwingen zum Beispiel auch, dass die **qbit**-Variablen $q_1, \ldots, q_n$, die einer unitären Transformation unterzogen werden, paarweise verschieden sind, oder dass die aktuellen Parameter einer Prozedur ebenfalls paarweise verschieden sind. Es ist aber erlaubt, dass Rückgabevariablen in der Menge der aktuellen Parameter enthalten sind(d.h. $y_i = x_j$ für ein i und j).

10.3 Semantik

Die Semantik der Programmiersprache QPL muss auch betrachtet werden. Die Semantik der Programmierprimitive der Sprache lässt sich über den Formalismus der Quantenflussdiagramme definieren. Für jedes Primitiv gibt es ein entsprechendes Flussdiagramm, dessen Semantik für die Sprache übernommen werden kann. Aus Platzgründen wird auf die Darstellung der "primitiven Flussdiagramme" verzichtet. Die Definition der entsprechenden Fragmente wird dem Leser überlassen.

10.4 Beispiele

Nun werden wieder einige Beispiele aufgeführt, die die Übersetzung von bereits vorgestellten Flussdiagrammen darstellen. Das erste Beispiel ist die Übersetzung des Quantenflussdiagramms aus Abbildung 4, das Einführungsbeispiel für Quantenflussdiagramme. Das zweite Beispiel zeigt die Implementierung einer Münzwurfprozedur in QPL, und das dritte Beispiel zeigt die Übersetzung der rekursiven Funktion X aus Abbildung 8. Dadurch, dass hier die abstrakte Syntax betrachtet wird, fehlt in der Darstellung Information über die Schachtelung von Programmteilen (d.h. Subterme). Zum Beispiel fehlt beim if-then-else-Primitiv eine Endklammerung, die das Parsen erschweren würde, falls diese abstrakte Syntax wirklich als textuelle Darstellung verwendet würde. Es sei also nochmals darauf hingewiesen, dass hier Terme als Beispiele gegeben werden, keine textuelle Darstellung. Die Termstruktur wird durch Einrückung eindeutig gemacht.

Beispiel 1

input p, q : **qbit**
measure p **then** $q *= N$ **else** $p *= N$
output p, q

(*newbit*)
$$\frac{}{\Pi \vdash \langle \Gamma \rangle \ \textbf{new bit}\ b := \textbf{0}\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*newqbit*)
$$\frac{}{\Pi \vdash \langle \Gamma \rangle \ \textbf{new qbit}\ q := \textbf{0}\ \langle q{:}\textbf{qbit}, \Gamma \rangle}$$

(*discard*)
$$\frac{}{\Pi \vdash \langle x{:}t, \Gamma \rangle \ \textbf{discard}\ x\ \langle \Gamma \rangle}$$

(*assign*$_0$)
$$\frac{}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ b := \textbf{0}\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*assign*$_1$)
$$\frac{}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ b := \textbf{1}\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*unitary*)
$$\frac{S \text{ is of arity } n}{\Pi \vdash \langle q_1{:}\textbf{qbit}, \ldots, q_n{:}\textbf{qbit}, \Gamma \rangle\ \bar{q} *= S\ \langle q_1{:}\textbf{qbit}, \ldots, q_n{:}\textbf{qbit}, \Gamma \rangle}$$

(*skip*)
$$\frac{}{\Pi \vdash \langle \Gamma \rangle\ \textbf{skip}\ \langle \Gamma \rangle}$$

(*compose*)
$$\frac{\Pi \vdash \langle \Gamma \rangle\ P\ \langle \Gamma' \rangle \qquad \Pi \vdash \langle \Gamma' \rangle\ Q\ \langle \Gamma'' \rangle}{\Pi \vdash \langle \Gamma \rangle\ P; Q\ \langle \Gamma'' \rangle}$$

(*if*)
$$\frac{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ P\ \langle \Gamma' \rangle \qquad \Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ Q\ \langle \Gamma' \rangle}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q\ \langle \Gamma' \rangle}$$

(*measure*)
$$\frac{\Pi \vdash \langle q{:}\textbf{qbit}, \Gamma \rangle\ P\ \langle \Gamma' \rangle \qquad \Pi \vdash \langle q{:}\textbf{qbit}, \Gamma \rangle\ Q\ \langle \Gamma' \rangle}{\Pi \vdash \langle q{:}\textbf{qbit}, \Gamma \rangle\ \textbf{measure}\ q\ \textbf{then}\ P\ \textbf{else}\ Q\ \langle \Gamma' \rangle}$$

(*while*)
$$\frac{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ P\ \langle b{:}\textbf{bit}, \Gamma \rangle}{\Pi \vdash \langle b{:}\textbf{bit}, \Gamma \rangle\ \textbf{while}\ b\ \textbf{do}\ P\ \langle b{:}\textbf{bit}, \Gamma \rangle}$$

(*proc*)
$$\frac{X{:}\bar{t} \to \bar{s}, \Pi \vdash \langle \bar{x}{:}\bar{t} \rangle\ P\ \langle \bar{y}{:}\bar{s} \rangle \qquad X{:}\bar{t} \to \bar{s}, \Pi \vdash \langle \Gamma \rangle\ Q\ \langle \Gamma' \rangle}{\Pi \vdash \langle \Gamma \rangle\ \textbf{proc}\ X : \bar{x}{:}\bar{t} \to \bar{y}{:}\bar{s}\ \{\ P\ \}\ \textbf{in}\ Q\ \langle \Gamma' \rangle}$$

(*call*)
$$\frac{}{X{:}\bar{t} \to \bar{s}, \Pi \vdash \langle \bar{x}{:}\bar{t}, \Gamma \rangle\ \bar{y} = X(\bar{x})\ \langle \bar{y}{:}\bar{s}, \Gamma \rangle}$$

(*permute*)
$$\frac{\Pi \vdash \langle \Gamma \rangle\ P\ \langle \Delta \rangle, \qquad \Pi', \Gamma', \Delta' \text{ permutations of } \Pi, \Gamma, \Delta}{\Pi' \vdash \langle \Gamma' \rangle\ P\ \langle \Delta' \rangle}$$

Abbildung 9: Typisierungsregeln

Beispiel 2

```
proc CoinToss :→ b : bit
{
  new qbit q := 0;
  q ∗ = H;
  new bit b := 0;
  measure q then skip else b := 1;
  discard q
}
in
  ...
  b := CoinToss();
  ...
```

Beispiel 3

```
proc X : p : qbit, q : qbit → p : qbit, q : qbit
{
  measure p then skip else
    new qbit r := 0;
    q, r ∗ = Hc;
    q, r = X(q, r);
    r, q ∗ = Nc;
    discard r
}
in
  ...
```

11 Erweiterungen des Typsystems

Für die Programmierung benötigt man sicherlich komfortablere Typen als nur **bit** und **qbit**. Zum Beispiel wäre es sinnvoll, Zahltypen oder strukturierte Typen zu haben. Für diese Erweiterungen werden zunächst einmal zwei Arten von Typkonstruktoren eingeführt: die Tupelbildung und die Summen, die eine Auswahl aus mehreren Typen ermöglicht. Zusätzlich wird eine Typkonstante **I** eingeführt, die dem "nil" aus den Listentypen in etwa entspricht. Wenn eine Variable vom Typ **I** ist, so trägt sie keine weitere Information als ihren Typ.

Tupel werden in QPL als $t_1 \otimes \ldots \otimes t_n$ geschrieben, falls $t_1, \ldots, t_n$ Typen sind. Für die Tupel werden zwei zusätliche Typisierungsregeln eingeführt, einerseits zur Tupeleinführung und andererseits zur Tupelbeseitigung:

$$(\textit{tuple}) \quad \frac{}{\Pi \vdash \langle x_1{:}t_1, \ldots, x_n{:}t_n, \Gamma\rangle\ x = (x_1, \ldots, x_n)\ \langle x : t_1 \otimes \ldots \otimes t_n, \Gamma\rangle}$$

$$(\textit{untuple}) \quad \frac{}{\Pi \vdash \langle x : t_1 \otimes \ldots \otimes t_n, \Gamma\rangle\ (x_1, \ldots, x_n) = x\ \langle x_1{:}t_1, \ldots, x_n{:}t_n, \Gamma\rangle}$$

Mit Hilfe von Tupeln lassen sich Typen entwerfen, die eine komfortable Zahlenverarbeitung ermöglichen.

Summen erlauben die Auswahl aus verschiedenen Typen und werden in QPL geschrieben als $t_1 \oplus \ldots \oplus t_n$, falls $t_1, \ldots, t_n$ Typen sind. Die Typisierungsregeln müssen hier wieder um zwei Regeln erweitert werden, nämlich für die Einführung eines Summentyps und für die Auswahl aus einem Summentyp (auch als "pattern matching" bekannt):

$$(\textit{inj}) \qquad \frac{}{\Pi \vdash \langle x{:}t_i, \Gamma\rangle\ y = \mathsf{in}_i\, x : t_1 \oplus \ldots \oplus t_n\ \langle y : t_1 \oplus \ldots \oplus t_n, \Gamma\rangle}$$

$$(\textit{case}) \qquad \frac{\Pi \vdash \langle x_1{:}t_1, \Gamma\rangle\ P_1\ \langle \Gamma'\rangle \qquad \ldots \qquad \Pi \vdash \langle x_n{:}t_n, \Gamma\rangle\ P_n\ \langle \Gamma'\rangle}{\Pi \vdash \langle y : t_1 \oplus \ldots \oplus t_n, \Gamma\rangle\ \textbf{case}\ y\ \textbf{of}\ \mathsf{in}_1\, x_1 \Rightarrow P_1 \mid \ldots \mid \mathsf{in}_n\, x_n \Rightarrow P_n\ \langle \Gamma'\rangle}$$

Summen und Tupel ermöglichen es nun, rekursive Datentypen zu definieren (z.B. Listentypen).

11.1 Zahlentypen

Mit Hilfe von Tupeln lassen sich Integer sowie Quanteninteger einer festen Länge definieren. Zum Beispiel ist $\mathbf{byte} = \bigotimes_{i=1}^{8} \mathbf{bit}$ und $\mathbf{qlong} = \bigotimes_{i=1}^{64} \mathbf{qbit}$. Darüber hinaus ist es sinnvoll, vordefinierte Operationen für diese Typen zu entwerfen, um die Programmierung zu vereinfachen. Sicher nützlich wären zum Beispiel Addition, Multiplikation, Subtraktion und Division für Quantenzahltypen.

Des weiteren lassen sich unendliche Zahlentypen definieren. Die ganzen Zahlen lassen sich zum Beispiel definieren als $\mathbf{int} := \bigoplus_{i=1}^{\infty} \mathbf{I}$. Quantenzahltypen unbestimmter Größe ließen sich – falls eine beliebige Anzahl von Quantebits zur Verfügung stünden – definieren als $\mathbf{qint} := \bigotimes_{i=1}^{\infty} \mathbf{qbit}$. Diese Definition kann sinnvoll sein, falls zusätzlich stets erfüllt ist, dass nur eine endliche Anzahl an Quantenbits allokiert werden müssen, um den Inhalt einer **qint**-Variablen zu repräsentieren.

11.2 Rekursive Typen

Von besonderem praktischem Wert für die Programmierung sind rekursive Typen, die es erlauben, Listen oder Bäume zu modellieren. Ein Listentyp für Quantenbits wird rekursiv definiert als $L := \mathbf{I} \oplus (\mathbf{qbit} \otimes L)$, ein Baumtyp für Quantenbits wäre $T := \mathbf{I} \oplus (T \otimes \mathbf{qbit} \otimes T)$. Man beachte, dass quantenmechanische Informationen stets nur an den "Blättern" eines strukturierten Typs auftreten können, da der $\oplus$-Operator eine klassische Auswahl darstellt. Ein strukturierter Typ mit einer "quantenmechanischen" Struktur kann durch Tupel und Summen nicht erzeugt werden. Fraglich ist auch, ob solche Typen mit quantenmechanischer Struktur algorithmisch sinnvoll verarbeitet werden können.

Listen von Quantenbits eignen sich gut für die Programmierung, wie im folgenden Beispiel der Quantenfouriertransfomration deutlich werden wird. Zudem erlauben solche Listen die Darstellung von beliebig langen Zahlen, die im Gegensatz zu der sonst üblichen Array-orientierten Implementierung von Quantenzahltypen keine Indexprüfungen benötigen.

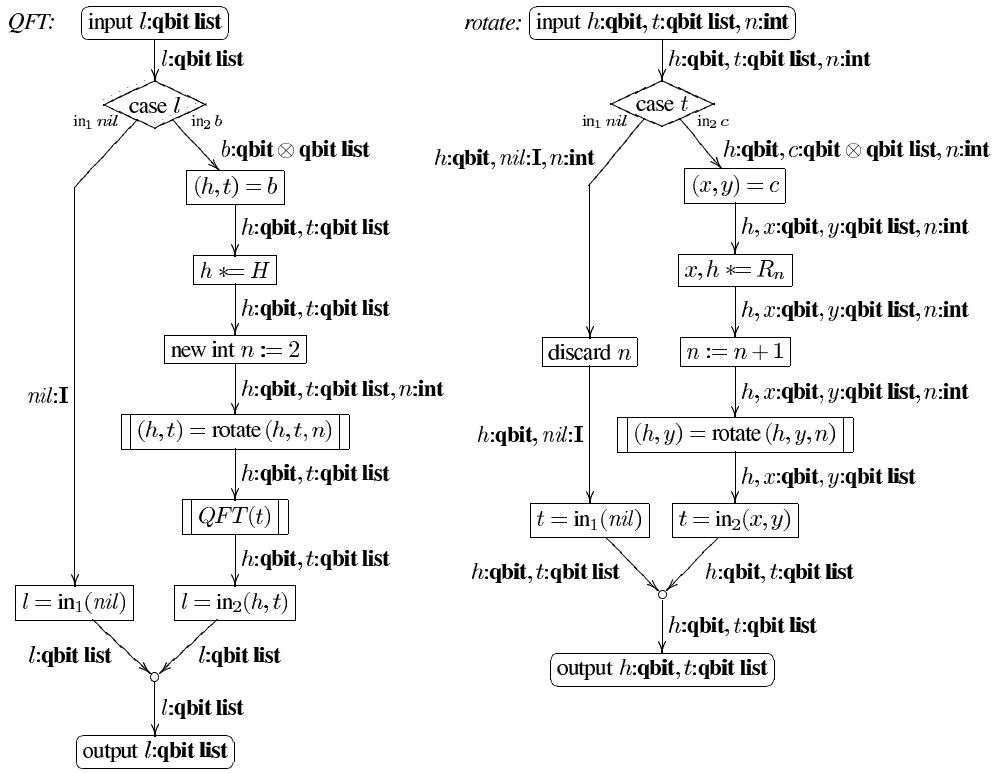


Abbildung 10: Quantenfouriertransformation

12 Fouriertransformation

Das abschließende Beispiel dieser Ausarbeitung ist die Quantenfourriertransformation, die auf einer Liste von **qbit** eine Fourriertransformation durchführt. Der hier verwendete Typ **qbit list** stellt eine Liste von Quantenbits dar und ist wie folgt definiert: $\mathbf{qbit\ list} := \mathbf{I} \oplus (\mathbf{qbit} \otimes \mathbf{qbit\ list})$. Zusätzlich wird eine parametrisierte unitäre Matrix R_n benötigt.

$$R_n := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{2\pi i/2^n} \end{pmatrix}.$$

Zusätzlich wird der Typ für natürliche Zahlen, **int**, zusammen mit einer vordefinierten Addition verwendet. Die Flussdiagramme werden um ein **case**-Primitiv erweitert mit dem Typ $A \oplus B \to A; B$, welches dem Typ nach ein Verzweigungsprimitiv ist.

Die Quantenfourriertransformation in textueller Notation

```
proc Rotate : h : qbit, t : qbit list, n : int → h : qbit, t : qbit list
{
  case t of
    in1 nil ⇒
      discard n;
      t = in1(nil)
    | in2 b ⇒
      (x, y) = c;
      x, h∗ = Rn;
      n := n + 1;
      (h, y) = Rotate(h, y, n);
      t = in2(x, y)
}
in
proc QFT : l : qbit list → l : qbit list
{
  case l of
    in1 nil ⇒
      l = in1(nil)
    | in2 b ⇒
      (h, t) = b;
      h∗ = H;
      new int n := 2;
      (h, t) = Rotate(h, t, n);
      QFT(t);
      l = in2(h, t)
}
in
...
```

13 Funktional oder nichtfunktional?

Abschließend muss noch die folgende Frage behandelt werden: Ist die hier vorgestellte Programmiersprache und der Flussdiagrammformalismus funktional oder nichtfunktional? Peter Selinger stellt in seinem Artikel die Formalismen so dar, als wären sie funktional [Sel02]. Er behauptet, dass jedes Primitiv eine Funktion darstellt, die auf eine wohldefinierte Variablenmenge operiert, und diese umformt. So wäre zum Beispiel eine Wertzuweisungsoperation $b := \mathbf{0}$ als Funktion auf b zu verstehen: $(:= \mathbf{0})(b)$.

Dennoch behaupte ich, dass der Formalismus nicht funktional, sondern imperativ ist. Ein Indiz hierfür ist schon, dass sich Funktionen höherer Ordnung nicht einfach in den Formalismus einbetten lassen. Es gibt jedoch zwei weitere Argumente, die noch stärker auf den imperativen Charakter von QPL hindeuten.

Zum einen existieren Wertzuweisungsoperatoren, die keine Überschattung

eines bereits existierenden Wertes durchführen, sondern den Wert an sich ändern. Dadurch wird das Paradigma der referentiellen Transparenz verletzt. Das Speichermodell ist nicht wie zum Beispiel in Haskell oder SML ein Umgebungsmodell, zu der Bindungen hinzugefügt oder entfernt werden. Vielmehr ist die Modellierung angelehnt an einen klassischen von-Neumann Rechner mit zusätzlichen QBit-Speichermodul.

Zum anderen existieren Formalismen, die auf ähnliche Weise imperative Programmiersprachen formalisieren, wie es Peter Selinger für QPL getan hat. Ein bekanntes und klassisches Beispiel hierfür ist die Spezifikationssprache Z ([Dra]). Hier werden auch globale Systemzustände definiert, auf die durch Wertzuweisungen oder Prozeduren (wohldefinierte und funktional anmutende) Transformationen durchgeführt werden. Dennoch bleibt Z eine Spezifikationssprache für imperative Programmiersprachen. Die Formulierung der Operationen als Funktionen dient bei Z einzig und allein dazu, die Wirkungsweise von imperativen Programmmen zu spezifizieren. Die Semantik von QPL wird hier in ähnliche Weise angegangen, dafür bleibt QPL und der Flusdiagrammformalismus imperativ.

Literatur

[Dra] Consensus Working Draft. Formal specification— z notation— syntax, type and semantics.

[Sel02] Peter Selinger. Towards a quantum programming language. http://quasar.mathstat.uottawa.ca/~selinger/papers/qpl.pdf.gz, January 2002.