Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

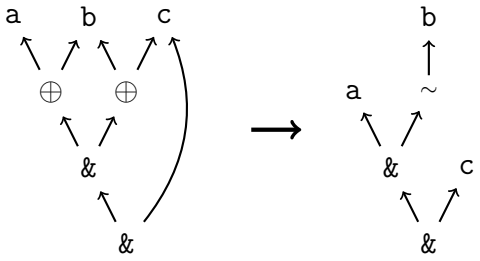
Lehrstuhl Prof. Dr.-Ing. Snelting

Normalisierung Graph-basierter Zwischensprachen

Bachelorarbeit von

Victor Pfautz

an der Fakultät für Informatik



Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr.-Ing. Jörg Henkel
Betreuender Mitarbeiter:	Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 24. Februar 2015 – 7. Mai 2015

Zusammenfassung

Moderne Zwischensprachen basieren auf der *Static Single Assignment* (SSA) Form, die es erlaubt, Programme als Graphen darzustellen. Dabei stellen Knoten SSA-Werte und Kanten Abhängigkeiten zwischen diesen Werten dar. Ziel dieser Arbeit ist es, Teilgraphen der Zwischensprachen mit gleicher Semantik zu normalisieren. Dadurch können gemeinsame Teilausdrücke identifiziert und weitere Optimierungen ermöglicht werden. Um das zu bewerkstelligen, wird zunächst zwischen Bitoperationen und arithmetischen Operationen unterschieden. Für Bitoperationen wird das *Shannon*-Verfahren vorgestellt, welches einen erheblichen Teil der mehrfach verwendeten Teilausdrücke zur Compilezeit berechnen kann. Um weitere Redundanzen eliminieren zu können, wird der Teilgraph in Partitionen aufgeteilt. In diesen Partitionen wird die Assoziativität aller Operationen gewährleistet, so können leicht Optimierungen wie `a & ~a` $\rightarrow$ `0` erkannt und durchgeführt werden.
Partitionen arithmetischer Operationen werden als Polynome ersten Grades dargestellt und anschließend zu einer Minimalform umgeformt. So wird `(a - b) + (a - b) + (a - b)` zunächst als `3 * a - 3 * b` gespeichert und anschließend zu `3 * (a - b)` zusammen gefasst. Dabei wird der Einfluss von *Strength Reduction* beachtet.
Im Anschluss wird ein Ausblick auf partitionsübergreifende Optimierungen gegeben. Dabei werden alle Partitionen als Ganzes betrachtet und so auch Optimierungen zwischen arithmetischen und Bitoperationen ermöglicht.

Abstract

Modern intermediate languages are based on the *Static Single Assignment* (SSA) form, which allows to represent a program as a graph. Nodes represent SSA values and edges represent dependencies between these values. This thesis aims to find and normalize subgraphs with equal semantics, which allows to find common subexpressions and further optimizations. To accomplish this, a distinction is made between bitwise and arithmetical instructions. In case of bitwise instructions, the *Shannon* process is able to compute a substantial part of the multiple used subexpressions at compile time. For the elimination of even more redundancy the subgraph is splitted in partitions, in which the associativity between each instruction is guaranteed. In this way it is easy to find and apply optimizations like `a & ~a` $\rightarrow$ `0`.
Partitions with arithmetical instructions are represented as first-degree polynomial. First, the expression `(a - b) + (a - b) + (a - b)` will be normalized into `3 * a - 3 * b`. A later phase considers *Strength Reduction* and transforms `3 * a - 3 * b` to `3 * (a - b)`.
Later, I present an outlook of optimizations across multiple partition that can handle all partition types and make optimizations between arithmetical and bitwise instructions possible.

Inhaltsverzeichnis

1. Einführung

Der Optimierungsvorgang moderner Compiler ist in mehrere Phasen aufgeteilt, die nacheinander ausgeführt werden. Es ist möglich Phasen mehrmals auszuführen. Die Phase für lokale Optimierungen betrachtet einzelne Operationen und versucht diese unabhängig vom größeren Zusammenhang zu optimieren. Die Reassoziations-Phase versucht den Programmcode zu normalisieren, sodass mehr lokale Optimierungen angewandt werden können oder mehr gleiche Teilterme erkannt werden können. Ziel dieser Arbeit ist es, Terme, die nur aus arithmetischen Operationen oder Bitoperationen bestehen, zu normalisieren, sodass mehr lokale Optimierungen angewandt werden können. Diese Normalisierung findet in der Zwischensprache statt, welche auf der *Static Single Assignment* (SSA) Form basiert und damit Programme als gerichteten Graphen darstellen kann.

Bei arithmetischen Operationen lässt sich `a + b - a` zu `(a - a) + b` umformen, sodass eine lokale Optimierung `a - a` zu `0` und `0 + b` zu `b` vereinfachen kann. Damit können zwei Operationen eingespart werden. Näheres dazu im Kapitel 4.4.

Bei Bitoperationen lässt sich beispielsweise `(a & b) & ~a` zu `(a & ~a) & b` umformen. Im Anschluss ersetzen lokale Optimierungen `(a & ~a)` durch `0` und `0 & b` durch `0`. Bei diesem Beispiel ist es auch möglich, das `a` in der Klammer zur Compilezeit zu berechnen. `(a & b) & ~a` kann zu `(0 & b) & ~a` umgeformt werden. Lokale Optimierungen erkennen, dass dieser Teilterm immer zu `0` ausgewertet wird. Diese Normalisierung basiert auf dem *Shannon*-Verfahren und wird in Kapitel 4.1 behandelt. Im Anschluss gehen Kapitel 4.2 und 4.3 auf weitere Verbesserungen dieses Verfahrens ein.

Bei diesen Optimierungen sind die Kosten und Nutzen jedoch nicht aus den Augen zu lassen. Bei Spezialfällen kann es unrentabel sein, eine Optimierung durchzuführen. Es ist denkbar, dass eine vermeintliche Optimierung in einigen Fällen zu mehr Operationen führt, was zu vermeiden ist.

2. Grundlagen

In dieser Arbeit sei eine Bitoperation ausschließlich `&`, `|`, $\sim$ oder $\oplus$. *Bitshift* wird hier nicht dazugezählt, da bei dieser Operation Bits nicht getrennt voneinander betrachtet werden können. Mit dieser Einschränkung können im Folgenden Ausdrücke mit n Bits gleich betrachtet werden wie Ausdrücke mit einem Bit. Eine Zahl, die nur aus Einsen besteht, wird als `1` geschrieben, eine Zahl, die nur aus Nullen besteht, als `0`.

Firm ist eine Graph-basierte Zwischensprache, welche Programme in SSA-Form darstellt. Die *Static Single Assignment* (SSA) Form ist eine spezielle Form der *Intermediate Representation*. Sie gibt an, dass jede Variable genau einmal zugewiesen wird und definiert wurde, bevor sie genutzt wird. Falls mehrmals in eine Variable geschrieben wird, werden verschiedene Versionen von dieser Variable erstellt. Da so klar ist, woher jede Variable kommt, können Optimierungen leichter durchgeführt werden.

Durch die Kombination von Datenabhängigkeiten und dem Steuerfluss, kann Firm ein Programm als gerichteten Graphen darstellen. Die C-Bibliothek für Firm ist libFirm [6]. Firm wird an der Universität Karlsruhe (KIT) [10] entwickelt. Das Ergebnis von Operationen wird als Knoten repräsentiert, Datenabhängigkeiten und der Steuerfluss als Kanten. Da es innerhalb eines Grundblocks keine vorgegebene Reihenfolge der Befehle gibt, ist diese allein durch die Datenabhängigkeiten bestimmt. So kann beispielsweise eine Addition erst ausgeführt werden, wenn ihre Operanden vorhanden sind. In Algorithmus 1 ist eine einfache Funktion gegeben, welche zwei Werte addiert. In Abbildung 2.1 ist die Firm-Repräsentation des Codes abgebildet. Jeder Knoten hat einen Modus, der

Algorithmus 1 Eine Funktion, die ihre beiden Argumente addiert.

```
1: function F(a,b)
2:     return a+b
```

angibt, um welchen Datentyp es sich handelt. Operationen sind auf bestimmte Modi beschränkt. So kann eine Multiplikation nur zwischen zwei gleichen Modi durchgeführt werden. Um Zeigerarithmetik zu ermöglichen, kann eine Addition aber auch zwischen einem Integer-Wert und einer Adresse stattfinden.

Der Optimierungsprozess von Firm basiert auf mehreren Phasen, welche mehrmals hintereinander ausgeführt werden können. Im Folgenden werden zum einen die Phase für lokale Optimierungen und zum anderen die Reassoziations-Phase betrachtet.

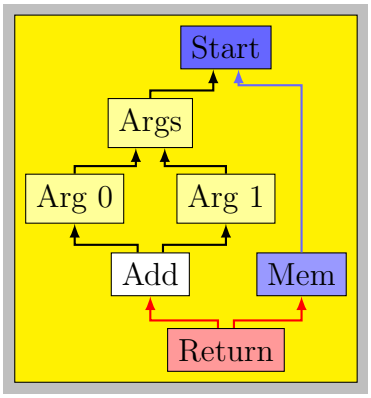


Abbildung 2.1.: Algorithmus 1 in Firm dargestellt.

2.1. Lokale Optimierungen

Lokale Optimierungen werden jeweils auf einer Operation ausgeführt, um Vereinfachungen wie $\mathtt{a} + 0 \rightarrow 0$ zu finden. Da diese Optimierung von Hand zu erstellen aufwendig ist, wurde von Bersch [1] ein Prototyp eines Generators für lokale Optimierungen entwickelt. Diese Arbeit wurde von Buchwald betreut, anschließend neu implementiert und steht nun libFirm als *Optgen* [3] zur Verfügung.

Um lokale Optimierungen zu finden, werden zunächst Terme mit gleichem Ergebnis, aber einer unterschiedlichen Anzahl an Operationen, gesucht. Anschließend wird jeder nichtminimale Term mit seiner minimalen Version gespeichert. Da sich viele Optimierungen zu einfachen Regeln zusammenfassen lassen, werden Regeln gesucht, die viele Optimierungen ermöglichen. In Kapitel 5.2 in *Optgen: A Generator for Local Optimizations* [3] ist eine Auflistung dieser Regeln, inklusive ihres Vorhandenseins in Standardcompilern, zu finden.

2.2. Reassoziation und Normalisierung

Die Reassoziations-Phase dient der Normalisierung des Graphen. Dabei wird durch Umordnen versucht, eine einheitliche Struktur zu erlangen. So können Regeln allgemeiner definiert und gemeinsame Teilausdrücke leichter gefunden werden. *Optgen* [3] hat ein Defizit an Reassoziationsregeln aufgezeigt. So konnten einige Testfälle nicht optimiert werden, da sie nicht in die für lokale Optimierungen nötige Struktur umgeformt wurden. Diese Arbeit beschäftigt sich damit, Terme zu normalisieren und umzuordnen, sodass möglichst viele der übrig gebliebenen Testfälle durch lokale Optimierungen gelöst werden können.

3. Verwandte Arbeiten

Es haben sich bereits verschiedene Arbeiten mit der Vereinfachung von Termen beschäftigt. Unter anderem fokussierten sich einige Arbeiten auf die Prozesse, die in der Reassoziationsphase stattfinden. Dabei wird zuerst in Kapitel 3.1 auf Schleifeninvarianten eingegangen. Danach gehen Kapitel 3.2 und 3.3 auf *Value Numbering* und *Strength Reduction* ein.

3.1. Behandlung von Schleifeninvarianten

In der Reassoziations-Phase finden noch weitere Prozesse statt, so auch das Umordnen von Schleifeninvarianten. Zum Beispiel ist ein Wert, welcher in einer Schleife verwendet, aber nicht verändert wird, in der Schleife invariant. Es bietet sich an, diesen Wert außerhalb der Schleife zu berechnen. Daher werden die dafür nötigen Operationen aus der Schleife gezogen. Muchnick verwendet diese Idee um Adressberechnungen in Schleifen zu optimieren [7].

Algorithmus 2 Elementzugriff auf ein zweidimensionales Array.

```
1: a ← array[lo1..hi1, lo2..hi2]
2: i ← c
3: ...
4: for all j = lo2 to hi2 do
5:     a[i, j] ← b + a[i, j]
```

Algorithmus 2 veranschaulicht den elementweisen Zugriff auf ein Array, aus einer Schleife heraus. Ohne Optimierung muss die gesamte Adressberechnung in jedem Schleifendurchlauf vollzogen werden. Die Adresse des Elements `a[i,j]` berechnet sich wie folgt:

```
base_a + ((i - lo1) * (hi2 - lo2 + 1) + j - lo2) * w.
```

`base_a` ist die Basisadresse des Arrays und `w` die Wortbreite eines Array-Elements. Dieser Term kann zu

```
base_a - (lo1*(hi2-lo2+1)+lo2)*w + (hi2-lo2+1)*i*w + j*w
```

umgeformt werden. Da `lo1`, `hi1`, `lo2`, `hi2` und `w` zur Compilezeit bekannt sind, kann `-(lo1*(hi2-lo2+1)+lo2)*w` bereits zur Compilezeit berechnet werden. `base_a` und `i` sind schleifeninvariant, so kann `base_a+(hi2-lo2+1)*i*w` außerhalb der Schleife berechnet werden. Nach dieser Optimierung ist nur noch `j*w` innerhalb der Schleife zu berechnen. Vereinfacht sieht Algorithmus 2 nun so aus:

Algorithmus 3 Optimierter Elementzugriff auf ein Array.

```
1: ...
2: x ← base_a - (lo1 * (hi2 - lo2 + 1) + lo2) * w + (hi2 - lo2 + 1) * i * w
3: for all j = lo2 to hi2 do
4:     *(x + j * w) ← b + *(x + j * w)
```

Der unäre Operator `*` steht hier für die Dereferenzierung, der nachfolgenden Adresse. In den folgenden zwei Kapiteln werden zwei weitere Verfahren vorgestellt, welche Algorithmus 3 noch weiter optimieren können.

3.2. Value Numbering

Die Adressberechnung `x+j*w` innerhalb der Schleife wird zweimal ausgeführt. Das kann mit *Value Numbering* [2] behoben werden. Bei diesem Verfahren erhalten alle Variablen mit gleichem Inhalt die selbe Nummer. Bei Zuweisungen wird diese Nummer übertragen, so können leicht gleiche Ausdrücke erkannt werden. Durch eine Normalform ist es leichter, gleiche Ausdrücke zu finden. In Algorithmus 3 wird daher `x+j*w` nur einmal pro Schleifendurchlauf berechnet.

Die Terme `b&a&c` und `b&c&a` haben offensichtlich das selbe Ergebnis. Allerdings können sie von *Value Numbering* nicht als gleiche Terme erkannt werden. Eine mögliche Normalform wäre `a&b&c`. Wenn beide Terme in diese Form überführt werden, kann *Value Numbering* beide Ausdrücke als gleich werten.

3.3. Strength Reduction

In Algorithmus 3 enthält der Term `x+j*w` eine Addition und eine Multiplikation, welche in jedem Schleifendurchlauf ausgeführt werden. Da `j` bei jedem Schleifendurchlauf um eins erhöht wird, ist die Erhöhung der Adresse pro Schleifendurchlauf invariant und muss daher nicht in der Schleife berechnet werden. In Algorithmus 4 wird die Adresse bei jedem Durchlauf um `w` erhöht. Das spart die Multiplikation im Schleifenrumpf ein. Um trotzdem die Schleifenabbruchbedingung zu gewährleisten, müssen vor der Schleife zwei

Algorithmus 4 Weiter optimierter Elementzugriff auf ein Array.

```
1: ...
2: y = x + lo2 * w
3: while y ≤ x + hi2 * w do
4:     *y ← b + *y
5:     y ← y + w
```

weitere Berechnungen durchgeführt werden. In Zeile 2 wird die Adresse des Elements `lo2` berechnet und in Zeile 3 die Adresse des Elements `hi2`.

In diesem Beispiel war es möglich, eine Multiplikation durch das zyklische Hinzuaddieren des Multiplikators einzusparen. Dieses Verfahren nennt sich *Operator Strength Reduction* [4]. Das Ziel von *Strength Reduction* ist es, komplizierte Operationen wie ganzzahlige Multiplikation oder Division durch leichtere Operationen wie Addition, Subtraktion oder Bitshift zu ersetzen. Ein weit verbreiteter Anwendungsfall sind Multiplikationen und Divisionen mit Konstanten. So kann `8*a` zu `a<<3` umgeformt werden.

3.4. Testverfahren

Die Compileroptimierung kann die Ausführungszeit von Programmen erheblich verbessern. Diese Verbesserung geht allerdings oft auf Kosten der Fehlerfreiheit. Um die Qualität von Compilern zu verbessern, gibt es unterschiedliche Möglichkeiten. Zum einen können Tests von Hand geschrieben werden. Da der Programmierer in der Regel nicht auf Probleme Testen kann, die ihm nicht bekannt sind, wird eine bessere Methode benötigt. *Csmith* [11] ist ein Generator, der zufällige Testfälle für C-Compiler erstellt. Dabei wird nicht nur getestet, ob der Compiler während dem Kompiliervorgang abstürzt, sondern auch, ob das Kompilat die gewünschte Ausgabe erzeugt. Nun stellt sich die Frage was die *richtige* Ausgabe ist. *Csmith* löst dieses Problem durch einen zufallsbasierten Differenztest (englisch: *randomized differential testing*). Dabei wird ein Testprogramm mit drei unterschiedlichen Compilern kompiliert und das Ergebnis verglichen. Die häufigste Ausgabe wird als die *korrekte* Ausgabe gewertet. Falls ein Kompilat eine andere Ausgabe ausgibt, wird ein Fehler ausgegeben.

Die Testprogramme von *Csmith* sind relativ lang und beinhalten viele Instruktionen. Um einen kleinen Testfall für einen Fehler zu finden, wurde *C-Reduce* [8] entwickelt. Dieses Programm verkleinert den Testfall zyklisch bis es einen Fixpunkt erreicht hat.

4. Entwurf und Implementierung

In diesem Kapitel werden verschiedene Vorgehensweisen präsentiert, wie ein Graph normalisiert werden kann. Zuerst wird in Kapitel 4.1 ein *Shannon*-Vereinfachung genanntes Verfahren vorgestellt, welches bei Bitoperationen einzelne Variablen zur Compilezeit berechnen kann. Anschließend werden in Kapitel 4.2 und 4.3 Erweiterungen dieses Verfahrens angesprochen, die es erlauben, noch mehr Terme zu normalisieren. Schließlich wird in Kapitel 4.4 die Normalisierung gleicher Operationen betrachtet.

4.1. Shannon-Verfahren

Zur besseren Lesbarkeit wird im Folgenden eine Zahl, welche nur aus Nullen besteht als `0` und eine Zahl, welche nur aus Einsen besteht als `1` geschrieben. Bei der *Shannon*-Zerlegung [9] wird ein schaltalgebraischer Term in zwei Teile aufgeteilt, einen mit `a == 1` und einen mit `a == 0`. Für eine beliebige schaltalgebraische Funktion `f(a)` gilt:

```
f(a) = (a & f(1)) | (~a & f(0)).
```

Zu beachten ist, dass `f(a)` nur aus Bitoperationen besteht. Alle anderen Operationen werden wie Variablen behandelt, in denen nichts ersetzt werden darf. `a` kann ein beliebiger Knoten sein. Bei der *Shannon*-Zerlegung sind zwei Dinge nützlich: Zum einen ist das Verfahren auf beliebig viele Bits übertragbar. Zum anderen kann ein Term wie folgt ersetzt werden:

```
f(a) & a = ((a & f(1)) | (~a & f(0))) & a
         = (a & f(1)) | (a & ~a & f(0))
         = f(1) & a.
```

Für bitweises Oder gilt:

```
f(a) | a = ((a & f(1)) | (~a & f(0))) | a
         = (~a & f(0)) | a
         = (~a | a) & (f(0) | a)
         = f(0) | a.
```

In Firm werden Ausdrücke als gerichtete Graphen dargestellt. Daher gilt es nun diese Fälle in Graphen zu erkennen.

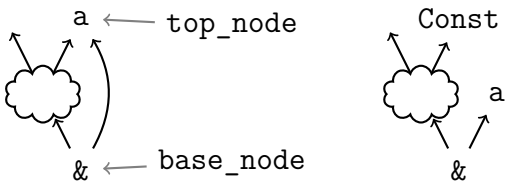


Abbildung 4.1.: Definition von `top_node` und `base_node`

Der untere Knoten, in Abbildung 4.1 das `&`, wird `base_node` genannt. Der obere Knoten, in Abbildung 4.1 das `a`, wird `top_node` genannt. Alle Knoten in der Wolke dürfen nur Verwender aus der Wolke oder die `base_node` als Verwender haben. Die `top_node` wird in der Wolke auf dem linken Pfad durch eine Konstante ersetzt. Auf dem rechten Pfad bleibt die `top_node` bestehen. Die Abbildung 4.2 zeigt ein Beispiel zu diesem Vorgehen.

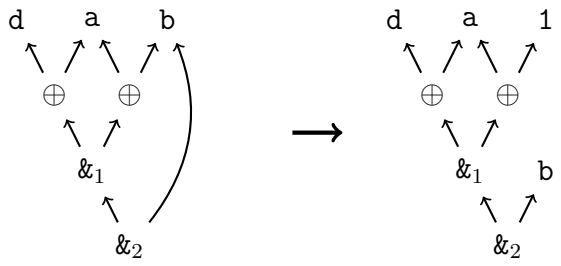


Abbildung 4.2.: Anwendung des *Shannon*-Verfahrens auf den Knoten `b`.

Vom Wurzelknoten aus gesehen besteht der linke Teilausdruck nur aus Bitoperationen. Der rechte Teilausdruck `b` wird komplett vom linken Teilausdruck verwendet. In diesem Beispiel ist $\&_2$ die `base_node` und `b` die `top_node`. Nun kann `b` im linken Teilausdruck durch 1 ersetzt werden. Somit wird eine Wiederverwendung von `b` eingespart. $\texttt{a} \oplus 1$ kann durch `~a` ersetzt werden.

Nun fällt auf, dass `a` zweimal in einem Term aus Bitoperationen verwendet wird. Hierbei handelt es sich um einen Spezialfall. Auf dem rechten Pfad ist es möglich, dass eine

Operation (∼) vorkommt, sodass die Substitution dennoch gültig ist. Diese wird im Folgenden `middle_node` genannt.

```
f(a) & ~a = ((a & f(1)) | (~a & f(0))) & ~a
          = (a & ~a & f(1)) | (f(0) & ~a)
          = f(0) & ~a.
```

Dieses Verfahren ist auf | übertragbar.

```
f(a) | ~a = ((a & f(1)) | (~a & f(0))) | ~a
          = (a & f(1)) | ~a
          = (a | ~a) & (f(1) | ~a)
          = f(1) | ~a.
```

In Abbildung 4.3 ist die Fortsetzung des obigen Beispiels zu sehen.

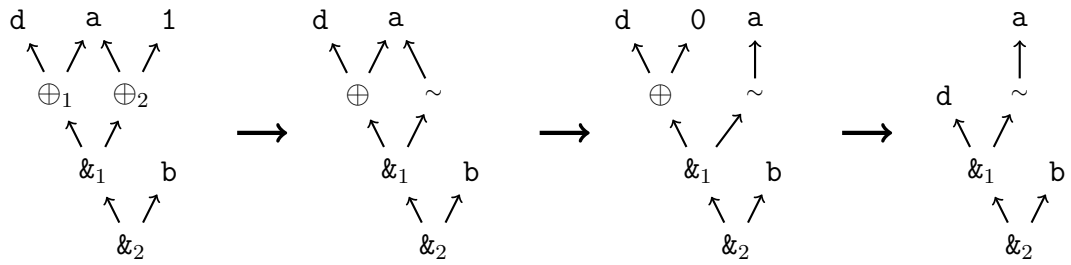


Abbildung 4.3.: Fortsetzung des Beispiels aus Abbildung 4.2.

In Abbildung 4.3 wird zunächst `a` ⊕ `1` durch ∼`a` ersetzt. Nun ist `&`$_1$ die `base_node` und das `a` die `top_node`. Der Operand `a` von $\oplus_1$ kann durch `0` ersetzt werden. Anschließend können lokale Optimierungen `d` ⊕ `0` durch `d` ersetzen.

Es wurde also zuerst das ⊕ durch ein ∼ ersetzt und anschließend das *Shannon*-Verfahren angewandt. Diese zwei Schritte können zusammengefasst werden. Das *Shannon*-Verfahren kann weiter verallgemeinert werden. Das bisherige Verfahren unterscheidet zwischen einem negierten und nicht negierten rechten Teilausdrucks. Auf mehrere Bits verallgemeinert ist das ⊕ mit einer Konstanten.

$$\texttt{f(a) \& (a} \oplus \texttt{C)} \rightarrow \texttt{f(1} \oplus \texttt{C) \& (a} \oplus \texttt{C)}$$

$$\texttt{f(a) | (a} \oplus \texttt{C)} \rightarrow \texttt{f(0} \oplus \texttt{C) | (a} \oplus \texttt{C)}$$

Hierbei ist zu beachten, dass die rechte Seite weitere Verwender haben kann, denn nur im linken Teilterm wird etwas verändert. Nun kann das obige Beispiel direkt mit ⊕ mit einer Konstanten gelöst werden und muss nicht zuerst in ein ∼ umgewandelt werden.

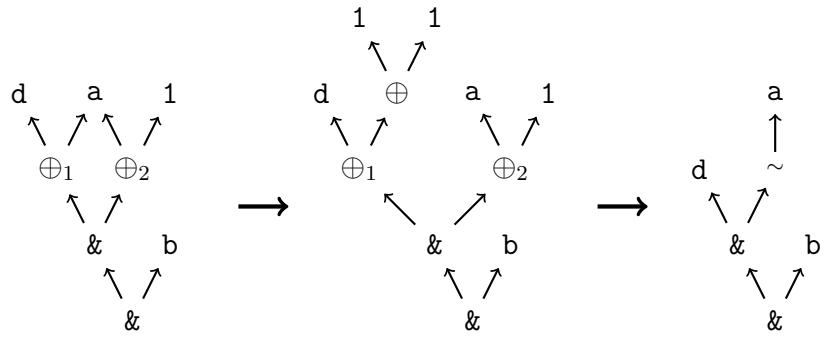


Abbildung 4.4.: Beispiel aus Abbildung 4.3 direkt, ohne Umweg über Negierung, gelöst.

In Abbildung 4.4 wird der Operand `a` von $\oplus_1$ durch $\oplus_2$ ersetzt, wobei in diesem `a` durch `1` ersetzt wird. Die so dazu gekommene Operation hat nur konstante Operanden und kann zur Compilezeit berechnet werden. Somit wurde auch die zweite Mehrfachverwendung aufgehoben und insgesamt eine Operation eingespart.

Der aufmerksame Leser mag sich fragen, ob es auch möglich ist $\oplus_1$ zu verwenden und so das `a` von $\oplus_2$ zu ersetzen. Das läuft darauf hinaus, dass `a` durch `d` ersetzt wird. So wird die Mehrfachverwendung von `a` eingespart, allerdings kommt eine neue Mehrfachverwendung durch `d` hinzu, sodass dieses Vorgehen hier nicht erwünscht ist. Auf dieses Thema wird in Kapitel 4.3 weiter eingegangen.

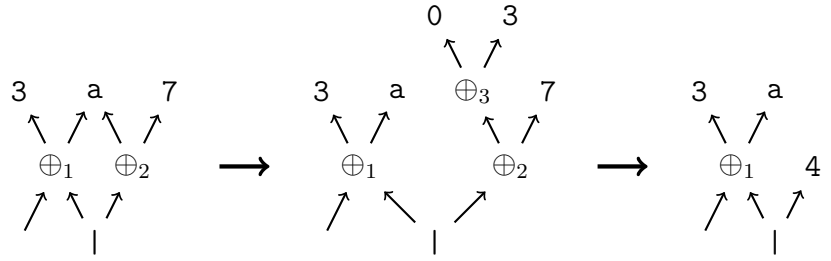


Abbildung 4.5.: Anwendung des *Shannon*-Verfahrens auf Zahlen mit mehreren Bits.

Bisher wurde nur ein Bit betrachtet, in Abbildung 4.5 ist ein Beispiel mit mehreren Bits zu sehen. $\oplus_1$ hat hier zwei Verwender, sodass dessen Operand `a` nicht ersetzt werden darf. $\oplus_2$ hat nur einen Verwender, sodass dessen Operand `a` ersetzt werden kann. Der linke Teilterm ist $\texttt{a} \oplus \texttt{3}$. In diesem Term wird wie bisher die `top_node` durch die Konstante ersetzt. Diese ist hier `0`, da die `base_node` ein `|` ist. Somit wird die `top_node` durch $\texttt{0} \oplus \texttt{3}$ ersetzt. Der rechte Teilterm von `|` besteht nur aus konstanten Operanden und kann durch lokale Optimierungen zu `4` vereinfacht werden.
In diesem Kapitel wurde das `Shannon`-Verfahren in der Theorie vorgestellt. Die folgenden Kapitel gehen auf die Implementierung dieses Verfahrens ein.

4.1.1. Analyse-Phase

Im Analysevorgang wird der Algorithmus 5 für jeden Knoten ausgeführt. Falls der aktuelle Knoten kein `&` oder `|` ist, kann er keine `base_node` sein und der Algorithmus wird in Zeile 2 abgebrochen. Die Schleife in Zeile 6 wird zweimal ausgeführt, sodass in Verbindung mit dem Vertauschen von `left` und `right` in Zeile 14 einmal vom rechten und einmal vom linken Operanden aus die `top_node` gesucht wird. In Zeile 8 bis 11 wird bestimmt, ob `left` eine `middle_node` oder eine `top_node` ist. Eine `middle_node` ist entweder ein $\oplus$ mit einer Konstanten oder ein $\sim$. In Zeile 12 werden die Besucherzähler zurückgesetzt. Anschließend kann die rekursive Suche nach `top_node` gestartet werden.

Algorithmus 5 Falls `base` eine `base_node` ist, werden entsprechende Optimierungsmöglichkeiten gespeichert.

```
 1: function TRY__BASENODE(base)
 2:     if not (base is And or base is Or) then
 3:         return
 4:     left ← base.left_operand
 5:     right ← base.right_operand
 6:     for 2 times do
 7:         top ← GET__TOPNODE__FROM__MIDDLENODE(left)
 8:         if top != left then
 9:             middle ← left
10:         else
11:             middle ← None
12:         RESET__VISITOR__COUNTERS()
13:         SEARCH__TOPNODE(right, base, base, middle, top)
14:         SWAP(left, right)
```

In Algorithmus 6 wird die Suche nach weiteren Pfaden zur `top_node` beschrieben. Der `node` Parameter ist der aktuell betrachtete Knoten, der rekursiv verändert wird. `middle` und `top` sind die möglichen Zielknoten. `base` ist der `base_node`, dieser bestimmt durch welche Konstante die `top_node` ersetzt wird. `other` ist der jeweils zuvor besuchte Knoten und daher der Knoten, dessen Operand ersetzt wird.

In Zeile 2 bis 8 wird geprüft, ob entweder die `middle_node` oder die `top_node` gefunden wurde. In diesem Fall, wird eine Optimierung gespeichert und die Rekursion beendet. Die Abfrage auf `base` $\neq$ `other`, in Zeile 6, ist nötig um Sonderfälle korrekt zu behandeln. Dazu mehr in Kapitel 4.1.4.

Nachdem der Besucherzähler für den Knoten `node` in Zeile 9 erhöht wurde, wird in

Zeile 10 geprüft, ob der Besucherzähler der Anzahl Verwender entspricht. Ist das der Fall wurde der aktuelle Knoten über jeden Verwender einmal besucht. Wenn der aktuelle Knoten zusätzlich eine Bitoperation ist, wird rekursiv über seine Operanden weiter nach der `top_node` gesucht.

Algorithmus 6 Rekursive Suche nach einem zweiten Pfad zur `top_node`.

```
1:  function SEARCH_TOPNODE(node, other, base, middle, top)
2:      if node = middle then
3:          middle', top' ← None, middle
4:      else
5:          middle', top' ← middle, top
6:      if node = top' and ((middle and len(middle.users) > 1) or base ≠ other) then
7:          optimizations.add({base, middle', top', other})
8:          return
9:      INC_VISITOR_COUNTER(node)
10:     if VISITOR_COUNT(node) = len(node.users) and is_bitop(node) then
11:         for all o in node.operands do
12:             SEARCH_TOPNODE(o, node, base, middle, top)
```

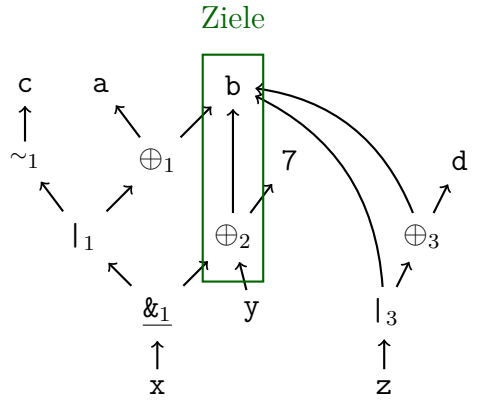


Abbildung 4.6.: Beispiel zur rekursiven Suche nach der `top_node`.

Abbildung 4.6 zeigt die rekursive Suche nach der `top_node` von $\&_1$ aus. Zunächst wird der rechte Operand $\oplus_2$ betrachtet. Da dieser ein $\oplus$ mit einer Konstanten ist, kann er ein `middle_node` sein. Somit wird der nicht konstante Operand `b` von $\oplus_2$ als `top_node` gespeichert und $\oplus_2$ als `middle_node`.

Nun wird von $|_1$ aus über die Operanden der Knoten traversiert, um alle möglichen Pfade zur `top_node` oder `middle_node` zu finden. Bei Knoten `c` wird nicht weiter traversiert, da `c` keine Bitoperation ist. Bei `b` wird abgebrochen, da ein Knoten aus der Zielmenge

gefunden wurde. Da `b` über beide Operanden von $\&_1$ gefunden werden konnte, ist `b` die `top_node` und $\oplus_2$ die `middle_node`. Bei der anschließenden Optimierung wird der Operand `b` von $\oplus_1$ durch eine Konstante ersetzt. $\oplus_1$ wird als `other_node` bezeichnet. Diese Werte werden in der Liste der Optimierungen gespeichert.

Anschließend wird $|_1$ als `top_node` angesehen und versucht über $\oplus_2$ diese zu erreichen. Beim Aufruf von `search_topnode` auf $\oplus_2$ ist der Besucherzähler eins und somit kleiner als die Anzahl Verwender. Daher muss dieser Knoten einen weiteren Verwender haben und die Rekursion wird nicht fortgeführt.

4.1.2. Optimierungen anwenden

Algorithmus 7 wendet die zuvor gespeicherten Optimierungen auf den Graphen an. Zeile 3 bis 10 prüfen ob die Optimierung noch anwendbar ist.

Algorithmus 7 Anwenden der Optimierungen, falls möglich.

```
 1: for all optimization in optimizations do
 2:     base, middle, top, other ← optimization
 3:     if not top in other.operands then
 4:         continue
 5:     if middle = None then
 6:         if not top in base.operands then
 7:             continue
 8:     else
 9:         if not (top in middle.operands and middle in base.operands) then
10:             continue
                                  ⎧0..0            if middle = None
11:     replacement ← replacement ⎨1..1            if middle is Not
                                  ⎩middle.tarval   if middle is Eor
12:     if base is And then
13:         replacement ← replacement ⊕ 1..1
14:     else
15:         replacement ← replacement ⊕ 0..0
16:     other.operands.replace(top, Const(replacement))
```

In Zeile 16 wird der Operand `top` von `other` durch eine Konstante ersetzt. Diese Konstante ist sowohl von `base` als auch von `middle` abhängig. Der Einfluss von `middle` wird in Zeile 11 berechnet. Dieser ist `0`, falls es keine `middle_node` gibt, `1..1` falls die `middle_node` ein $\sim$ ist und die Konstante des $\oplus$, falls `middle_node` ein $\oplus$ mit einer

Konstanten ist. Anschließend wird in Zeile 12 bis 15 der Typ der `base_node` verrechnet. Wie in Kapitel 4.1 beschrieben ist dieser für `& 1..1` und für `| 0`.

Dieser Vorgang wird für alle gespeicherten Optimierungen durchgeführt. Falls eine Optimierung durch eine bereits durchgeführte Optimierung nicht mehr anwendbar ist, wird diese in Zeile 3 bis 10 erkannt und ausgelassen.

4.1.3. Beispiel

In Abbildung 4.7 sind Mengen in den Graph eingezeichnet. Set $\&_1$ und Set $|_3$ beinhalten alle Knoten, welche durch die Rekursion von $\&_1$ und $|_3$ aufgerufen werden. Für alle Knoten, welche von einer Partition verwendet werden, wird überprüft, ob diese ein Ziel sind, jedoch wird in diesen Knoten die Rekursion nicht fortgeführt.

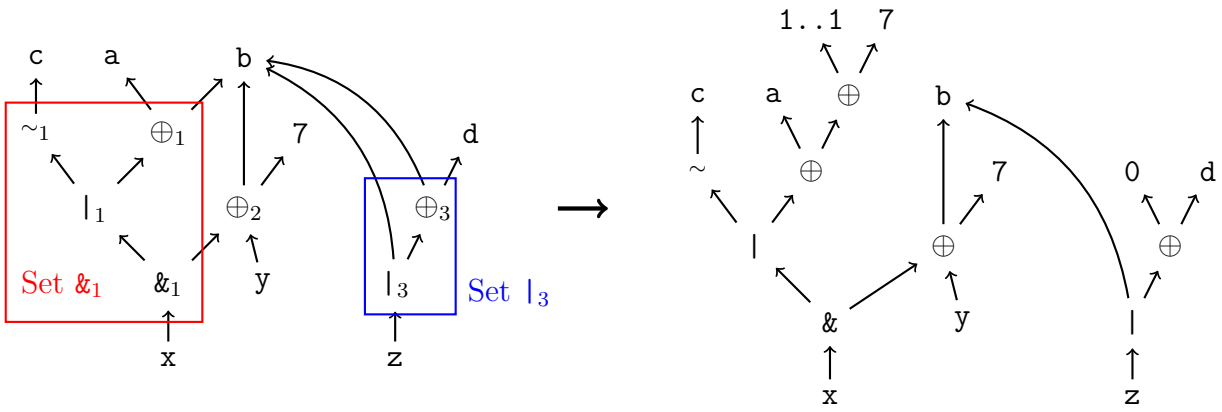


Abbildung 4.7.: Beispiel zur Anwendung des *Shannon*-Verfahrens.

`b` wurde von $\&_1$ aus über zwei Wege gefunden. Dabei wurde die `middle_node` $\oplus_2$ mit der Konstanten `7` verwendet. Da die `base_node` ein `&` ist, wird der Operand `b` von $\oplus_1$ durch `1..1` $\oplus$ `7` ersetzt.

In Set $|_3$ ist ebenfalls eine Optimierung möglich: `b` kann von $|_3$ über zwei Wege gefunden werden. Da hier keine `middle_node` existiert und die `base_node` $|_3$ ein `|` ist, wird der Operand `b` von $\oplus_3$ durch `0` $\oplus$ `0` = `0` ersetzt.

4.1.4. Sonderfälle

Dieses Kapitel beschäftigt sich mit der korrekten Behandlung von Sonderfällen. Der erste Fall von links in Abbildung 4.8 stellt eine `base_node` mit zwei gleichen Operanden dar. Es ist denkbar das *Shannon*-Verfahren, von beiden Operanden aus, auf den jeweils anderen Operanden anzuwenden. In beiden Fällen gibt es keine `middle_node`, `base_node = other_node` ist erfüllt und die Optimierung wird in Algorithmus 6 Zeile 6 ignoriert.

Der zweite und dritte Fall beinhalten zusätzlich eine `middle_node`. Auch in diesen Fällen gibt es zwei mögliche Ersetzungen. Bei `a & (a` $\oplus$ `Const)` $\rightarrow$ `(1` $\oplus$ `Const) & (a` $\oplus$ `Const)` kann `1` $\oplus$ `Const` direkt zu einer Konstante ausgewertet werden. So wird eine Verwendung von `a` eingespart und die Anzahl der Operationen bleibt gleich. Bei `a & (a` $\oplus$ `Const)` $\rightarrow$ `a & (1` $\oplus$ `Const)` wird `1` $\oplus$ `Const` direkt zu einer Konstanten vereinfacht. So wird nicht nur eine Wiederverwendung, sondern auch eine Operation eingespart. Daher ist diese Substitution zu bevorzugen. Da nur bei der ersten Ersetzung die Gleichung `base_node = other_node` gilt und die `middle_node` nur einen Verwender hat, wird diese ignoriert und die bessere Ersetzung gewählt.

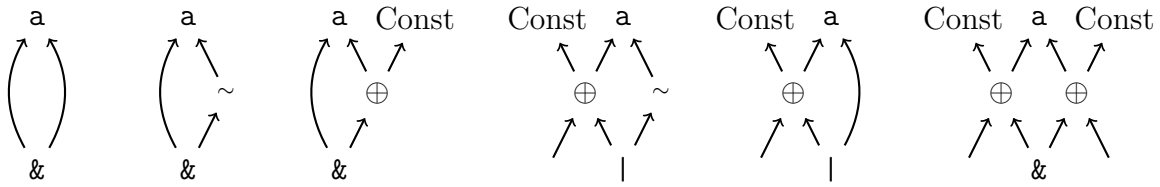


Abbildung 4.8.: Spezialfälle des *Shannon*-Verfahrens

Im vierten Fall ist nur die Substitution `(Const` $\oplus$ `a) |` $\sim$ `a` $\rightarrow$ `(Const` $\oplus$ `a) |` $\sim$ `(Const` $\oplus$ `0)` möglich, da sonst die Wiederverwendung von $\oplus$ verfälscht wird. Um die `top_node a` über den linken Pfad zu finden muss die Rekursion an $\oplus$ vorbei. Das ist nur möglich wenn alle Verwender von $\oplus$ besucht wurden, was hier nicht der Fall ist. Über den rechten Pfad kann `a` gefunden werden und da `base_node = other_node` nicht gilt, wird der Operand `a` von $\sim$ durch eine Konstante ersetzt.

Im fünften Fall darf wie im vorherigen Fall, `a` nur auf dem rechten Pfad ersetzt werden. Die Gleichung `base_node = other_node` ist erfüllt, allerdings hat die `middle_node` mehrere Verwender und somit wird die Optimierung korrekt durchgeführt.

Beim sechsten Fall darf keine Substitution erfolgen. Die Suche nach einer `top_node` wird in beiden $\oplus$ abgebrochen, da sie nicht über alle ihre Verwender besucht wurden. Daher wird auch dieser Sonderfall korrekt behandelt.

4.1.5. Laufzeit

In den vorherigen Kapiteln wurde das *Shannon*-Verfahren im Detail betrachtet. In diesem Kapitel wird die worst-case Laufzeit bestimmt. Sei N die Anzahl Knoten und E die Anzahl Kanten. Algorithmus 5 wird für jeden Knoten aufgerufen. Die Schleife in Zeile 6 wird pro Knoten zweimal durchlaufen, setzt bei jedem Durchlauf alle Besucherzähler zurück und startet `search_topnode`. Da die Besucherzähler über einen gemeinsamen Startwert definiert sind, ist das Zurücksetzen in konstanter Laufzeit möglich.

In Algorithmus 6 wird rekursiv nach der `top_node` gesucht. Dieser Algorithmus wird $\mathcal{O}(N)$-mal von Algorithmus 5 aufgerufen. Nachdem in konstanter Zeit in Zeile 9 der Besucherzähler erhöht wurde, wird in Zeile 10 die Rekursion fortgeführt, wenn der aktuelle Knoten eine Bitoperation ist und über alle seine Verwender einmal besucht wurde. Daher ist die Rekursion durch die Anzahl Kanten beschränkt. Da für jede mögliche `base_node` die Suche nach einer `top_node` gestartet wird, hat die Analyse-Phase eine Laufzeit von $\mathcal{O}(E \cdot N)$.

Bei einer Optimierung wird eine Kante durch eine Konstante ersetzt, daher können maximal $\mathcal{O}(E)$ Optimierungen durchgeführt werden. In Algorithmus 7 wird in Zeile 3 bis 10 geprüft ob eine gegebene Optimierung noch möglich ist. Da die Anzahl Operanden Konstant sind, hat dieser Teil eine Laufzeit von $\mathcal{O}(1)$. Der restliche Ersetzungsvorgang geschieht ebenfalls in konstanter Laufzeit, daher benötigt die Anwendung einer Optimierung $\mathcal{O}(1)$. Die Anwendung aller Optimierungen benötigt $\mathcal{O}(E)$ Laufzeit.
Die Laufzeit des Analysevorgangs überwiegt und somit ist die Gesamtlaufzeit $\mathcal{O}(E \cdot N)$.

4.2. Verkettung

In Kapitel 4.1 wurde das *Shannon*-Verfahren vorgestellt. Der linke Graph von Abbildung 4.9 kann so nicht vom *Shannon*-Verfahren behandelt werden. In diesem Kapitel wird beschrieben, wie Knoten umsortiert werden, damit das *Shannon*-Verfahren häufiger angewandt werden kann.

Beim *Shannon*-Verfahren kann die `top_node` ersetzt werden, wenn auf dem rechten Pfad entweder keine Operation, ein $\sim$ oder ein $\oplus$ mit einer Konstante ist. Falls auf dem rechten Pfad dieselbe Operation wie die `base_node` ist, kann das *Shannon*-Verfahren nicht ausgeführt werden. Als `base_node` kommen nur `&` und `|` in Frage. Falls auf dem rechten Pfad dieselbe Operation auftaucht, kann das Assoziativgesetz angewandt werden, sodass auf dem rechten Pfad keine störende Operation mehr vorhanden ist. Dieses Verfahren wird Verkettung (*Chaining*) genannt. Das Ziel ist es, Variablen und nicht-Bitoperationen möglichst weit nach unten und gleichzeitig Bitoperationen möglichst weit nach oben zu bekommen. Abbildung 4.9 veranschaulicht dieses Vorgehen.

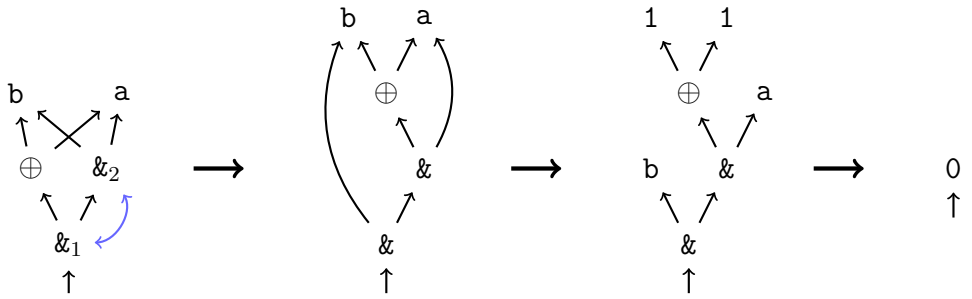


Abbildung 4.9.: Anwendung von Verkettung, damit das *Shannon*-Verfahren angewandt werden kann.

In Abbildung 4.9 ist `a` eine mögliche `top_node`. Als `base_node` kommt nur $\&_1$ in Frage. Auf dem linken Pfad ist ein $\oplus$, allerdings ohne Konstante, und auf dem rechten Pfad unterbricht $\&_2$ den direkten Pfad. Somit gibt es zu `a` keine mögliche `base_node`. Die andere mögliche `top_node` ist `b`. Die Argumentation ist hier analog zu `a` und somit existiert hier auch keine `base_node`. Das *Shannon*-Verfahren kann in diesem Graph somit nicht angewendet werden.

Nun werden nacheinander alle `&` und `|` Operationen betrachtet. Falls zwei aufeinander folgende Operationen gleich sind, werden die Bitoperationen nach oben geschoben. In diesem Beispiel wird das Assoziativgesetz auf die beiden `&` angewandt. Seien `g` und `f` Bitoperationen und `a` nicht. So wird bei `g & (f & a)`, `g` mit `a` vertauscht. Bei `g & (a & b)`, kann wahlweise `g` mit `a` oder `b` vertauscht werden. Bei `a & (f & a)` wird nichts vertauscht.

Gegebenenfalls muss dieses Verfahren mehrmals angewendet werden. Ist diese Umordnung vollzogen kann das *Shannon*-Verfahren angewandt werden. Dieser ersetzt die Operanden von $\oplus$ einmal von `a` und einmal von `b` aus durch entsprechende Konstanten. Schließlich können lokale Optimierungen den Ausdruck zu `0` vereinfachen. Da die Verkettung den Graph für das *Shannon*-Verfahren vorbereitet, muss sie vor dem *Shannon*-Verfahren ausgeführt werden.

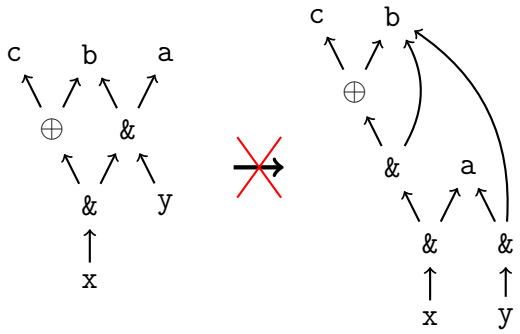


Abbildung 4.10.: Eine Mehrfachverwendung verhindert die Anwendung der Verkettung und somit auch des *Shannon*-Verfahrens.

In Abbildung 4.10 wird das obere `&` von `y` verwendet und hat somit mehrere Verwender. Daher kann die Verkettung hier nicht angewandt werden. Es ist denkbar die Mehrfachverwendung durch Duplikation aufzulösen. Zu diesem Zeitpunkt ist allerdings noch nicht bekannt ob das `b` später mit dem *Shannon*-Verfahren ersetzt werden kann oder nicht. Ohne das sicher zu wissen, verlängert die Duplikation eher die Anzahl Operationen, als sie zu verringern. In dieser Arbeit wurde aus Effizienzgründen auf diese Optimierung verzichtet.

4.3. Ersetzen bekannter Äquivalenzen

Beim *Shannon*-Verfahren wird die Tatsache genutzt, dass ein Teilgraph nur relevant ist, wenn ein Knoten einen bestimmten Wert hat. Dieses Vorgehen kann auf Bedingungen erweitert werden.

$$\texttt{f(a, b) \& (a} \oplus \texttt{b)}$$

`f` ist genau dann relevant, wenn $\texttt{a} \oplus \texttt{b} == 1$ gilt. Das ist genau dann der Fall, wenn $\sim\texttt{a} == \texttt{b}$ bzw. $\texttt{a} == \sim\texttt{b}$. Also kann `f(a, b)` durch `f(a,` $\sim$`a)` oder `f(`$\sim$`b, b)` ersetzt werden. Somit gilt:

$$\texttt{f(a, b) \& (a} \oplus \texttt{b)} \rightarrow \texttt{f(a, } \sim\texttt{a) \& (a} \oplus \texttt{b)}$$

$$\texttt{f(a, b) \& (a} \oplus \texttt{b)} \rightarrow \texttt{f(}\sim\texttt{b, b) \& (a} \oplus \texttt{b).}$$

Bei `|` ist `f` nur relevant, falls der andere Term `0` ist, also $\texttt{a} == \texttt{b}$ gilt. Somit kann wie folgt ersetzt werden:

$$\texttt{f(a, b) | (a} \oplus \texttt{b)} \rightarrow \texttt{f(a, a) | (a} \oplus \texttt{b)}$$

$$\texttt{f(a, b) | (a} \oplus \texttt{b)} \rightarrow \texttt{f(b, b) | (a} \oplus \texttt{b).}$$

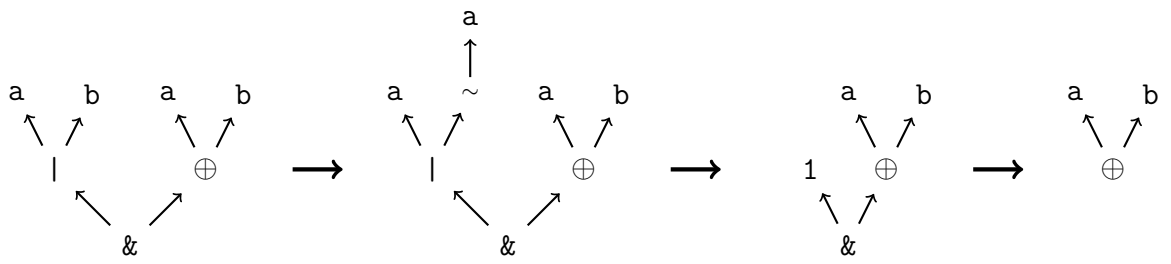


Abbildung 4.11.: Hier kann wahlweise das linke `a` durch $\sim$`b` oder das linke `b` durch $\sim$`a` ersetzt werden. Ist das geschehen, können lokale Optimierungen den Graph weiter vereinfachen.

In Abbildung 4.11 kann $\texttt{a} \mid \texttt{b}$ als `f(a, b)` betrachtet werden. So wird im ersten Schritt auf der linken Seite `b` durch $\sim$`a` ersetzt.

$$\texttt{(a | b) \& (a} \oplus \texttt{b)} \rightarrow \texttt{(a | } \sim\texttt{a) \& (a} \oplus \texttt{b)}$$

Im zweiten und dritten Schritt werden lokale Optimierungen angewandt.

Beim *Shannon*-Verfahren ist es möglich, dass auf dem rechten Pfad ein ∼ oder ⊕ auftaucht. Es ist denkbar, dass auch die Bedingung negiert sein kann.

```
f(a, b) & ∼(a ⊕ b)
```

`f` wird nur betrachtet falls `∼(a⊕b)==1` gilt. Das ist genau dann der Fall wenn `(a⊕b)==0` gilt und somit, wenn `a==b` ist. Folgende zwei Substitutionen sind möglich:

```
f(a, b) & ∼(a ⊕ b) → f(a, a) & ∼(a ⊕ b)

f(a, b) & ∼(a ⊕ b) → f(b, b) & ∼(a ⊕ b).
```

Bei `|` wird `f` nur betrachtet wenn `a==∼b` bzw. `∼a==b` gilt.

```
f(a, b) | ∼(a ⊕ b) → f(a, ∼a) | ∼(a ⊕ b)

f(a, b) | ∼(a ⊕ b) → f(∼b, b) | ∼(a ⊕ b)
```

Die Normalisierungen, die `f(a,b)` durch `f(a,∼a)` ersetzen, erhöhen die Anzahl an Operationen. Allerdings wird eine Mehrfachverwendung eingespart. Da `f` nur aus Bitoperationen besteht, ist die Wahrscheinlichkeit hoch, dass so eine weitere Optimierung möglich ist. Auch bei dieser Normalisierung ist es möglich, dass nicht nur ein ∼, sondern auch ein ⊕ mit einer Konstanten (`C`) auf der rechten Seite steht.

```
f(a, b) & ((a ⊕ b) ⊕ C) → f(a, a⊕∼C) & ((a ⊕ b) ⊕ C)

f(a, b) & ((a ⊕ b) ⊕ C) → f(b⊕∼C, b) & ((a ⊕ b) ⊕ C)

f(a, b) | ((a ⊕ b) ⊕ C) → f(a, a⊕C) | ((a ⊕ b) ⊕ C)

f(a, b) | ((a ⊕ b) ⊕ C) → f(b⊕C, b) | ((a ⊕ b) ⊕ C)
```

Dabei kann `∼C` zur Compilezeit berechnet werden. Unter der Annahme, dass ⊕ mit einer Konstanten die Optimierung schwieriger macht als mit einem ∼, sollte auf diese Normalisierung verzichtet werden.

4.4. Entfernen von Redundanzen in Partitionen gleicher Operationen

In diesem Kapitel werden verschiedene Vorgehensweisen vorgestellt, wie Mehrfachverwendungen in Teilausdrücken gleicher Operationen effizient entfernt werden können. Es werden auch Möglichkeiten angesprochen, wie die Aufteilung in Partitionen genutzt werden kann, um partitionsübergreifend Redundanzen zu erkennen und zu beheben. Dieses Verfahren wird *Setsort* genannt. Zuletzt werden in diesem Kapitel verschiedene Verfahren vorgestellt, wie der Graph effizient neu gebaut werden kann.

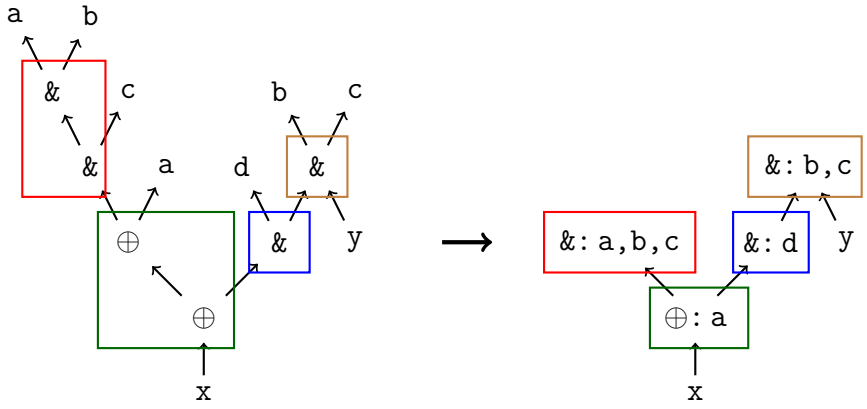


Abbildung 4.12.: Aufteilung eines Graphs in Partitionen.

In Abbildung 4.12 ist ein Graph und seine Einteilung in Partitionen zu sehen. Dabei ist zu beachten, dass eine Operation, welche mehrere Verwender hat, genau dann in die Partition ihrer Verwender aufgenommen wird, wenn alle Verwender in der selben Partition sind. Das verhindert, dass Teilterme verfälscht werden. Zwei Operationen gelten als gleich, wenn die Operation an sich und ihr Modus übereinstimmen. So werden Konflikte verschiedener Modi verhindert. Falls eine Variable mehrmals in einem Set landet, wird unterschiedlich verfahren: Bei arithmetischen Ausdrücken werden sie als Polynome zusammengefasst (siehe Kapitel 4.4.1), dabei wird die Subtraktion sowie die skalare Multiplikation, wie eine Addition behandelt. Bei Bitoperationen löschen sich gegebenenfalls gleiche oder zueinander inverse Variablen aus, dazu mehr in Kapitel 4.4.3.

4.4.1. Arithmetische Ausdrücke

Frailey [5] empfiehlt, Ausdrücke wie `a - b + a` in der Form `a + (-b) + a` darzustellen, da im Gegensatz zur Subtraktion die Addition assoziativ ist. Weiter werden gleiche Variablen als skalare Multiplikation `2·a + (-b)` zusammengefasst. In einer späteren Phase

kann *Strength Reduction* [4] angewandt werden, um die Multiplikation durch einfachere Operationen zu ersetzen. In unserem Beispiel wird `2·a` durch `a + a` oder `a << 1` ersetzt.

Bei arithmetischen Ausdrücken ist es praktisch, alle Vorkommen einer Variablen zu kennen. So ist es möglich, Mehrfachverwendungen schnell zu erkennen und zu optimieren. Da hier nur Addition, Subtraktion sowie skalare Multiplikation betrachtet werden, sind alle anderen Teilterme, die nicht aus diesen Operationen bestehen, als Variablen anzusehen. Es sei zu beachten, dass alle Operationen auf ganzen Zahlen agieren. Gleitkommaarithmetik wird hier nicht optimiert. Gewollt ist eine Umformung wie beispielsweise:

$$\texttt{(3·a + b) - (a - (b - c))} \rightarrow \texttt{((a + b) << 1) - c.}$$

Um das zu erreichen, werden die Terme in Partitionen unterteilt. Wobei jeweils gespeichert wird, wie oft eine Variable in der jeweiligen Partition vorhanden ist. Bei skalaren Multiplikationen wird der Skalar eingerechnet.

$$\texttt{(3·a + b) - (a - (b - c))} \rightarrow \texttt{2·a + 2·b + (-1)·c}$$

Bei Subtraktionen muss die Negation des rechten Verwenders beachtet werden. Im daraus folgenden Term werden alle Variablen genau einmal verwendet. Allerdings gehen zunächst gemeinsame Teilausdrücke verloren. Im Folgenden werden verschiedene Möglichkeiten aufgezeigt diese Teilausdrücke wieder zu erlangen und sie zu optimieren.

Betragsmäßig gleiche Faktoren können zusammengezogen und der gemeinsame Faktor ausgeklammert werden. So wird der obige Term zu `2·(a + b) + (-1)·c`. Um die endgültige Form zu erhalten, werden im Gesamtausdruck sowie in allen Teilausdrücken negative Teilausdrücke nach dem ersten positiven Teilausdruck gestellt. Falls kein positiver Teilausdruck vorhanden ist, wird ein Minus vor den ersten Teilterm gesetzt. Nun kann jedes folgende `x + -n·y` durch `x - n·y` ersetzt werden. Bei negativen Faktoren zieht *Strength Reduction* ein Minus aus dem Faktor, was für mehr Operationen sorgt.

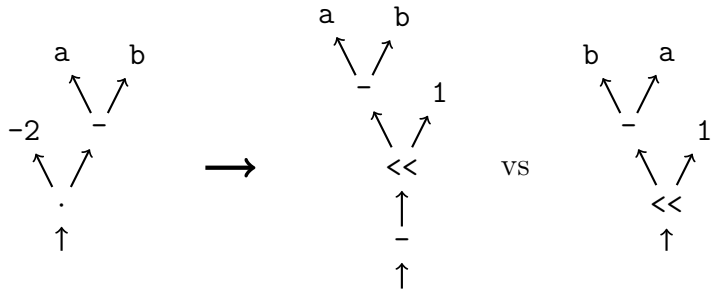


Abbildung 4.13.: Ein negativer Faktor sorgt für eine zusätzliche Operation nach *Strength Reduction* und ist somit wenn möglich zu vermeiden.

In Abbildung 4.13 benötigt der mittlere Term drei Operationen, während der rechte zwei benötigt. Somit sind negative Faktoren zu vermeiden. Der Term `2·(a+b)+(-1)·c` wird zu `2·(a+b)-c` beziehungsweise nach *Strength Reduction* zu `((a+b)<<1)-c`.

```
2·c - 3·(b - (a - c)) → 3·(a - b) - c
```

Hier wird veranschaulicht, dass negative Variablen sowohl in der Klammer (`b`), als auch außen (`c`), nach hinten gezogen werden.

4.4.2. Ersetzen komplexer Operationen durch Einfachere

In diesem Kapitel wird genauer auf *Strength Reduction* [4] eingegangen. Es werden weitere mögliche Erweiterungen vorgestellt, wie das Neubauen nach *Setsort* verbessert werden kann.

Falls der Skalar eine Zweierpotenz ist, kann die Multiplikation durch eine einzige *shift*-Operation ersetzt werden: `2·a` → `a<<1`. Falls das nicht der Fall ist, sind weitere Operationen nötig: `3·a` → `(a<<1)+a` oder `(a<<2)-a`. Es ist also von Vorteil, Zweierpotenzen als Faktoren zu nutzen.

```
2·(a + b) - 3·c → 2·(a + b - c) - c
```

Strength Reduction kann `2·(a+b)-3·c` durch `((a+b)<<1)-((c<<1)+c)` ersetzen, so werden fünf einfache Operationen benötigt. Bei `2·(a+b-c)-c` gibt es nur Zweierpotenzen als Faktor. Hier werden vier einfache Operationen benötigt. Es bietet sich also an, Faktoren, welche keine Zweierpotenz sind, zu Faktoren, welche Zweierpotenzen sind, hinzuzufügen. Das spart Operationen, sorgt allerdings auch dafür, dass es Mehrfachverwendungen gibt.

Faktor	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Ops.	0	1	2	1	2	3	2	1	2	3	4	3	4	3	2	1	2	3	4	3

Tabelle 4.1.: Anzahl einfacher Operationen welche für eine Multiplikation mit einem Faktor nach *Strength Reduction* benötigt werden

In Tabelle 4.1 ist die Abhängigkeit der Anzahl einfacher Operationen zu verschiedenen Faktoren aufgelistet. Es ist vorteilhaft, *komplizierte* Faktoren zusammenzufassen oder in andere leichtere zu unterteilen.

$$\texttt{11·a + 13·b + 19·c} \rightarrow \texttt{11·(a + b + c) + 2·b + 8·c}$$

Der erste Ausdruck benötigt jeweils vier Operationen für 11, 13 und 19, sowie zwei für die Additionen, insgesamt also 14 Operationen. Der zweite Ausdruck benötigt nur 10 Operationen. In diesem Beispiel ist die Auswirkung sehr deutlich, da $19 - 11 = 8$ und $13 - 11 = 2$ Zweierpotenzen sind und diese nur eine Operation benötigen. Das Problem an diesem Vorgehen ist, dass Variablen mehrmals vorkommen und so für mehr Mehrfachverwendungen sorgen.

Eine andere Möglichkeit ist es zu einem Faktor einen weiteren Faktor zu finden, welcher ein Vielfaches des Ersten ist.

$$\texttt{10·a + 20·b} \rightarrow \texttt{10·(a + 2·b)}$$

Der erste Term benötigt $3 + 1 + 3 = 7$ Operationen, der zweite $3 + 1 + 1 = 5$. Das spart zwei Operationen, ohne weitere Mehrfachverwendungen zu erhalten.

4.4.3. Bitoperationen

In diesem Kapitel wird genauer auf die Möglichkeiten von *Setsort* auf Bitoperationen eingegangen.

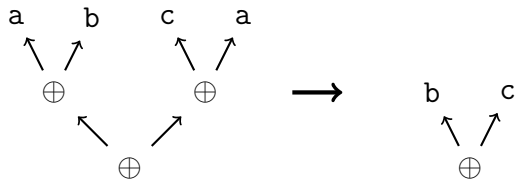


Abbildung 4.14.: Beispiel zur Anwendung von Setsort auf einen Term aus $\oplus$

Beim Einfügen einer Operation in eine Partition wird überprüft, ob sie selbst oder ihr Inverses bereits in der Partition liegt. Ist das der Fall, schlägt eine Optimierung an. Im Fall von $\oplus$ heben sich zwei gleiche Operanden auf und zwei zueinander Inverse werden zu einer 1. In Abbildung 4.14 wird `a` zweimal in dieselbe Partition eingefügt und somit beide Vorkommen entfernt. Bei `&` wird `a & a` $\rightarrow$ `a` und `a & ~a` verwandelt die ganze Partition in eine `0` (siehe Abbildung 4.15). Bei `|` gilt `a | a` $\rightarrow$ `a` und `a | ~a` ermöglicht es die ganze Partition durch eine `1` zu ersetzen.

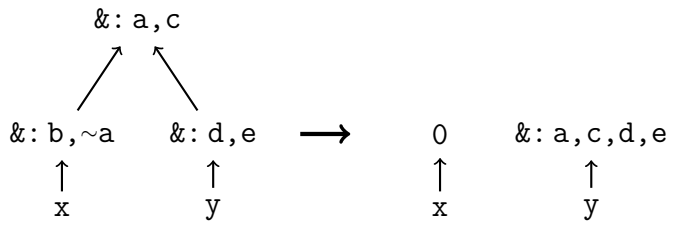


Abbildung 4.15.: Elimination, da inverses Element in Operand vorhanden ist

Wenn der Graph in Partitionen aufgeteilt ist, können partitionsübergreifende Optimierungen angewandt werden. In Abbildung 4.16 hat die obere $\oplus$-Partition mehrere $\oplus$-Partitionen als Verwender.

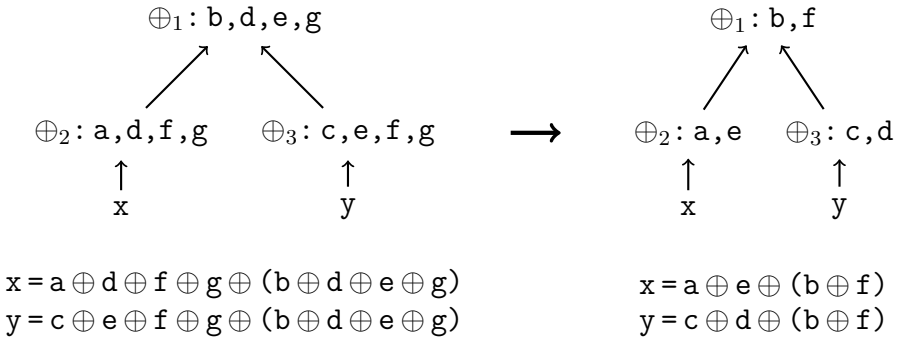


Abbildung 4.16.: Elimination, da inverses Element in Operand vorhanden ist

Da `f` in allen Verwendern vorhanden ist, kann es aus den Verwendern herausgezogen und in den Operand eingefügt werden. Da `g` sowohl in $\oplus_2$ als auch in $\oplus_3$ vorhanden ist, kann `g` aus $\oplus_2$ und $\oplus_3$ entfernt werden und in $\oplus_1$ eingefügt werden. Da `g` allerdings in $\oplus_1$ vorhanden ist, gilt $\texttt{g} \oplus \texttt{g} == \texttt{0}$ und wird somit aus $\oplus_1$ entfernt. Da `e` zweimal in `y` enthalten ist, kann es aus `y` entfernt werden, darf aber nicht aus `x` entfernt werden. Also wird `e` aus $\oplus_1$ und $\oplus_3$ entfernt und in $\oplus_2$ eingefügt. Analog verhält es sich mit `d`. Diese Optimierungen können als Mengen-Operationen dargestellt werden:

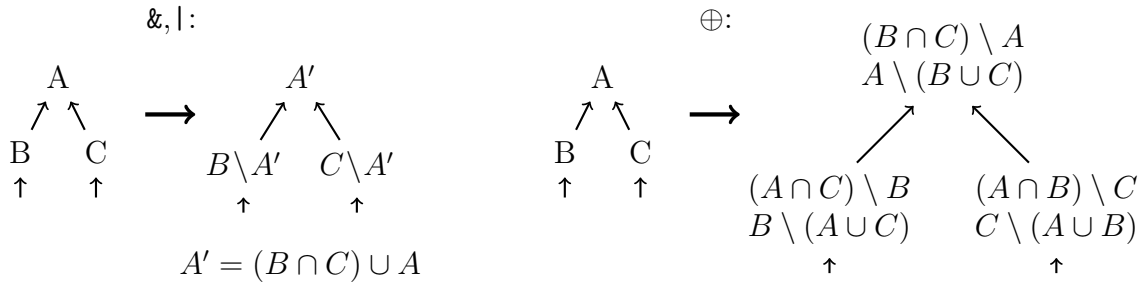


Abbildung 4.17.: Mengen Operationen

Bei den Mengenoperationen aus Abbildung 4.17, muss ∼ gesondert behandelt werden. Bei `&` und `|` kann es einen Zweig komplett durch eine Konstante ersetzen. Bei ⊕ ist es sinnvoll ∼ als `1` darzustellen, da ∼`a` == `a` ⊕ `1` gilt. Diese `1`en lassen sich mit den gegebenen Mengenregeln handhaben.

Falls ein Verwender nicht vom selben Typ wie die obere Partition ist, kann man sich eine leere Partition auf der Verwenderkante vorstellen. Diese Operationen können auf N Verwender übertragen werden. Seien $X_i, i \in I$ die Mengen, wobei $i = 0$ die obere Menge ist. Bei `&` und `|` wird A' zu $(\bigcap_{i \in I \setminus 0} X_i) \cup X_0$. Bei ⊕ wird jede Menge X_j zu $((\bigcap_{i \in I \setminus j} X_i) \setminus X_j) \cup (X_j \setminus (\bigcup_{i \in I \setminus j} X_i))$.

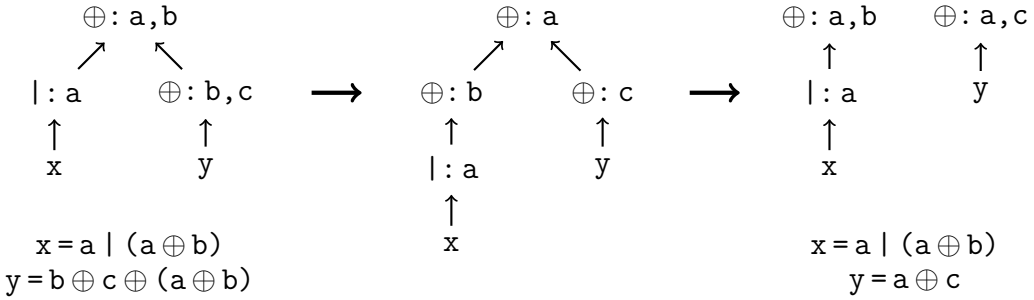


Abbildung 4.18.: Anwendung von Mengenarithmetik bei Verwenderpartition eines anderen Typs

In Abbildung 4.18 ist die `|`-Partition keine ⊕-Partition, aus diesem Grund stellt man sich hier eine leere Partition vor. Nun finden die Mengenoperationen das `b`, welches im rechten Verwender, sowie in der oberen Partition vorkommt und ziehen dieses in den linken Verwender (die bisher leere Partition).

Im mittleren Bild fällt auf, dass die obere Partition nur noch eine Variable beinhaltet und keine anderen Operanden hat. In diesem Fall kann `a` in alle ihre Verwender gezogen werden und die Partition aufgelöst werden. Im rechten Graph ist *Setsort* fertig. Es konnte eine Operation eingespart werden.

Nun liegt die Frage nahe, ob im linken Teilterm `x`, das *Shannon*-Verfahren angewandt werden kann. Das ist nicht möglich, da `a` ⊕ `b` zwei Verwender in unterschiedlichen Partitionen hat. Aus dem selben Grund kann auch das einfache *Setsort* nicht die Mehrfachverwendung von `b` in `y` entfernen, da `a` ⊕ `b` mehrere Verwender hat und somit in einer anderen Partition liegt. Allerdings kann nach den Mengenoperationen Shannon oder eine lokale Optimierung angewandt werden, um `a | (a` ⊕ `b)` durch `a | b` zu ersetzen. Diese Optimierungen sind effektiv, um Rechenregeln auf bereits bekannte Mehrfachverwendungen anzuwenden. Die Schwierigkeit ist dabei zum einen, herauszufinden, wann sich eine Optimierung lohnt, und zum anderen, wie das anschließende neu aufbauen der Partitionen implementiert wird.

5. Evaluation

In den vorherigen Kapiteln wurden verschiedene Lösungsansätze präsentiert um Terme aus Bitoperationen oder arithmetischen Operationen zu normalisieren. In diesem Kapitel soll evaluiert werden, inwiefern eine Verbesserung eingetreten ist. Dafür werden zunächst in Kapitel 5.1 die verschiedenen Testverfahren vorgestellt. Anschließend werden in Kapitel 5.2 die Ergebnisse verglichen und kritisch betrachtet.

5.1. Testverfahren

Optgen [3] erstellt automatisch lokale Optimierungen, dabei konnten einige Optimierungen nicht zu einfachen Regeln zusammen gefasst werden. Ziel dieser Arbeit ist es, möglichst einfache und effiziente Wege zu finden, diese Regeln zu vereinfachen. Es gibt sowohl Tests, die zeigen, ob etwas optimiert werden soll, als auch Tests, bei denen eine Optimierung zu mehr Operationen führen würde und daher nicht optimiert werden sollten. Die Tests bestehen aus Variablen, $\oplus$, $\sim$, `&`, `|`, `+`, $\cdot$, unäres, als auch binäres `-`. Diese wurden in arithmetische Operationen und Bitoperationen unterteilt. Mischungen aus beiden Operatortypen wurden in dieser Arbeit nicht behandelt.

Für das *Shannon*-Verfahren werden Tests mit Bitoperationen genutzt, wobei drei bis fünf Operationen zum Einsatz kommen. Die Operanden werden als Argumente an die Testfunktionen übergeben und das Ergebnis in einer globalen Variablen gespeichert, somit ist der Test von anderen Optimierungen unabhängig. Diese Testfälle werden kompiliert, Assemblercode ausgegeben und die Anzahl Operationen verglichen. Falls ein Test mehr als die Anzahl Operationen, die *Optgen* herausgefunden hat, ausgibt, ist dieser als fehlgeschlagen anzusehen. Nun werden Möglichkeiten gesucht, mit möglichst wenig Aufwand die Anzahl der übrigen Testfälle zu minimieren. Um die Korrektheit des produzierten Codes zu gewährleisten, wird die *firm-testsuite* und *Csmith* [11] in Verbindung mit *C-Reduce* [8] genutzt.

Das *Shannon*-Verfahren greift ausschließlich bei Bitoperationen, weshalb hier gezielt Tests mit Bitoperationen verwendet wurden. Bei *Setsort* werden zusätzlich ausschließlich arithmetische Tests miteinbezogen. Da bei *Setsort* Mehrfachverwendungen aktiv behandelt werden können, haben diese Tests mehrere globale Variablen als Rückgabe.

5.2. Ergebnisse

Mit dem *Shannon*-Verfahren konnten 82% der Testfälle, die aus drei Bitoperationen bestehen optimiert werden, bei den Testfällen mit 4 Bitoperationen werden 51% und bei den Testfällen mit 5 Bitoperationen 37% der Testfälle optimiert. Bei einer größeren Zahl an Operationen kommen mehr nicht-triviale Optimierung hinzu, daher lässt die relative Erfolgsquote bei mehr Operationen nach. So wurden bei den Tests mit drei Bitoperationen 70 optimiert, bei vier Bitoperationen 533 von 1055 und bei fünf Bitoperationen 3347 von 9150. Durch Vorbereitung von *Verkettung* konnten bei den Tests mit fünf Bitoperationen weitere 953 Optimierungen durchgeführt werden. Das Ersetzen bekannter Äquivalenzen konnte zusätzlich 623 Fälle optimieren.

	Anzahl Operationen		
	3	4	5
vorher	85	1 055	9 150
Shannon	15	522	5 803
Verkettung	4	360	4 850
bekannte Äquivalenzen	0	306	4 227

Tabelle 5.1.: Anzahl verbleibender Optimierungsmöglichkeiten bei Testfällen mit 3 bis 5 Bitoperationen. In jeder Zeile kommt eine Normalisierung hinzu.

	Anzahl Operationen							
	32 Bit				64 Bit			
	3	4	5	5*	3	4	5	5*
cparser (master)	85	1 055	9 484	872	85	1 055	9 150	1 145
cparser (normalization)	**0**	**306**	**4 555**	**400**	**0**	**306**	**4 227**	**514**
GCC 4.9.1	68	995	9 509	392	62	969	8 937	178
GCC 5.1	68	989	9 444	392	62	961	8 927	178
Clang 3.5	60	937	9 097	740	60	935	8 731	679
Clang 3.6	46	915	9 082	683	46	913	8 698	683

Tabelle 5.2.: In der Tabelle sind die verbleibenden Optimierungsmöglichkeiten abgebildet. Sie sind unterteilt in 32 Bit und 64 Bit. In den Spalten 3 bis 5 sind Testfälle mit 3 bis 5 Bitoperationen abgebildet. Die Spalte 5* besteht aus Testfällen mit 5 arithmetischen Operationen.

In Tabelle 5.2 ist ein Vergleich verschiedener Compiler abgebildet. Dank der in dieser Arbeit beschriebenen Normalisierungen konnten die erfolgreichen Optimierungen auf Bitoperationen deutlich erhöht werden. In diesen Spalten schneidet der *cparser* deutlich

besser ab als die Vergleichscompiler. Bei arithmetischen Ausdrücken weißt der *cparser* ein deutliches Defizit an Optimierungen auf. Diese konnten durch *Setsort* erheblich verbessert werden. Bei arithmetischen Operationen und 64 Bit kann der *cparser* immer noch erheblich weniger Optimierungen anwenden als *GCC*. Da *GCC 5.1* im Verlauf dieser Arbeit neu erschienen ist, wurde dieser mit in die Liste der Compiler aufgenommen. Dieser weißt allerdings bei Normalisierung und lokalen Optimierungen nur geringe Verbesserungen gegenüber der Version 4.9.1 auf.

In Tabelle 5.3 sind die Ergebnisse von SPEC CINT2000 abgebildet. Da bei 186.crafty viele Bitoperationen vorkommen, sind die Auswirkungen dort am größten.

Testprogramm	ohne Optimierung	mit Optimierung	Δ (absolut)	Δ (relativ)
164.gzip	286 624 309 168	286 624 309 143	-25	-0.000000%
175.vpr	202 915 390 153	202 915 390 153	0	0.000000%
176.gcc	149 040 629 602	149 028 369 222	-12 260 380	-0.008226%
181.mcf	47 711 626 521	47 711 626 509	-12	-0.000000%
186.crafty	185 764 407 760	185 575 236 123	-189 171 637	-0.101834%
197.parser	291 126 093 684	291 125 732 729	-360 955	-0.000124%
253.perlbmk	1 107 580 056	1 107 575 061	-4 995	-0.000451%
254.gap	218 144 689 882	218 143 617 367	-1 072 515	-0.000492%
255.vortex	316 348 871 452	316 348 871 425	-27	-0.000000%
256.bzip2	279 390 352 089	279 390 915 422	563 333	0.000202%
300.twolf	300 493 433 664	300 493 444 439	10 775	0.000004%
Durchschnitt			-18 390 585	-0.010083%

Tabelle 5.3.: Auswirkung des *Shannon*-Verfahrens und *Setsort* auf die Anzahl Operationen bei SPEC CINT2000 Testprogrammen.

6. Fazit und Ausblick

In dieser Arbeit wurden verschiedene Verfahren vorgestellt, um Terme aus Bitoperationen oder arithmetischen Operationen zu normalisieren, sodass lokale Optimierungen weitere Operationen entfernen konnten. Die Ergebnisse von *186.crafty* zeigen, dass das *Shannon*-Verfahren eine sehr effiziente Möglichkeit ist, um ausschließlich aus Bitoperationen bestehende Terme zu vereinfachen. Allerdings ist das Verfahren an eine bestimmte Struktur gebunden. Durch passende Restrukturierung kann dafür gesorgt werden, dass diese Struktur erreicht wird und die Terme daraufhin mit dem *Shannon*-Verfahren optimiert werden können. Dies wurde mit Verkettung der Operationen erreicht. Dabei blieb die Einschränkung, dass es keine Mehrfachverwendungen geben darf. Beispielsweise kann der Ausdruck

$$\texttt{f(a, b) \& \sim(a | b)} \rightarrow \texttt{f(0, 0) \& \sim(a | b)}$$

ersetzt werden, obwohl `~(a | b)` einen weiteren Verwender hat. Allerdings sind diese Fälle sehr selten und schwer zu erkennen. Das gleiche Verhalten existiert bei Mehrfachverwendungen in `f(a, b)`. Um zu verhindern, dass diese Mehrfachverwendungen das *Shannon*-Verfahren blockieren, ist es möglich `f(a, b)` zu duplizieren. Abbildung 6.1 zeigt ein Beispiel für diesen Sachverhalt.

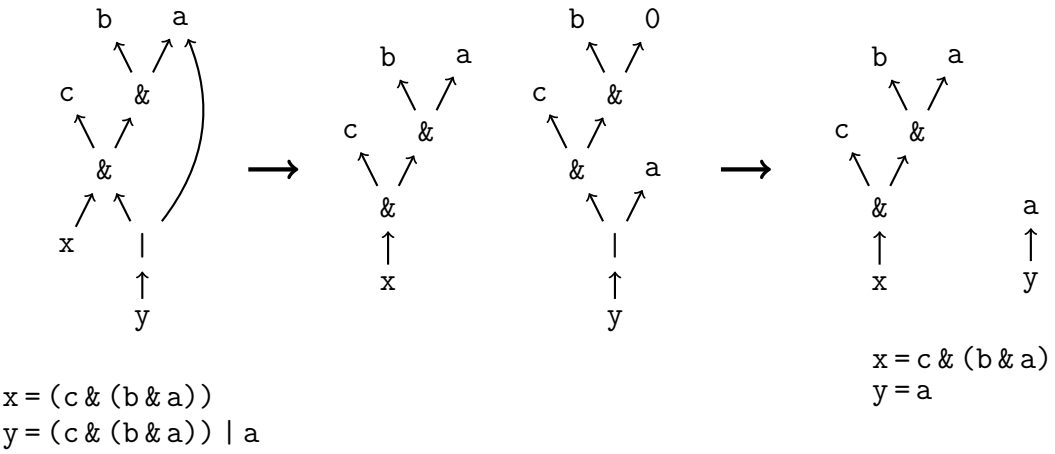


Abbildung 6.1.: Term, der trotz Mehrfachverwendung vereinfacht werden kann.

Der Teilausdruck `c & (b & a)` in Abbildung 6.1 hat zwei Verwender. Zum einen `x` und zum anderen das `|`, welches wiederum von `y` verwendet wird. Wenn `x` keinen Teilausdruck aus `y` verwenden würde, könnte das *Shannon*-Verfahren das `a` durch eine `0` ersetzen. Um `x` nicht zu verlieren, wird im ersten Schritt die Mehrfachverwendung durch Duplizieren des Teilausdrucks aufgehoben und nach Anwendung des *Shannon*-Verfahrens das `a` durch eine `0` ersetzt. Das sorgt zunächst für mehr Operationen, nachdem lokale Optimierungen angewandt wurden, kann aber eine Operation und ein Zwischenergebnis eingespart werden. Die Wahrscheinlichkeit ist höher, dass durch dieses Vorgehen mehr Operationen entstehen als Operationen eingespart werden können. Außerdem ist zu diesem Zeitpunkt noch nicht klar ob, in dem mehrfach verwendeten Teilausdruck `a` mehrmals vorkommt. So ist es möglich, dass das `Shannon`-Verfahren gar nichts optimieren kann. Dieses Problem könnte mit einer Regel für *Setsort* gelöst werden.

Bei Termen ohne mehrfach verwendete Teilausdrücke ist das *Shannon*-Verfahren ein sehr effizientes Verfahren, um Bitoperationen zu vereinfachen. Die Aufteilung eines Terms in Partitionen, derselben Operation (*Setsort*) bietet viel Potential, um unabhängig von der Assoziativität agieren zu können. Es ist möglich, dieses Verfahren auf Multiplikationen zu erweitern, sodass $a \cdot b \cdot a \cdot b$ zu $a^2 \cdot b^2$ normalisiert werden kann. So kann bei höheren Potenzen linear potenziert werden.

Literaturverzeichnis

[1] T. Bersch. Generierung lokaler Optimierungen, Aug. 2012.

[2] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Softw., Pract. Exper.*, 27(6):701–724, 1997.

[3] S. Buchwald. Optgen: A generator for local optimizations. In B. Franke, editor, *Compiler Construction*, volume 9031 of *Lecture Notes in Computer Science*, pages 171–189. Springer Berlin Heidelberg, 2015.

[4] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, Sept. 2001.

[5] D. J. Frailey. Expression optimization using unary complement operators. *SIGPLAN Not.*, 5(7):67–85, July 1970.

[6] G. Lindenmaier. libFIRM – a library for compiler optimization research implementing FIRM. Technical Report 2002-5, Sept. 2002.

[7] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[8] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. *SIGPLAN Not.*, 47(6):335–346, June 2012.

[9] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1949.

[10] M. Trapp, G. Lindenmaier, and B. Boesler. Documentation of the intermediate representation FIRM. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

Erklärung

Hiermit erkläre ich, Victor Pfautz, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift

A. Sonstiges

A.1. Formelsammlung

In diesem Kapitel sind alle interessanten Umformungen und Optimierungen aufgelistet, die bei dieser Arbeit zum Vorschein kamen. Eine Umordnung spart keine Operationen ein und ist mit „↔“ gekennzeichnet. Eine Optimierung sorgt für weniger Mehrfachverwendungen oder weniger Operationen und ist mit „→“ gekennzeichnet.

A.1.1. Termübergreifende Optimierungen

Shannon-Verfahren:
`a & f(a)` → `a & f(1)`
`a | f(a)` → `a & f(0)`

Verkettung und Shannon-Verfahren:
`(a & b) & f(a, b)` → `(a & b) & f(1, 1)`
`(a | b) | f(a, b)` → `(a | b) | f(0, 0)`

De Morgan und Shannon-Verfahren:
`~(a | b) & f(a, b)` → `~(a | b) & f(0, 0)`
`~(a & b) | f(a, b)` → `~(a & b) | f(1, 1)`

Ersetzen bekannter Äquivalenzen:
`(a ⊕ b) & f(a, b)` ↔ `(a ⊕ b) & f(a, ~a)`
`~(a ⊕ b) & f(a, b)` → `~(a ⊕ b) & f(a, a)`
`(a ⊕ b) | f(a, b)` → `(a ⊕ b) | f(a, a)`
`~(a ⊕ b) | f(a, b)` ↔ `~(a ⊕ b) | f(a, ~a)`

A.1.2. Lokale Optimierungen

Anwendung des Distributivgesetz auf Xor:
(a & b) ⊕ (a & c) → a & (b ⊕ c)
a ⊕ (a & b) ↔ a & ~b
a ⊕ (a | b) ↔ ~a & b
(a | b) ⊕ (a | c) → ~a & (b ⊕ c)

Xor Elimination:
a ⊕ (~a | b) → ~(a & b)
a ⊕ (~a & b) → (a | b)

Gemischt:
(a | b) & ~(b & c) → (a | b) ⊕ (b & c)
(a | b) ⊕ (a & b) → (a ⊕ b)
(a | b) & (~a | c) → b ⊕ (a & (c ⊕ b))
(a & b) | (~a & c) → c ⊕ (a & (c ⊕ b))
(a | b) ⊕ (a & (b | d)) → a ⊕ (b | (a & d))
a ⊕ (~(a & c) & b) → (a ⊕ b) | (c & b)
(c & b) ⊕ ~((a ⊕ b) | c) → ~b ⊕ (a | c)

(a | b) ⊕ (a & b) → a ⊕ b
(a | b | c) ⊕ (a & b & c) → (a ⊕ b) | (a ⊕ c)
(a | b | c | d) ⊕ (a & b & c & d) → (a ⊕ b) | (a ⊕ c) | (a ⊕ d)
usw.