Bachelorarbeit

Optimierung von Kontrollflussgraphen durch Graphersetzungssysteme

Maik Marschner

Matrikelnummer: 3068370

21. August 2016

angefertigt am
Lehrstuhl Graphische Datenverarbeitung
Institut für Mensch-Maschine-Kommunikation
Gottfried Wilhelm Leibniz Universität Hannover

Erstprüfender: Prof. Dr. F.-E. Wolter
Zweitprüfender: Prof. Dr. M. Rohs
Betreuer: M. Sc. M. Klein

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ausschließlich mit Hilfe der angegebenen Quellen und Hilfsmittel angefertigt habe.

Hannover, den 21. August 2016 Maik Marschner

Zusammenfassung

Zur Optimierung von Programmen während der Kompilierung ist häufig eine aufwendige Analyse erforderlich, um den Kontroll- und Datenfluss zu rekonstruieren. Am Welfenlab wird derzeit die Programmierplattform *Buggy* entwickelt, in der Programme durch Kontrollflussgraphen repräsentiert werden. Diese beschreiben den gesamten Kontroll- und Datenfluss des Programms. Dadurch entfällt die semantische Analyse und es ergeben sich zahlreiche Optimierungsmöglichkeiten.

In dieser Arbeit wird ein Optimierer entwickelt, der Graphersetzungssysteme nutzt, um *Buggy*-Kontrollflussgraphen zu optimieren. Mit diesen können Graphen anhand einer Menge von Ersetzungsregeln manipuliert werden. Zur Optimierung wird eine Reihe von Ersetzungsregeln erarbeitet. Mit diesen werden neben grundlegenden Optimierungen wie der Auswertung von Berechnungen mit Konstanten auch Optimierungen auf einer abstrakteren semantischen Ebene ermöglicht. So können unter anderem rekursive Funktionen optimiert werden und Teile eines Programms durch effizientere Implementierungen ersetzt werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Am Welfenlab wird seit einiger Zeit eine neue Programmierplattform namens *Buggy* entwickelt. Diese erlaubt es, Programme in verschiedenen, jeweils für ein bestimmtes Problem geeigneten, Sprachen zu formulieren. Die Programme werden dann in Graphen überführt, aus denen in einem weiteren Schritt kompilierbare Programme in einer anderen Programmiersprache generiert werden. Dadurch ist es insbesondere möglich, Programme, die in verschiedenen Ausgangssprachen implementiert wurden, zu kombinieren.

Bislang wird dabei als Eingabesprache *Lisgy*, eine an *Clojure* angelehnte funktionale Programmiersprache, und als Zielsprache *Go* unterstützt. Weitere Codegenerierer für *C++* und *JavaScript* befinden sich derzeit in Entwicklung. Unabhängig von der Ziel- und den Eingangssprache liegt dabei als Zwischenschritt das vollständige Programm als Graph vor. Die *Buggy*-Graphen bieten somit ein großes Potential für Optimierungen. Allerdings existiert derzeit noch kein Optimierer für *Buggy*-Programme.

Um Programme zu optimieren, wird traditionell aus dem Quelltext ein Kontrollflussgraph erzeugt und anschließend analysiert. Dabei kann die Semantik häufig nicht genau erfasst werden, sodass die Optimierungsmöglichkeiten eingeschränkt sind.

Eine besondere Eigenschaft von *Buggy* ist, dass die aus der Eingabesprache erzeugten Graphen den vollständigen Kontrollfluss des Programms enthalten. Die aufwendige Kontrollflussanalyse entfällt somit und ein Optimierer kann direkt mit den bestehenden Graphen arbeiten.

1.2 Ziele

Im Rahmen dieser Bachelorarbeit soll ein Programm entwickelt werden, das *Buggy*-Programme mithilfe von Graphersetzungssystemen optimiert. Dazu soll direkt mit den von *Buggy* erzeugten Kontrollflussgraphen gearbeitet werden.

Neben grundlegenden Optimierungen wie beispielsweise der Auswertung von Berechnungen mit Konstanten sollen auch abstraktere semantische Strukturen optimiert werden. Dazu sollen Programmteile gefunden und durch äquivalente, effizientere Implementierungen ersetzt werden. Dabei wird auch die Optimierung von Techniken aus der funktionalen Programmierung, die in *Buggy* unterstützt werden, im Fokus stehen.

Ziel der Optimierungen ist in erster Linie, die Ausführungsgeschwindigkeit von *Buggy*-Programmen zu verbessern. Darum soll der entwickelte Optimierer anhand einiger Programmbeispiele evaluiert werden.

1.3 Einordnung der Arbeit

Die Optimierung von Programmen ist eine der aufwendigsten Aufgaben eines Compilers. Der Optimierer der *GNU Compiler Collection* übersetzt das Programm dabei in mehrere Zwischensprachen mit verschiedenen Abstraktionsebenen [Nov06, S. 187]. Auf jeder dieser Ebenen sind unterschiedliche Optimierungen möglich. Zur Analyse des Programms werden während der Optimierung Kontrollflussgraphen gepflegt, auf denen einige der Optimierungen durchgeführt werden [FSFa, Kapitel 12 und 14].

Es existieren bereits Demonstrationen für Optimierer, die aus Graphersetzungssystemen generiert werden [Ass00]. Damit ist es möglich, Graphersetzungsregeln zu spezifizieren, aus denen anschließend ein Optimierer generiert wird. Die aufwendige und fehleranfällige Entwicklung eines Optimierers kann durch diese Abstraktion vereinfacht werden.

Da *Buggy*-Programme ausschließlich als Graphrepräsentation vorliegen, wird dieser Ansatz aufgenommen. Der Optimierer wird in der vorliegenden Arbeit allerdings nicht aus den Graphersetzungsregeln generiert, sondern die Regeln werden manuell implementiert. Eine höhere Abstraktion wird durch die bereitsgestellten Funktionen für die Implementierung der Ersetzungsregeln erreicht.

Ein Anwendungsgebiet von Graphersetzungssystemen ist das *Term Graph Rewriting* [Plu99]. Diese Systeme beschränken sich auf Ersetzungen in Graphen von funktionalen Ausdrücken, um diese in Normalformen umzuwandeln oder auszuwerten. Unter anderem findet das *Term Graph Rewriting* bei der Implementierung von funktionalen Programmiersprachen Anwendung. Für diese Systeme ist bekannt, unter welchen Bedingungen sie terminieren und die Graphen auf vorhersehbare Weise transformieren. Bei den *Buggy*-Graphen handelt es sich nicht um Term-Graphen. Einige Ersetzungsregeln des zu entwickelnden Optimierers werden jedoch ähnlich funktionieren wie die Ersetzungsschritte beim *Term Graph Rewriting*.

Von gängigen Compilern werden zahlreiche Optimierungen unterstützt. Dazu gehört unter anderem auch das Inlining von Funktionen [FSFb, Abschnitt 3.10]. Diese Optimierung funktioniert auch für Lambda-Funktionen in C++. Auch die Auswertung von konstanten Programmteilen und die Entfernung von unerreichbaren Programmteilen kann vom Optimierer durchgeführt werden. Die Optimierungen beschränken sich jedoch auf das zu kompilierende Programm. Bibliotheken werden üblicherweise als externe Funktionen aufgerufen, eine Optimierung zwischen dem Programm und den Bibliotheken findet nicht statt.

Die Optimierung von endrekursiven Funktionen wird ebenfalls häufig von Compilern unterstützt. Dies wird erreicht, indem *Tail Calls* optimiert werden. Dabei wird für den rekursiven Funktionsaufruf kein neuer Platz im Stack belegt, sondern der zuvor genutzte Platz wiederverwendet. In einigen Programmiersprachen wird diese Optimierung vom jeweiligen Standard gefordert, beispielsweise in Scheme [SDF+09]. Diese Optimierungen können in *Buggy* nicht am Kontrollflussgraphen durchgeführt werden. Für die Programmiersprache Lisp wurden bereits Möglichkeiten untersucht, rekursive Funktionen innerhalb der Programmiersprache in iterative Funktionen zu transformieren [Ris73]. Unter anderem werden auch endrekursive Funktionen, dort als *triviale Rekursion* bezeichnet, zu iterativen Funktionen umgeformt.

Kapitel 2

Graphersetzungssysteme

So vielseitig nutzbar wie Graphen sind auch Graphersetzungssysteme. Neben Anwendungen bei der Implementierung und Optimierung von Programmiersprachen finden Graphersetzungssysteme auch in der Bildverarbeitung und im Software Engineering Anwendung [BFG95, S. 21 ff.]. Formal sind Graphersetzungssysteme daher vielfach spezifiziert und untersucht worden. In diesem Abschnitt wird die Theorie der Graphersetzungssysteme untersucht, um eine Grundlage für die spätere Implementierung zu schaffen.

2.1 Definitionen

Ein Graphersetzungssystem ist eine Menge von *Graphersetzungsregeln*. Je nach Art des Graphersetzungssystems sind diese im Detail unterschiedlich definiert. Allgemein können sie als ein Tupel (g_l, g_r, M) definiert werden [Dö95, S. 17]. Dabei ist der Graph g_l die linke Seite, also der Teilgraph, für den die Regel gilt und der Graph g_r ist die rechte Seite der Regel, durch die der gefundene Teilgraph ersetzt wird. M ist eine Menge von Beschreibungen, wie die Regel anzuwenden ist.

Eine Regel $r = (g_l, g_r, M)$ ist auf einen Graphen G *anwendbar*, falls es in G einen zu g_l isomorphen Teilgraphen gibt [Dö95, S. 19]. Eine Regelanwendung besteht daher aus mehreren Schritten. Zunächst wird ein Teilgraph von G gesucht, der isomorph zu g_l ist. Anschließend wird dieser Teilgraph durch eine Kopie von g_r ersetzt. Dabei wird der zu g_l isomorphe Teilgraph entfernt. Zuletzt wird der eingefügte Teilgraph entsprechend der Beschreibungen im M in den Graphen eingebettet.

Prinzipiell werden die Ersetzungsregeln auf den ursprünglichen Graphen angewandt, bis für keine Ersetzungsregel passende Teilgraphen gefunden werden können. Da die Anwendung einer Ersetzungsregel dazu führen kann, dass weitere Regeln anwendbar werden, kann es dabei zu Zykeln und somit zu einer endlosen Kette von Ersetzungen kommen. Ein Graphersetzungssystem heißt *terminierend*, falls für jeden beliebigen Graphen nach einer endlichen Anzahl von angewandten Regeln keine weitere Regel mehr anwendbar ist.

Eine weitere wichtige Eigenschaft eines Graphersetzungssystems ist die *Konfluenz*. Ein Graphersetzungssystem heißt *konfluent*, falls der resultierende Graph unabhängig von den angewandten Ersetzungsregeln und der Reihenfolge, in der diese angewandt werden, ist [Ass00, S. 593]. Da die Reihenfolge der Ersetzungsregeln praktisch nicht klar festgelegt werden kann, stellt die Konfluenz sicher, dass ein Graphersetzungssystem deterministische Graphen erzeugt.

2.2 Arten von Graphersetzungssystemen

Für die verschiedenen Anwendungsbereiche existieren verschiedene Arten von Graphersetzungssystemen. Dabei werden die Graphen und auch die Ersetzungsregeln eingeschränkt, sodass Aussagen über die Konfluenz und Terminierung gemacht werden können.

Term Graph Rewriting

Das *Term Graph Rewriting* findet in der Evaluierung von Funktionstermen Anwendung [Plu99, S. 4]. Dabei werden die Terme in Term-Graphen überführt, die anschließend mit Ersetzungsregeln umgeformt werden. In einem Term-Graphen sind alle Knoten von einem Wurzelknoten aus erreichbar.

Jede Ersetzungsregel enthält auf der linken Seite einen Knoten, der im Graph vorhanden sein muss. Dieser wird zunächst gesucht und weiterhin werden nur Knoten betrachtet, die von diesem aus erreichbar sind. Dadurch ist ein schnelles Matching möglich, das sogar parallelisiert werden kann [Plu99, S. 17 ff.].

Bei den Kontrollflussgraphen in *Buggy* handelt es sich nicht um Term-Graphen. Das Matching von Teilgraphen und die Anwendung der Ersetzungsregeln kann jedoch adaptiert werden.

2.3 Performanz

Ein wesentlicher Schritt bei der Anwendung einer Graphersetzungsregel ist das Erkennen von zu einem gegebenen Graphen isomorphen Teilgraphen. Dieses Problem ist NP-vollständig [Coo71]. Die Kontrollflussgraphen sind jedoch gerichtet und azyklisch, zudem ist jeder Knoten mit verschiedenen Meta-Informationen versehen, somit also *beschriftet*. Das Problem lässt sich auf den Spezialfall des *Labeled Subgraph Matching* einschränken. Bei diesem Problem sind isomorphe Teilgraphen in einem beschrifteten Graphen zu finden. Dafür existieren effiziente Algorithmen, wie zum Beispiel in [Dö95, S. 24 ff.] beschrieben.

Anwendung auf Kontrollflussgraphen

In den zu betrachtenden Kontrollflussgraphen enthält jeder Knoten unter anderem den Typ der Operation, die er durchführt, sowie eine eindeutige ID. Im folgenden ist V die Menge der Knoten und E die Menge der Kanten in einem Kontrollflussgraphen. Die angenommenen Laufzeiten der einzelnen Operationen im Graphen können der Tabelle 2.1 entnommen werden. Insbesondere ist auch der Zugriff auf eine Vorgänger oder Nachfolger eines Knotens in amortisierter konstanter Zeit möglich.

Die Ersetzungsregeln werden so formuliert, dass sie zunächst prüfen, ob ein Knoten zu einem *potenziell passenden* Teilgraphen gehören (dies ist in $O(|V|)$ möglich, da dazu lediglich der Typ der Knoten geprüft werden muss). Von den potenziell passenden Teilgraphen wird anschließend eine nur von der Regel abhängige Anzahl $k < |V|$ an Vorgängern der ausgewählten Knoten betrachtet. Für k existiert eine obere Schranke, sodass sich die Laufzeit für das Matching von Teilgraphen auf insgesamt $O(|V|^2)$ beläuft. Wird zum Zugriff auf die Knoten anhand des Labels eine Hashmap verwendet, ist eine amortisierte Laufzeit von $O(|V|)$ möglich.

Operation	Laufzeit (amortisiert)	Laufzeit (worst case)
Knoten anhand der ID abrufen	$O(1)$	$O(\|V\|)$
Knoten einfügen	$O(1)$	$O(\|V\|)$
Knoten löschen	$O(1)$	$O(\|V\|)$
Kante einfügen	$O(1)$	$O(\|E\|)$
Kante löschen	$O(1)$	$O(\|E\|)$
Vorgänger-/Nachfolgerknoten eines Knotens abrufen	$O(1)$	$O(\|V\|)$
Kante löschen	$O(1)$	$O(\|E\|)$

Tabelle 2.1: *Laufzeiten einiger Operationen im Graphen*

In vielen Regeln ist die rechte Seite der Ersetzungsregel konstant und die Laufzeit der Konstruktion des neuen Teilgraphen unabhängig von der Größe des Graphen. In diesem Fall können die Knoten und Kanten der rechten Seite in $O(|V| + |E|)$ eingefügt werden. Es gibt jedoch Regeln, für die Teilgraphen kopiert werden müssen. Da maximal der gesamte Graph kopiert wird, kann die Laufzeit dafür mit $O(|V| + |E|)$ abgeschätzt werden. Die Laufzeit für die Anwendung einer Ersetzungsregel kann insgesamt unter den obigen Annahmen mit $O((|V|^2 + |E|)$ abgeschätzt werden.

Später werden einige Ersetzungsregeln eingeführt, bei denen g_l rekursiv definiert ist und daher die Größe von g_l nicht im Voraus bekannt ist. Diese werden jedoch ebenfalls lediglich Vorgängerknoten eines Knotens eines potenziell passenden Teilgraphen betrachten. Dies sind maximal $|V|$ Knoten, sodass sich eine Laufzeit von $O(|V|^2)$ für die Erkennung eines Teilgraphen für eine Ersetzungsregel ergibt.

2.4 Terminierung und Konfluenz

Wie zuvor beschrieben, terminiert ein Graphersetzungssystem im Allgemeinen nicht immer, da bei der Anwendung einer Ersetzungsregel der Graph möglicherweise größer wird und Teilgraphen enstehen, auf die weitere Ersetzungsregeln angewandt werden können. Außerdem ist das Graphersetzungssystem möglicherweise nicht konfluent, falls eine Regel einen passenden Teilgraph für eine weitere Regel erzeugt oder zerstört, sodass die Reihenfolge der angewandten Ersetzungsregeln den resultierenden Graphen beeinflusst.

Ein Ansatz, um dieses Problem zu lösen, ist die *Stratifizierung* des Graphersetzungssystems [Ass00, S. 599 ff.]. Dabei wird das Graphersetzungssystem in Gruppen (*Strata*) aufgeteilt, sodass jede Gruppe ein terminierendes, deterministisches Graphersetzungssystem ist. Endlosschleifen bei der Anwendung der Graphersetzungsregeln können entstehen, falls eine Regel einen auf eine weitere Regel passenden Teilgraphen erzeugt. Graphersetzungsregeln, die gegenseitig passende Teilgraphen erzeugen, werden daher verschiedenen Gruppen zugeteilt.

Die Graphersetzungsregeln dieser Gruppen werden in einer festgelegten Reihenfolge angewandt. Sind die einzelnen Gruppen konfluent, folgt daraus die Konfluenz des gesamten Graphersetzungssystems.

Auf diese Weise kann zwar die Terminierung und die Konfluenz sichergestellt werden, das Graphersetzungssystem liefert allerdings nicht zwingend einen optimalen Graphen. Falls etwa eine Ersetzungsregel in der zweiten Gruppe einen Teilgraphen erzeugt, der von einer Ersetzungs-

regel aus der ersten Gruppe erkannt werden könnte, ist dies durch die festgelegte Reihenfolge eventuell nicht mehr möglich. In dieser Hinsicht kann die Stratifizierung als ein heuristisches Verfahren betrachtet werden, das auf Grundlage der implementierten Ersetzungsregeln und deren gegenseitigen Abhängigkeiten konfiguriert werden muss.

Kapitel 3

Überblick über Buggy

Bevor auf die Implementierung des Optimierers und die Ersetzungsregeln eingangen wird, folgt in diesem Abschnitt zunächst ein kurzer Überblick über *Buggy*.

3.1 Aufbau und Funktionsweise

Buggy ist eine modular aufgebaute Programmierplattform. Eine Eingangssprache, zum Beispiel *Lisgy*, wird von einem Programm in einen Kontrollflussgraphen übersetzt, der im folgenden auch als *Buggy*-Graph bezeichnet wird. Über die Eingangssprache gibt es die Möglichkeit, *Compound*-Knoten aus einer Bibliothek zu nutzen. Die Implementierungen dieser Knoten werden bei der Kompilierung in den Graphen eingefügt. Dann enthält der Graph den gesamten Kontroll- und Datenfluss des Programms.

Diese Graphen werden mit Typen versehen und es wird eine Typprüfung durchgeführt. Auf dieser Ebene sind die Typen unabhängig von der Programmiersprache, in der das Programm oder verwendete Komponenten implementiert wurden.

Aus den typisierten Graphen wird kompilierbarer Programmcode erzeugt. Derzeit wird die Programmiersprache *Go* als Zielsprache unterstützt. Dem Codegenerator müssen dazu Implementierungen der verwendeten atomaren Knoten in der Zielsprache zur Verfügung stehen. Dazu existiert eine Komponentenbibliothek, in der die verfügbaren Komponenten und die Implementierungen für verschiedene Programmiersprachen gespeichert werden.

3.2 Struktur der Kontrollflussgraphen

Bei den Kontrollflussgraphen handelt es sich um *labeled port multigraphs*. Formal können diese Graphen als Tupel $G = (V, E, \iota, s, t, l)$ mit den folgenden Eigenschaften definiert werden [AK07, S. 6]:

- V ist eine endliche Menge von Knoten
- $\iota = (\iota_1, \iota_2) : V \to N \times P(P)$ weist jedem Knoten einen Namen und eine Menge an Ports zu, wobei $\iota_1(v) = n$ und $\iota_2(v) = P$ für $\iota(v) = (n, P)$
- $E \subseteq \{((v,p),(u,r)) \in (V \times P)^2 \mid p \in \iota_2(v), r \in \iota_2(u)\}$ ist eine endliche Menge von Kanten
- $s, t : E \to V \times P$ sind Funktionen, die den Startknoten und -port respektive den Endknoten und -port einer Kante angeben

– $l = (l_V, L_E)$ ist die Beschriftungsfunktion. Jedem Knoten $v \in V$ wird eine eindeutige Bezeichnung, ein Namen, und eine Menge von Ein- und Ausgangsports zugewiesen. Jeder Kante $((v,p),(u,r)) \in E$ wird ein Quell- und ein Ziel zugewiesen, $l_E((v,p),(u,r)) = (p,r)$

Diese sehr allgemeine Definition kann für *Buggy*-Kontrollflussgraphen enger gefasst werden. Der Kontrollflussgraph besteht aus atomaren Knoten und *Compound*-Knoten, die aus weiteren Knoten zusammengesetzt sind. Daher wird das Tupel G um eine Funktion $p : V \to V \cup \{\#\}$ erweitert, die für jeden Knoten den Knoten angibt, in dem dieser enthalten ist (oder #, falls sich der Knoten auf der obersten Ebene befindet).

Weiterhin wird zwischen Eingangsports (P_I) und Ausgangsports (P_O) unterschieden, $P = P_I \cup P_O$. Eine Kante $e = ((v,p),(u,r)) \in E$ kann grundsätzlich von einem Ausgangsport zu einem Eingansport verlaufen. In *Compound*-Knoten sind auch Kanten von Eingangs- zu Eingangsknoten, falls $v = p(u)$ und von Ausgangs- zu Ausgangsknoten, falls $u = p(v)$, möglich. Kanten dürfen die Grenzen von *Compound*-Knoten nicht überschreiten: $\forall e = ((v,p),(u,r)) \in E \ : \ p(v) = p(u) \ \vee \ v = p(u) \ \vee \ u = p(v)$.

In *Buggy* gilt außerdem $\forall (v,p) \in (V \times P) \ : \ |\{((u,r),(w,q)) \in E \ | \ w = v, q = p\}| \leq 1$, das heißt, dass jeder Knoten in jedem Port höchstens einen Vorgänger hat.

Compound-Knoten

Die meisten *Compound*-Knoten sind Komponenten aus einer Bibliothek, die *Buggy* bereitstellt. Später werden Nutzer in dieser ihre wiederverwendbaren Komponenten veröffentlichen können. Die *Compound*-Knoten werden außerdem für die Repräsentation von rekursiven Funktionen im Kontrollflussgraphen genutzt. Dazu wird die Implementierung der rekursiven Funktion als *Compound*-Knoten an der Stelle des Aufrufs eingefügt. Ruft sich die Funktion innerhalb dieses Knotens selbst auf, wird dies durch einen Knoten mit dem Namen des *Compound*-Knotens signalisiert. Die Abbildung 3.1 zeigt beispielhaft den Graphen einer rekursiven Funktion.

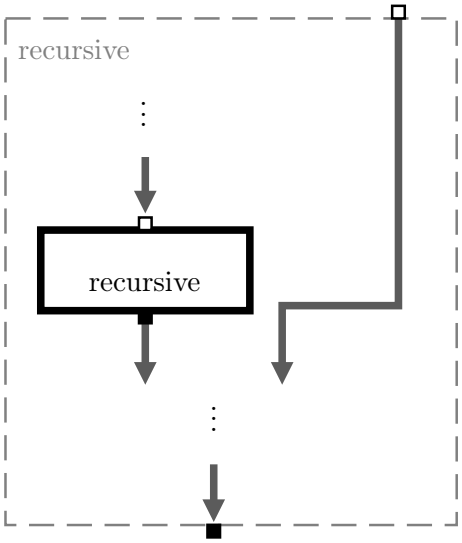


Abbildung 3.1: *Ein rekursiver Compound-Knoten mit einem Eingangsport*

Lambda-Funktionen

Insbesondere in funktionalen Programmiersprachen gehören häufig Lambda-Funktionen zum Sprachumfang. Damit können Funktionen wie Werte von Va-

riablen betrachtet und genutzt werden. In *Buggy*-Graphen werden Lambda-Funktionen mit `functional/lambda`-Knoten definiert. Diese enthalten einen *Compound*-Knoten, in dem sich die Implementierung befindet. Abbildung 3.2 zeigt einen `functional/lambda`-Knoten für eine Additionsfunktion.

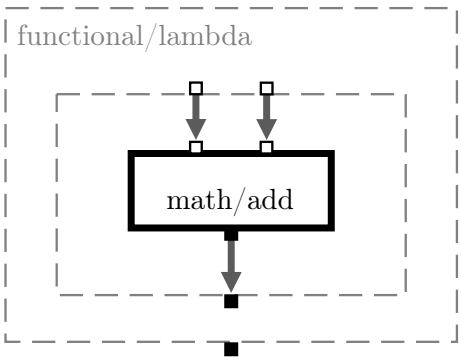


Abbildung 3.2: *Knoten einer Lambda-Funktion*

Die *Compound*-Knoten der Implementierungen von Lambda-Funktionen sind die einzigen Knoten mit Eingangsports, an denen keine Kanten enden. In allen anderen Fällen hat ein Knoten in jedem Eingangsport genau einen Vorgänger.

Typen

Weiterhin hat jeder Port einen Typ. Mögliche Typen sind derzeit `number` (Zahlen), `bool` (Boolesche Werte), `string` (Zeichenketten), `function` (Lambda-Funktionen) und Listen. Ports können auch mit dem Typ `generic` gekennzeichnet sein, in diesem Fall wird der Typ während der Typisierung des Graphen aus dem Kontext abgeleitet.

Darstellung der Graphen

In den Abbildungen der Kontrollflussgraphen werden Eingangsports weiß und Ausgangsports schwarz gezeichnet. *Compound*-Knoten werden grau gestrichelt, während atomare Knoten mit durchgezogenen schwarzen Linien gezeichnet werden.

3.3 Einbindung des Optimierers

Aus der Ausgangssprache wird ein Kontrollflussgraph erzeugt, in dem *Compound*-Knoten aus der Komponentenbibliothek zunächst nicht eingesetzt sind. Diese werden in einem zweiten Schritt eingesetzt. Dabei werden auch genaue Versionen der atomaren Operationen festgelegt. Anschließend steht ein vollständiger Kontrollflussgraph des Programms zur Verfügung. Auf diesem wird noch eine Typprüfung durchgeführt und generische Typen werden gegebenenfalls aufgelöst.

Der Optimierer arbeitet auf diesem typisierten und geprüften Kontrollflussgraphen. Somit kann vorausgesetzt werden, dass das Programm kompilierbar und der Kontrollflussgraph valide sind. Dadurch, dass der Optimierer auf dieser Ebene arbeitet, ist er unabhängig von der Zielsprache und den Ausgangssprachen des Programms und der verwendeten Komponenten.

Der vom Optimierer erzeugte Graph hat das gleiche Format wie der Ausgangsgraph. Somit können diese von den gleichen Programmen verarbeitet werden. Dazu gehören unter anderem die Codegeneratoren und Programme zum Erstellen von Abbildungen des Graphen.

Um zu gewährleisten, dass der Kontrollflussgraph bei der Optimierung korrekt bleibt, müssen bei hinzugefügten Knoten die Typen der Ports korrekt gesetzt werden. Dazu stehen in den *Buggy*-Tools einige Funktionen zur Verfügung.

Genutzt werden kann der Optimierer als eigenständiges Programm, indem als Eingabe ein Kontrollflussgraph wie oben beschrieben, angegeben wird. Alternativ kann der Optimierer auch als JavaScript-Modul von anderen Programmen genutzt werden. Auf diese Weise kann die Optimierung direkt in die *Buggy*-Konsolenanwendung integriert werden.

Kapitel 4

Implementierung des Graphersetzungssystems

Die zuvor theoretisch erklärte Graphersetzungssystem soll nun in einem Programm umgesetzt werden. Da die bestehenden Programme von *Buggy* in JavaScript implementiert sind und daher bereits eine Bibliothek mit nützlichen Graphfunktionen zur Verfügung steht, wird ebenfalls diese Programmiersprache genutzt.

Der Optimierer besteht aus einer Schnittstelle, die auch von zukünftigen weiteren Programmen genutzt werden kann und einer Kommandozeilenanwendung, die die Schnittstelle für den Nutzer verfügbar macht. Mit dieser können *Buggy*-Programme eingelesen, optimiert und anschließend ausgegeben werden. Außerdem gibt es Optionen, um weitere Informationen zu den erfolgten Optimierungen auszugeben. Die verfügbaren Kommandozeilenparameter können Tabelle 4.1 entnommen werden.

Parameter	Beschreibung
`-f <file>`	Pfad zum Programmgraphen, der optimiert werden soll. Wird dieser nicht angegeben, wird der Graph von der Standardeingabe gelesen.
`-o <file>`	Pfad der Ausgabedatei für den optimierten Programmgraphen. Wird diese Option nicht angegeben, wird der Graph in die Standardausgabe geschrieben.
`--intermediate`	Falls angegeben, wird nicht nur der optimierte Graph ausgegeben, sondern auch alle Graphen, die als Zwischenergebnisse während der Anwendung der Ersetzungsregeln entstehen.
`--stats`	Falls angegeben, werden Statistiken über die erfolgten Ersetzungen ausgegeben. Dazu gehören unter anderem die Anzahl der entfernten Knoten und Kanten und die Anzahl der Regelanwendungen.
`--verbose`	Falls angegeben, werden die angewandten Regeln und weitere Informationen über den Programmablauf während der Ersetzung ausgegeben.

Tabelle 4.1: *Kommandozeilenparameter des Optimierers*

Durch die Unterstützung für das Einlesen des *Buggy*-Programmes von der Standardeingabe und die Ausgabe auf der Standardausgabe wird die Kombination mit anderen Programmen, beispielsweise zur Generierung von Abbildungen der Graphen, vereinfacht.

4.1 Funktionsweise des Optimierers

Der Optimierer besteht prinzipiell aus einer Reihe von Ersetzungsregeln, die solange auf den Programmgraphen angewandt werden, bis sich dieser nicht mehr ändert. Um allerdings Sonderfälle in den Ersetzungsregeln zu vermeiden und um sicherzustellen, dass die Optimierung terminiert, wird diese in mehrere Phasen aufgeteilt. Durch die Aufteilung der Regeln in die verschiedenen Phasen wird eine Stratifizierung (vgl. Abschnitt 2.4) des zugrundeliegenden Graphersetzungssystems erreicht.

Optimierung I – Abstrakte semantische Strukturen

In dieser Phase werden größere Programmteile durch äquivalente kleinere oder effizientere Programmteile ersetzt. Nicht benötigte *Compound*-Knoten werden schrittweise von außen nach innen entfernt und nach jeder entfernten Stufe werden die Ersetzungsregeln aus dieser Phase erneut auf den Graphen angewandt. Dadurch ist es möglich, Ersetzungsregeln zu nutzen, deren linke Seiten *Compound*-Knoten enthalten. Beispielsweise kann eine absteigende Sortierung einer Liste mit anschließender Auswahl des ersten Elements als *Compound*-Knoten vom Typ `array/sort` gefolgt von einem `array/first`-Knoten erkannt und durch eine Maximumssuche ersetzt werden.

Entfernen von Compound-Knoten

In *Buggy*-Programmen existieren *Compound*-Knoten, um atomare Knoten zu kombinieren und als Funktionen zu bündeln, die Implementierung von Lambda-Funktionen anzugeben und um rekursive Funktionen zu definieren. Im ersten Fall kodiert der *Compound*-Knoten nur semantische Informationen, ist daher nicht erforderlich und kann daher entfernt werden. Abbildung 4.1 zeigt, wie nicht benötigte *Compound*-Knoten entfernt werden.

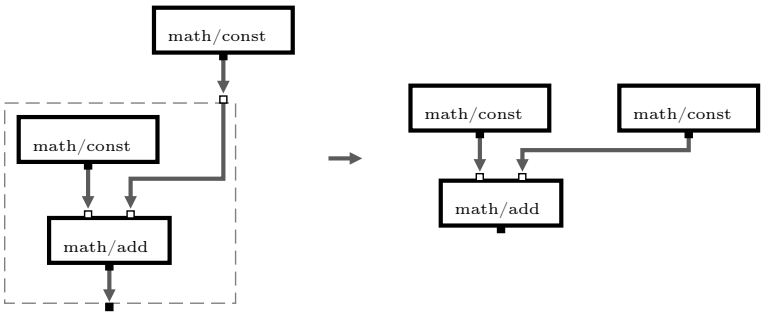


Abbildung 4.1: *Ein nicht benötigter Compound-Knoten wird aus einem Kontrollflussgraphen entfernt*

In der vorherigen Phase wird bereits ein großer Teil der nicht benötigten *Compound*-Knoten entfernt. Durch das Entfernen der *Compound*-Knoten wird der Graph in eine Form transformiert, in der weniger Sonderfälle abgefragt werden müssen. Als Zwischenschritt werden daher zunächst alle unnötigen *Compound*-Knoten entfernt. Außerdem wird nach jeder Regelanwendung in den nächsten Phasen geprüft, ob weitere *Compound*-Knoten entfernt werden können. Dies ist beispielsweise möglich, falls eine Funktion nach der Entfernung eines nicht erreichbaren Programmteils nicht mehr rekursiv ist.

Durch diesen Schritt können die Grenzen vom *Compound*-Knoten in den weiteren Ersetzungsregeln nicht gesondert werden. Diese stellen dann immer die Grenzen von rekursiven Funktionen oder Lambda-Funktionen dar, auf die die meisten Ersetzungen ohnehin nicht angewandt werden könnnen. Für rekursive Funktionen und Lambda-Funktionen werden später spezielle Regeln eingeführt, die weitere Ersetzungen, auch über diese Grenzen hinweg, erlauben.

Optimierung II

In dieser Phase werden alle weiteren Ersetzungen durchgeführt. Dazu gehören unter anderem die Ersetzung konstanter Berechnungen und das Entfernen von nicht benötigten Programmteilen. Die verwendeten Ersetzungsregeln werden in Kapitel 5 erläutert.

Bereinigung

Für alle Ersetzungsregeln wird gefordert, dass der resultierende Programmgraph korrekt und äquivalent zum ursprünglichen Programmgraphen ist. Dadurch, dass einige Ersetzungen optimistisch mit der Hoffnung, weitere Ersetzungen zu ermöglichen, durchgeführt werden, ergeben sich unter Umständen verbesserungswürdige Programmgraphen.

Zum Beispiel werden in der zweiten Optimierungsphase konstante Parameter rekursiver Funktionen in die Funktionen verschoben, um innerhalb der Funktion weitere Ersetzungen zu ermöglichen. Falls innerhalb der Funktion aber keine weiteren Ersetzungen möglich sind, wird der konstante Parameter dadurch bei jedem rekursiven Aufruf neu berechnet. In dieser Phase würde der konstante Parameter wieder aus der rekursiven Funktion extrahiert werden, sodass er nur einmal berechnet werden muss.

4.2 Definition von Ersetzungsregeln

Um die Ersetzungsregeln so flexibel wie möglich zu gestalten, sind diese als Funktionen implementiert, die einen Graphen übergeben bekommen, diesen verändern und zurückgeben. Jede dieser Funktion besteht aus zwei Teilen: einer Matcher-Funktion, mit der ein Knoten gesucht wird, auf den die Regel angewandt werden kann, und eine Ersetzungsfunktion, die einen gefundenen Knoten ersetzt.

Um die Ersetzungsfunktionen zu erzeugen, werden Funktionen höherer Ordnung verwendet. Mit der Funktion `rule(match, replace)` kann eine Regel erzeugt werden, die mit der angegebenen Matcher-Funktion einen Knoten sucht, für den die Regel anwendbar ist. Falls ein solcher Knoten gefunden wird, wird die Funktion `replace` aufgerufen und die Ersetzungsregel wird angewandt. Alle Ersetzungsregeln sind auf diese Weise implementiert.

Die Matcher-Funktionen werden ebenfalls über Funktionen höherer Ordnung definiert, sodass die aufwendige Logik aus den Regeln entfernt wird. Der am häufigsten verwendete Matcher ist `match.byIdAndInputs(id, inputs)`, der einen Knoten anhand des Typs und der Vorgänger findet. Für die Vorgänger werden weitere Matcher angegeben. Ferner sind Matcher für beliebige Knoten (`match.any()`) und für Lambda-Funktionen implementiert (`match.lambdaFunction()`). Letzterer kann noch eine Reihe von Optionen übergeben bekommen, um rekursive Lambda-Funktionen oder solche mit Seiteneffekten auszuschließen. Konstante Knoten können mit `match.constant(value)` gefunden werden, wobei der Parameter den Wert der Konstanten angibt.

Für häufig auftretende Ersetzungsfunktionen werden ebenfalls Funktionen höherer Ordnung implementiert, die die Ersetzungsfunktionen erzeugen. Mit `replace.withNode(node)` kann der gefundene Knoten durch einen anderen Knoten ersetzt werden, anschließend ungenutzte Vorgänger werden entfernt. Die Funktion `replace.bridgeOver(edgeCreator)` erlaubt es, einen Knoten zu überbrücken, indem neue Kanten hinzugefügt werden und löscht anschließend den Knoten. Der Parameter `edgeCreator` ist dabei eine Funktion, die die benötigten Kanten abhängig vom gefundenen Knoten erzeugt.

Die Regeln zur Ersetzung von Alternativen mit konstanter Bedingung lässt sich zum Beispiel wie folgt mit diesen Hilfsfunktionen definieren.

```
rule(
  match.byIdAndInputs('logic/if', {
    control: match.constant(true),
    input1: match.any(),
    input2: match.any()
  }),
  (graph, node, match) => replace.bridgeOver((graph, node, match) => [{
    source: match.inputs.input1.node,
    target: match.node
  }]
)
```

Dabei sucht `match.byIdAndInputs` Knoten, die vom angegebenen Typ sind (hier `logic/if`) und deren Vorgänger der einzelnen Ports (`control`, `input1` und `input2`) von den jeweils angegebenen Matchern akzeptiert werden. In diesem Beispiel muss der Vorgänger des `control`-Ports eine Konstante mit dem Wert `true` sein. Die Vorgänger der beiden anderen Ports sind beliebig. Mit `replace.bridgeOver` wird der gefundene `logic/if`-Knoten dann überbrückt, indem Kanten vom Vorgänger des `input1`-Ports zu allen Nachfolgern des `logic/if`-Knotens eingefügt werden. Der überbrückte Knoten wird anschließend entfernt.

Auf diese Weise können die Ersetzungsregeln auf einer abstrakteren Ebene definiert werden, gleichzeitig ergibt sich eine große Flexibilität für einige Regeln, indem beliebige Funktionen zur Ersetzung verwendet werden können.

4.3 Einfügen von Knoten

Bei der Ersetzung müssen häufig größere Teilgraphen erzeugt und eingefügt werden. Um dies zu vereinfachen, wird eine Funktion `createSubgraph(graph, node, subgraph)` implementiert. Mit dieser können zusammenhängende Graphen erzeugt und in einen bestehenden Graphen eingefügt werden. Dabei ist `subgraph` ein Objekt, das einen Knoten und dessen Vorgänger und Nachfolger für jeden Port angibt. Die Vorgänger und Nachfolger werden ebenfalls über ein solches Objekt deklariert. Es können aber auch die Namen bestehender Knoten angegeben werden, sodass der neue Teilgraph mit dem Graph verbunden wird.

Die Abbildung 4.2 demonstriert die Verwendung dieser Funktion und zeigt den resultierenden Graphen. Die Funktionen unter `nodes.*` erzeugen dabei Werte für Knoten des entsprechenden Typs. Dabei sind `isLess` und `than` die Eingangsports und `output` der Ausgangsport des `math/less`-Knotens. `s1` und `s2` sind die Eingangsports des `math/add`-Knotens.

```
createSubgraph(graph, node, {
  node: nodes.math.less(),
  predecessors: {
    isLess: {
      node: nodes.math.add(),
      predecessors: {
        s1: nodes.constant(1),
        s2: nodes.constant(3)
    },
    than: nodes.constant(10)
  },
  successors: {
    output: nodes.logic.not()
  }
})
```

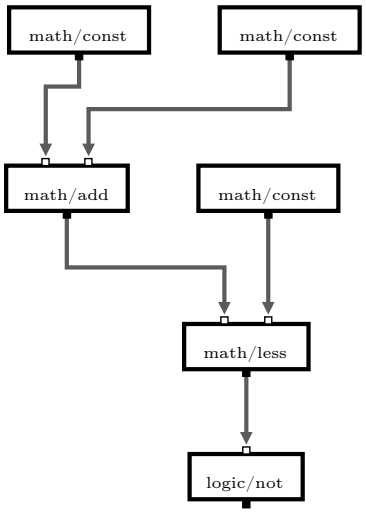


Abbildung 4.2: *Beispiel für die Erzeugung eines Graphen mit* `createSubgraph`

Kapitel 5

Ersetzungsregeln

5.1 Anforderungen an die Ersetzungsregeln

Mithilfe der in Abschnitt 3.2 aufgeführten Definitionen und Anforderungen an den Kontrollflussgraphen können folgende Regeln für die Ersetzungsregeln festgelegt werden:

- Beim Entfernen eines Knotens müssen rekursiv auch alle enthaltenen Knoten entfernt werden.
- Ein Knoten v wird nicht mehr benötigt, falls nach dem Entfernen eines Nachfolgers keine Kante $e = ((v, p), (u, r)) \in E,\ p \in P_O$ mehr existiert. In diesem Fall kann der Knoten entfernt werden, sodass der Pfad so weit wie möglich entfernt wird.
  Falls Seiteneffekte (beispielsweise das Schreiben von Dateien) erhalten werden sollen, dürfen Knoten mit Seiteneffekten nicht entfernt werden.
- Falls ein Vorgänger u eines *Compound*-Knotens v am Eingangsport r entfernt aber nicht ersetzt wird und somit auch die Kante $((u, p), (v, r)) \in E$ entfernt wird, muss der Eingangsport r von v entfernt werden.
- Die Vorgänger von atomaren Knoten dürfen nur entfernt werden, falls auch der atomare Knoten entfernt wird. Ports von atomaren Knoten dürfen nicht entfernt werden, daher hätten diese andernfalls Eingangsports, zu denen keine Kante führt.

Von der Implementierung des Optimierers wird gefordert, dass das *Buggy*-Programm nach Anwendung jeder einzelnen Ersetzungsregel korrekt ist. Dadurch sind die Umformungen leichter nachvollziehbar.

5.2 Konstante Berechnungen

Zunächst werden eine Reihe von Ersetzungsregeln implementiert, um konstante Operationen aufzulösen. Dabei wird pro Regelanwendung nur eine Operation durch eine Konstante ersetzt, verschachtelte Berechnungen können dann durch mehrfache Anwendung ersetzt werden. Auf diese Weise können Programmteile, die unabhängig von Eingaben zur Laufzeit sind, bereits vorberechnet werden. Besonders bei rekursiven Programmen und Schleifen sind dadurch Performanzverbesserungen zu erwarten.

Im folgenden heißt ein Knoten im Kontrollflussgraphen *konstant*, falls die Werte aller Ausgangsports bekannt sind. Der wichtigste konstante Knoten in *Buggy* ist `std/const`, ein Knoten, der einen Ausgangsport hat, der einen festen Wert liefert. Der Ausgangsport ist generisch, sodass der Knoten für konstante Werte von jedem möglichen Typ genutzt werden kann. Der Knoten `math/const` hat die gleiche Funktion, unterstützt allerdings nur numerische Werte.

5.2.1 Mathematische Operationen

Die einfachsten mathematischen Operationen in *Buggy* sind Addition (`math/add`), Subtraktion (`math/subtract`), Multiplikation (`math/multiply`) und Division (`math/divide`). Sofern an beiden Eingängen der Knoten Konstanten stehen, kann er Knoten selbst durch einen Knoten mit einer entsprechend berechneten Konstante ersetzt werden. Abbildung 5.1 demonstriert die Funktionsweise der Ersetzungsregel.

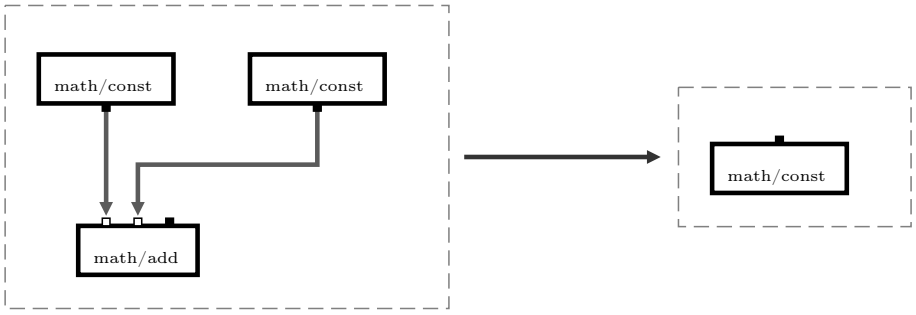


Abbildung 5.1: *Ersetzung einer Addition von zwei Konstanten, der Wert des erzeugten Knotens entspricht der Summe der Eingangswerte links*

Ist einer der beiden Operanden das neutrale Element bezüglich der jeweiligen Operation, können auch dann Ersetzungen durchgeführt werden, wenn der andere Operator nicht konstant ist. Bei der Addition mit 0 und der Multiplikation mit 1 kann die Operation entfernt werden und der Eingangsknoten, der nicht gleich 0 bzw. 1 ist, direkt mit den Nachfolgeknoten verbunden werden. Bei der Subtraktion mit 0 und der Division durch 1 funktioniert dies auch, allerdings nur, wenn 0 der Subtrahend bzw. 1 der Divisor ist, da beide Operationen nicht kommutativ sind.

Bei der Multiplikation kann eine weitere Ersetzung durchgeführt werden, falls einer der beiden Operanden 0, also das absorbierende Element bezüglich der Multiplikation, ist. In diesem Fall kann die Multiplikation durch eine Konstante 0 ersetzt werden und beide Eingangspfade können entfernt werden.

Eine weitere Ersetzungsregel für die Multiplikation ergibt sich aus der Assoziativität. Im *Buggy*-Graphen für die Berechnung von x^4 wird beispielsweise das Produkt $x \cdot x \cdot x \cdot x$ berechnet. Dabei werden drei Multiplikationen durchgeführt. Wegen $x \cdot x \cdot x \cdot x = (x \cdot x) \cdot (x \cdot x)$ ist es ausreichend, das Produkt $x \cdot x$ zu berechnen und das Ergebnis anschließend an beiden Eingängen des Multiplikationsknotens zu verwenden. Dabei wird eine Multiplikation eingespart. Durch wiederholte Anwendung der Regel können auch höhere Potenzen optimiert werden.

Die Kommutativität der Multiplikation und der Addition wird durch eine weitere Regel ausgenutzt. Folgen zwei Additionen nacheinander, und sind zwei der drei Summanden konstant, werden die Summanden so getauscht, sodass die zwei Konstanten addiert werden (vgl. Abbildung 5.2). Anschließend können die zuvor erläuterten Regeln angewandt werden, um eine der Additionen auszurechnen. Analog funktioniert diese Regel auch für die Multiplikation und jede weitere zweistellige kommutative Operation (beispielsweise für die logischen Operationen $\vee$ und $\wedge$).

Weiterhin gibt es in *Buggy* die Vergleichsoperationen $=$, $\neq$, $<$, $\leq$, $>$ und $\geq$ für Zahlen. Die entsprechenden Knoten haben zwei Eingänge und einen Ausgang, der das Ergebnis des Vergleichs als booleschen Wert angibt. Sind beide Eingänge konstant, kann das Ergebnis des Vergleichs berechnet und der Vergleich inklusive der Vorgängerknoten durch eine boolesche Konstante ersetzt werden.

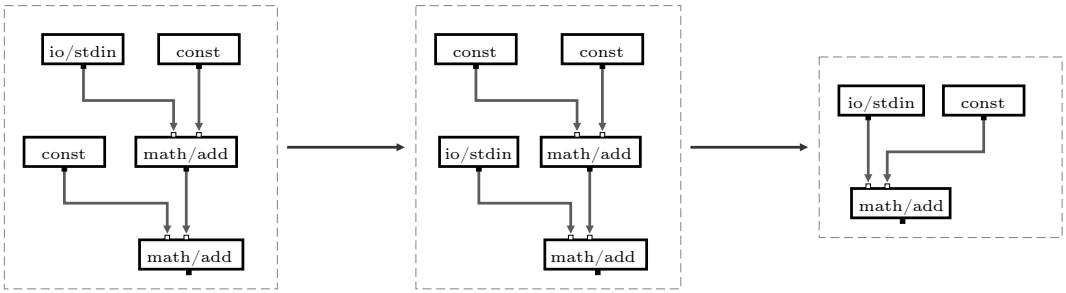


Abbildung 5.2: *Tausch zweier Summanden und anschließende Ersetzung einer Addition. Der Knoten* `io/stdin` *ist eine Benutzereingabe und der Wert daher erst zur Laufzeit bekannt.*

Zusätzlich kann überprüft werden, ob beide Eingänge den gleichen Vorgängerknoten haben und die Werte folglich gleich sind. Der Vergleich kann dann auch ersetzt werden, falls die Eingangsknoten nicht konstant sind.

5.2.2 Logische Verknüpfungen

Neben den Grundrechenarten bietet *Buggy* auch die logischen Verknüpfungen Konjunktion (`logic/and`), Disjunktion (`logic/or`) und Negation (`logic/not`). Hinzu kommen die booleschen Werte `true` und `false`, die mit `std/const`-Knoten angegeben werden können. Weitere Operationen können durch Kombination als *Compound*-Knoten definiert werden. Da die Ersetzungsregeln auch über Grenzen von *Compound*-Knoten hinweg funktionieren, können auch die zusammengesetzten Verknüpfungen mithilfe der folgenden Regeln vorberechnet werden, falls die Operanden konstant sind.

Die Knoten `logic/or` und `logic/and` werden durch eine entsprechend berechnete boolesche Konstante ersetzt, falls beide Vorgängerknoten konstant sind. Die Negation kann ebenfalls ersetzt werden, falls der Vorgängerknoten konstant ist. Die Vorgängerknoten können bei diesen Ersetzungsregeln entfernt werden, sofern sie keine weiteren Nachfolgeknoten haben.

Da die Konjunktion und die Disjunktion kommutativ sind, können die Operanden, wie bei der Addition und Multiplikation gezeigt, getauscht werden, sodass weitere Ersetzungsregeln angewandt werden können.

Um unnötige Vorkommen der Negation zu entfernen, werden Vorkommen der Vergleichsoperationen $=$, $\neq$, $<$, $\leq$, $>$ und $\geq$, falls auf diese nur eine Negation folgt, durch die jeweilige inverse Operation ersetzt. Die logische Negation `logic/not` wird, falls auf diese eine Negation folgt, durch die Identitätsfunktion (`std/id`) ersetzt, die von einer weiteren Ersetzungsregel entfernt wird. Effektiv werden somit zwei aufeinanderfolgende Negationen entfernt und der Vorgängerknoten der ersten Negation direkt mit den Nachfolgeknoten der zweiten Negation verbunden.

Außerdem werden Konjunktionen mit `true` und Disjunktion mit `false` entfernt. Konjunktionen mit `false` und Disjunktionen mit `true` werden durch Konstanten ersetzt, falls der Pfad der Eingangsknoten keine Seiteneffekte hat. Die De Morganschen Gesetze können ebenfalls genutzt werden, um Knoten zu entfernen, effektiv werden dabei die Ausdrücke $\neg a \vee \neg b$ durch $\neg(a \wedge b)$ und $\neg a \wedge \neg b$ durch $\neg(a \vee b)$ ersetzt.

5.2.3 Pure Funktionen

Neben den arithmetischen und logischen Funktionen sind auch viele weitere atomare Funktionen in *Buggy* pure Funktionen. Die Werte ihrer Ausgangsports hängen lediglich von den Werten der Eingangsports ab und die Funktionen haben keine Nebeneffekte.

Falls die Vorgängerknoten von Knoten solcher Funktionen konstant sind, kann die Funktion während der Optimierung berechnet und das Ergebnis als Konstante anstelle der Funktion eingesetzt werden. Für einige Funktionen werden daher weitere Ersetzungsregeln implementiert. Somit können unter anderem Typkonvertierungen von Konstanten, beispielsweise von einer Zahl zu einer Zeichenkette, während der Optimierung durchgeführt werden.

Da letztlich jedes *Buggy*-Programm nur aus atomaren Knoten besteht, können mit diesen Regeln auch größere Programmteile vom Optimierer vorberechnet werden. Begrenzt wird dies jedoch dadurch, dass keine rekursiven Funktionen ausgerechnet werden und nicht für alle ersetzbaren atomaren Knoten Ersetzungsregeln verfügbar sind. Langfristig soll es in *Buggy* jedoch nur eine feste Anzahl an atomaren Knoten geben, sodass die Ersetzungsregeln für diese dann nur selten erweitert werden müssen.

5.3 Unerreichbare Programmteile

5.3.1 Bedingte Anweisungen

Bedingte Anweisungen sind in *Buggy* mit dem speziellen Knoten `logic/mux` realisierbar. Dieser kann als Auswahl von Werten anhand einer Bedingung mit *lazy evaluation* aufgefasst werden. Normalerweise werden in *Buggy* alle Programmzweige ausgewertet, was bei bedingten Anweisungen jedoch nicht gewünscht ist. Durch `logic/mux` wird, je nach Ergebnis der Auswertung der Bedingung, nur einer der beiden möglichen Vorgänger-Zweige ausgewertet werden und das Ergebnis auf den Ausgangsport gelegt.

Ist die Bedingung konstant, können die `logic/mux`, der nicht ausgewählte Teilgraph und die Bedingung entfernt werden und der gewählte Teilgraph direkt mit den Nachfolgern des `logic/mux`-Knotens verbunden werden.

Zu beachten ist, dass der Eingang für die Bedingung der dritte Eingangsport des `logic/mux`-Knotens ist. Da diese Reihenfolge eher ungewöhnlich ist, existiert ein *Compound*-Knoten `logic/if`. Bei diesem ist der Eingang für die Bedingung der erste Eingangsport. Intern nutzt dieser Knoten `logic/mux`, sodass die Ersetzungsregeln diesen Knoten nicht gesondert berücksichtigen müssen.

5.3.2 Ungenutzte Lambda-Funktionen

Nach Anwendung einiger Graphersetzungsregeln verbleiben Lambda-Funktionen im Kontrollflussgraphen, die keine Nachfolger haben. Diese werden nicht ausgewertet oder anderweitig genutzt und können daher entfernt werden.

5.3.3 Ungenutzte Programmteile

Durch das Entfernen ungenutzter Programmteile ergibt sich zwar nicht unmittelbar ein Geschwindigkeitsgewinn, allerdings kann die Größe des Kontrollflussgraphen und somit auch die Größe des resultierenden Programms reduziert werden.

Es gibt in *Buggy* einige atomare Knoten, die im folgenden als *Senke* bezeichnet werden. Dazu gehört unter anderem `io/stdout` zur Ausgabe von Text. Ein Pfad von Knoten wird in einem *Buggy*-Graphen benötigt, falls der letzte Knoten eine Senke ist. Einige Knoten können bei ihrer Auswertung jedoch Seiteneffekte hervorrufen, sodass sie nicht entfernt werden dürfen, falls sie im ursprünglichen Graphen ausgewertet werden würden.

In *Buggy* werden atomare und rekursive Knoten ausgewertet, sobald die Werte aller Eingangsports bekannt sind. Daher dürfen Knoten mit Seiteneffekten generell nur entfernt werden, falls nicht alle Eingangsports belegt sind. Die beiden zuvor beschriebenen Regeln sind davon nicht betroffen. Die Implementierungen von Lambda-Knoten werden erst beim Aufruf der Lambda-Funktion ausgewertet, sodass Seiteneffekte in ungenutzten Lambda-Funktionen nicht auftreten. Im Fall von `logic/mux` wird, abhängig von der Auswertung der Bedingung, nur einer der beiden möglichen Zweige ausgewertet, sodass mögliche Seiteneffekte aus dem nicht ausgewählten Zweig nicht auftreten würden.

Zur Entfernung der ungenutzten Knoten werden zunächst alle Knoten markiert, die gemäß den obenstehenden Bedingungen nicht entfernt werden dürfen. Als nächstes werden alle Vorgängerknoten markiert. Weiterhin werden enthaltenen Knoten, sofern sie mindestens einen markierten Nachfolgeknoten haben, ebenfalls markiert. Dies wird für alle neu markierten Knoten rekursiv wiederholt, bis keine neuen Knoten mehr markiert werden. Anschließend sind alle benötigten Knoten markiert und es werden alle nicht markierten Knoten und alle Kanten, die mit mindestens einem nicht markierten Knoten verbunden sind, entfernt.

Knoten von Lambda-Funktionen müssen dabei besonders berücksichtigt werden. Der enthaltene *Compound*-Knoten für die Implementierung hat zwar keine Nachfolgeknoten, muss jedoch trotzdem markiert werden, falls die Lambda-Funktion markiert wird. Der Grund dafür ist, dass die Implementierung eines Lambda-Knotens ebenfalls benötigt wird, falls der Lambda-Knoten benötigt wird.

5.4 Optimierung funktionaler Konzepte

Zur Zeit ist Buggy in erster Linie eine Plattform für funktionale Programmierung. Bei der Nutzung von *Lisgy* wird dies besonders deutlich. In diesem Abschnitt werden einige Ersetzungsregeln eingeführt, um die grundlegenden funktionalen Operationen in *Buggy* zu optimieren. Ziel ist es, dass eine funktionale Programmierweise, etwa die Nutzung von partiellen Funktionen, die Geschwindigkeit des generierten Programmes nicht zu stark reduziert.

5.4.1 Lambda-Funktionen

In den Kontrollflussgraphen werden Lambda-Funktionen mit `functional/lambda`-Knoten definiert. Diese enthalten einen *Compound*-Knoten mit der Implementierung. In vielen Fällen sind die Lambda-Funktionen daher während der Optimierung bekannt. Werden die Lambda-Funktionen dynamisch verändert, beispielsweise durch das Binden von Parametern, ist dies nicht der Fall und es gibt weniger Optimierungsmöglichkeiten.

Das Binden von Argumenten an eine Funktion ist mit dem Knoten `functional/partial` möglich. Diese erhalten als Eingang eine Lambda-Funktion und einen Wert für ein Argument und geben eine neue Funktion zurück, bei der das Argument entsprechend gesetzt ist.

Funktionen können mit `functional/call`-Knoten ausgewertet werden. Die Knoten erhalten eine Lambda-Funktion und geben den Rückgabewert der ausgewerteten Funktion zurück.

5.4.2 Inlining von Lambda-Funktionen

Wird eine statisch bekannte Lambda-Funktion ausgewertet, kann der `functional/call`-Knoten durch die Implementierung der Lambda-Funktion ersetzt werden. Dadurch wird der Zusatzaufwand des Funktionsaufrufes vermieden und es können weitere Ersetzungen ermöglicht werden. Ein Beispiel dazu findet sich in Abbildung 5.3. Im ersten Schritt wird dabei der Aufruf der Lambda-Funktion durch deren Implementierung ersetzt und im zweiten Schritt wird die nicht mehr genutzte Lambda-Funktion entfernt.

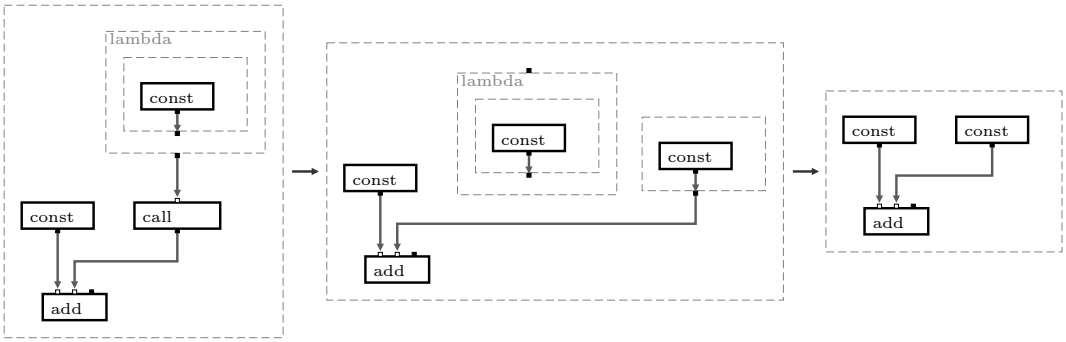


Abbildung 5.3: *Inlining einer Lambda-Funktion in zwei Schritten*

Häufiger treten jedoch Auswertungen von Lambda-Funktionen auf, deren Parameter unmittelbar vor der Auswertung an die Funktion gebunden werden (vgl. Abbildung 5.4). In diesen Fällen ist es ebenfalls möglich, die Implementierung der Lambda-Funktion direkt einzufügen.

Eine Besonderheit hierbei ist, dass der zu ersetzende Teilgraph beliebig groß sein kann, da vor dem `functional/call`-Knoten beliebig viele Parameter an die Funktion gebunden werden können. Allerdings haben die `functional/partial`-Knoten in dieser Kette immer die gleiche Struktur. Jeder `functional/partial`-Knoten hat als Vorgänger an einem Eingangsport eine Funktion oder einen weiteren `functional/partial`-Knoten und am anderen Eingangsport einen beliebigen anderen Knoten, die Knotenkette ist also gewissermaßen rekursiv aufgebaut.

Die Parameter p_i in Abbildung 5.4 stehen nur stellvertretend für die Werte, die an die Funktion gebunden werden. Zur Vereinfachung ist die Lambda-Funktion nur durch einen einfachen Knoten angedeutet. Praktisch können die Teilgraphen für die gebundenen Werte beliebig aufwendig sein. Für diese Regel ist nur wichtig, dass sie mit den `functional/partial`-Knoten verbunden sind.

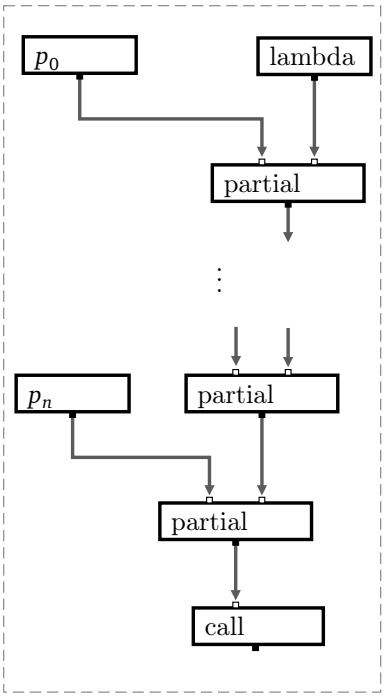


Abbildung 5.4: *Kontrollflussgraph für eine Lambda-Funktion, an die mehrere Parameter gebunden werden*

Zum Erkennen der passenden Teilgraphen wird ausgenutzt, dass zum Matching der Teilgraphen Funktionen verwendet werden. Dadurch ist auch die Definition von rekursiven Matchern für derartige Teilgraphen möglich.

```
let matchPartialOrLambda = match.oneOf([
  match.lambda(),
  match.byIdAndInput('functional/partial', {
    fn: matchPartialOrLambda,
    value: match.any()
  })
})

let matchCallWithParameters = match.byIdAndInput('functional/call', {
  fn: matchPartialOrLambda
})
```

Wichtig ist dabei, dass `match.oneOf` die Matcher-Funktionen in der angegebenen Reihenfolge auf den jeweiligen Knoten anwendet und abbricht, falls der Knoten zur Matcher-Funktion passt. Somit ist das Erreichen einer Lambda-Funktion die Abbruchbedingung des rekursiven Matchers. Andernfalls erfolgt ein rekursiver Aufruf des Matchers für den vorangegangenen `functional/partial`-Knoten.

Zur Ersetzung wird der `functional/call`-Knoten durch die Implementierung der Lambda-Funktion ersetzt. Die Werte, die mit `functional/partial`-Knoten an die Funktion gebunden

werden, werden anschließend in der richtigen Reihenfolge durch neue Kanten im Graphen mit den jeweiligen Ports in der Implementierung der Lambda-Funktion verbunden werden.

Im Beispiel in Abbildung 5.5 ist die Addition (`math/add`) in eine Lambda-Funktion eingebettet, an die anschließend als erster Parameter die Konstante 1 gebunden wird. Daduch entsteht eine Inkrement-Funktion, an die anschließend eine zweite Konstante gebunden wird, bevor die resultierende Funktion ausgewertet wird. Durch die zuvor definierten Ersetzungsregel wird der Kontrollflussgraph so umgeschrieben, dass durch die Verwendung der Lambda-Funktion kein Mehraufwand gegenüber einer direkten Addition mit 1 entsteht. Anschließend können dann die zuvor vorgestellten Ersetzungsregeln angewandt werden, was mit der Lambda-Funktion nicht möglich war.

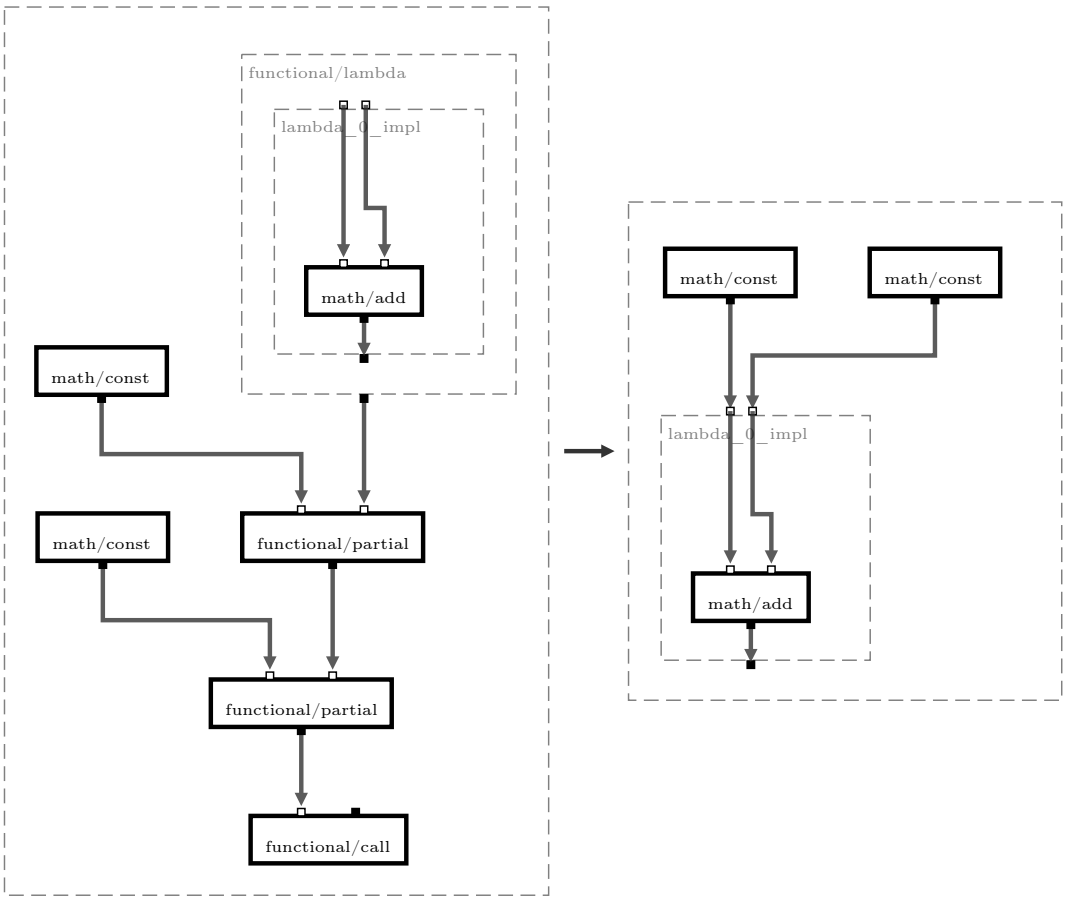


Abbildung 5.5: *Inlining einer Lambda-Funktion mit zwei gebundenen Parametern*

5.4.3 Partielle Funktionsanwendung

Nicht immer wird eine Lambda-Funktion direkt ausgewertet, nachdem Parameter gebunden werden. Stattdessen können die partiell angewandten Funktionen auch erst an anderer Stelle genutzt und ausgewertet werden. Die Implementierung von partiell angwandten Funktionen ist statisch in den meisten Fällen nicht bekannt. Im folgenden werden Ersetzungsregeln eingeführt, die in einigen Fällen partiell angewandte Funktionen berechnen und somit die `functional/partial`-Knoten durch Lambda-Funktionen ersetzen können.

Falls ein Wert an eine Lambda-Funktion gebunden wird, dessen berechnender Teilgraph nicht von anderen Knoten abhängt und keine Seiteneffekte hat, kann dieser in die Lambda-Funktion verschoben werden. Der entsprechende Eingangsport der Lambda-Funktion wird anschließend entfernt. Abbildung 5.6 zeigt eine einfache Anwendung dieser Regel. Praktisch wird durch diese Ersetzungsregel eine partiell angewandte Lambda-Funktion berechnet. Hätte der Teilgraph für die Berechnung des gebundenen Wertes Seiteneffekte, würden diese anschließend bei jedem Aufruf der resultierenden Lambda-Funktion auftreten, weshalb diese Regel in diesem Fall nicht anwendbar wäre.

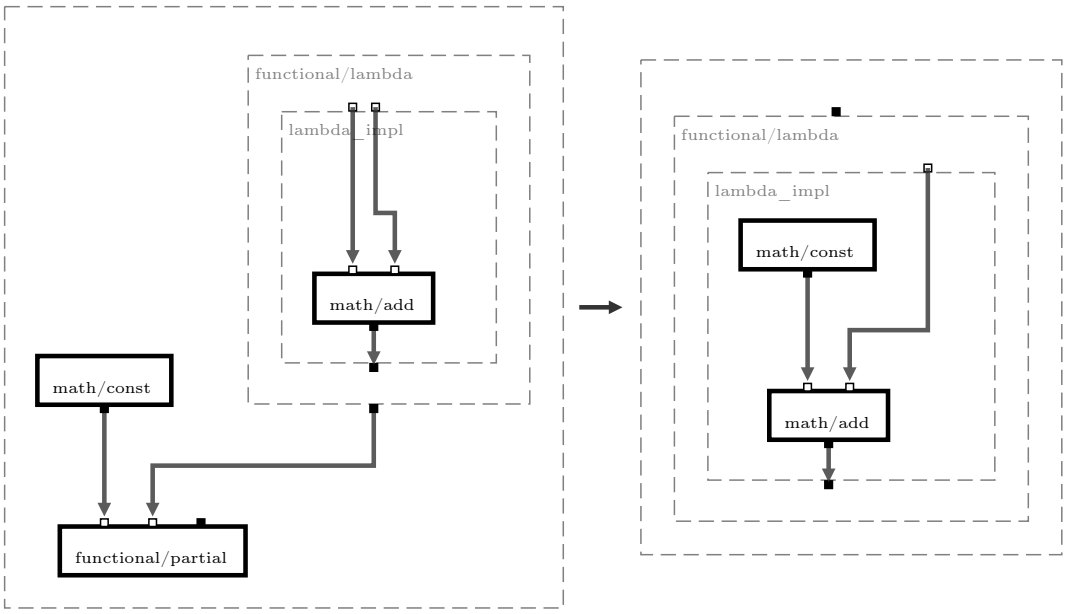


Abbildung 5.6: *Einsetzen eines Teilgraphen in eine Lambda-Funktion zur Berechnung einer partiell angewandten Funktion*

Ein Sonderfall liegt vor, falls ein `functional/partial`-Knoten für eine Lambda-Funktion λ_1 mit einem Argument an eine Lambda-Funktion λ_2 gebunden wird. In diesem Fall können die beiden Lambda-Funktionen zu einer neuen Funktion λ_2' kombiniert werden. Der ursprünglich an λ_1 gebundene Wert wird dazu an λ_2' gebunden. Da der Teilgraph, der den gebundenen Wert berechnet, dazu nicht verschoben werden muss, gelten die Beschränkungen aus der vorherigen Regel für λ_1 nicht. Allerdings ist diese Regel nur anwendbar, falls λ_1 genau ein Argument hat, da λ_2' andernfalls mehr Argumente als λ_2 hätte, sodass sie von den nachfolgenden Knoten nicht wie λ_2 genutzt werden könnte.

Dieser Fall tritt unter anderem in der Implementierung der *fold*-Operation auf, die wiederum unter anderem in der Quicksort-Implementierung genutzt wird.

5.5 Optimierung von rekursiven Funktionen

5.5.1 Konstante Berechnungen in rekursiven Funktionen

Die *Compound*-Knoten von rekursiven Funktionen können in der ersten Phase der Optimierung nicht entfernt werden, da sie für die rekursiven Aufrufe benötigt werden. Die Werte der Eingangsports entsprechen bei rekursiven Aufrufen außerdem nicht mehr den Vorgängern des *Compound*-Knoten im Kontrollflussgraphen. Daher sind Optimierungen über die Grenzen von rekursiven *Compound*-Knoten zunächst nicht möglich.

Anhand des Kontrollflussgraphen lässt sich jedoch einfach feststellen, ob ein Eingangsport eines rekursiven *Compound*-Knotens immer den gleichen Vorgänger und somit stets den gleichen Wert hat. Falls der Vorgängerpfad keine weiteren Abzweigungen hat, kann dieser in die rekursive Funktion verschoben werden, sodass innerhalb des *Compound*-Knotens weitere Ersetzungen ermöglicht werden.

Konstante Berechnungen aus rekursiven Funktionen entfernen

Falls dadurch innerhalb der rekursiven Funktion keine weiteren Optimierungen ermöglicht werden, ergibt sich durch diese Regel ein Mehraufwand, da Werte, die zuvor einmalig vor dem initialen Aufruf der rekursiven Funktion berechnet wurden, jetzt bei jedem rekursiven Aufruf neu berechnet werden müssen. Die folgende Regel extrahiert konstante Berechnungen aus rekursiven Funktionen. Da sie die zuvor beschriebene Regel umkehrt, wird sie erst in der Bereinigungsphase des Optimierers angewandt. Andernfalls würde das Graphersetzungssystem nicht terminieren.

Dies wird über zwei Ersetzungsregeln erreicht. Mit der ersten Ersetzungsregel werden Knoten in rekursiven *Compound*-Knoten, die keine Vorgänger haben und somit unabhängig von den eingängen des *Compound*-Knotens sind, aus diesem entfernt. Der *Compound*-Knoten erhält neue Eingangsports und die verschobenen Knoten werden über diese mit deren ursprünglichen Nachfolgern verbunden.

Die zweite Regel sucht Knoten in rekursiven *Compound*-Knoten, die nur von den Eingangsports des rekursiven Knotens abhängen. Außerdem müssen diese Eingänge bei jedem rekursiven Aufruf wieder an die gleichen Ports weitergeleitet werden. In diesem Fall ist auch der Wert des Knotens konstant und er kann aus dem rekursiven *Compound*-Knoten entfernt werden. Der neue Knoten wird über einen neuen Eingangsport des rekursiven *Compound*-Knotens wieder mit dessen ursprünglichen Nachfolgern verbunden.

Abbildung 5.7 zeigt die Funktionswiese dieser beiden Ersetzungsregeln. In diesem Beispiel werden die Multiplikation und die beiden Faktoren bei jedem rekursiven Aufruf berechnet. Nach Anwendung der Regeln werden diese nur noch einmalig berechnet und die Werte werden den rekursiven Aufrufen übergeben. Die verwendeten Knoten stehen hierbei nur beispielhaft für beliebige Knoten, die die zuvor benannten Voraussetzungen erfüllen.

5.5.2 Ungenutzte Eingangsports

Im Beispiel in Abbildung 5.7 behält der rekursive *Compound*-Knoten einige Eingangsports, die bei jedem Aufruf der Funktion den gleichen Wert erhalten. Die Vorgänger der Eingangsports können dann in die Funktion verschoben werden. Effektiv werden die Eingangsports dann nicht genutzt, sodass sie von einer weiteren Ersetzungsregel entfernt werden können. Grundsätzlich

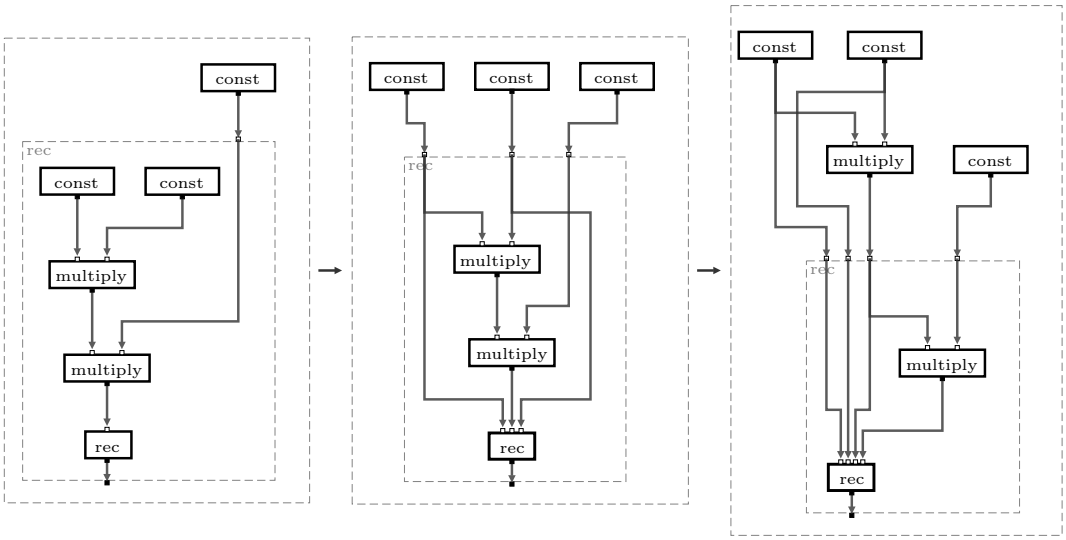


Abbildung 5.7: *Extraktion einer von den Funktionsargumenten unabhängigen Berechnung aus einer rekursiven Funktion in zwei Schritten*

können alle Eingangsports von *Compound*-Knoten entfernt werden, die keine Nachfolger haben, sofern die Vorgängerknoten des Eingangsports keine Seiteneffekte haben.

5.5.3 Endrekursive Funktionen

Bei der funktionalen Programmierung treten häufig rekursive Funktionen auf, sodass Optimierungen für diese benötigt werden.

Im folgenden wird die Berechnung der Fakultät als Beispiel genutzt. Diese berechnet sich rekursiv als $n! = n \cdot (n-1)!$ mit $0! = 1$. In der Präfix-Schreibweise von *Lisgy* lässt sich diese Formel wie folgt implementieren:

```
(import all)

(defco fac [n]
  (logic/mux
    (* n (fac (- n 1)))  ; n * (n - 1)!, falls 1 < n
    1                    ; 1 sonst
    (< 1 n)
  )
)
```

In imperativen Programmiersprachen würde man die Fakultät nicht rekursiv berechnen. Da die Parameter und die lokalen Variablen für Funktionsaufrufe auf dem Programmstack abgelegt werden, kommt es bei großen n zu Stack-Überläufen und somit zum Absturz des Programmes. Außerdem ergibt sich durch die Funktionsaufrufe ein in diesem Fall vermeidbarer Zusatzaufwand.

Die Funktion `fac` würde daher wie folgt mithilfe einer Schleife implementiert werden.

```
function fac(n) {
  let fac = 1
  while (n > 1) {
    fac = n * fac
    n = n - 1
  }
  return fac
}
```

Es ist allerdings im Allgemeinen nicht programmatisch möglich, aus einem rekursiven Algorithmus einen iterativen Algorithmus zu generieren, der nicht den Programmstack simuliert. Daher wird dieser Aufwand zunächst größtenteils dem Entwickler überlassen. Der Optimierer übernimmt eine kleinere Aufgabe: die Umformung von *endrekursiven* Funktionen zu Schleifen.

Eine Funktion heißt endrekursiv, wenn die einzigen rekursiven Aufrufe die letzten Befehle der Funktion sind. Bei imperativen Programmiersprachen bedeutet dies, dass die Rückgabewerte der rekursiven Aufrufe stets direkt zurückgegeben werden. Bei *Buggy* reicht diese Definition nicht aus, da *compound nodes* generell nur an einer Stelle einen Wert zurückgeben, der jedoch zuvor über einen `mux`-Knoten gewählt werden kann. Hier kann der zurückgegebene Wert einer endrekursiven *compound node* eine Kombination aus `mux`-Knoten und rekursiven Aufrufen sein.

Im Allgemeinen haben die endrekursiven Funktionen die in Tabelle 5.1 angedeutete Form. Die verschachtelten Bedingungen werden in *Buggy* mit `logic/mux`-Knoten umgesetzt.

Imperativ	Lisgy	*Buggy*-Graph

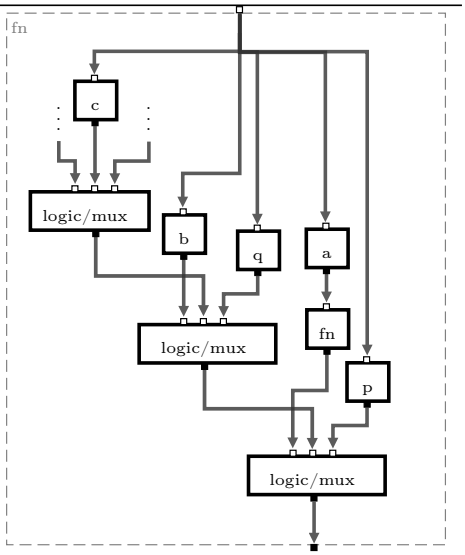
`function fn(x) {`
`  start(x)`
`  if (p(x)) {`
`  return fn(a(x))`
`  } else if (q(x)) {`
`  return b(x)`
`  } else if (...) {`
`  ...`
`  } else {`
`  return c(x)`
`  }`
`}`
	`(defco fn [x]`
	`  (logic/mux`
	`  (fn (a x))`
	`  (logic/mux`
	`  (b x)`
	`  (logic/mux`
	`  ...`
	`  (c x)`
	`  ...`
	`  )`
	`  (q x)`
	`  )`
	`  (p x)`
	`  )`
	`)`
		

Tabelle 5.1: *Struktur endrekursiver Funktionen in imperativem Programmcode, Lisgy und in Buggy-Graphen*

Die Funktion `start` steht für alle Operationen, die vor der Rückgabelogik auf den Parameter angewandt werden. In dem entsprechenden *Lisgy*-Programm und auch im *Buggy*-Graphen

entfällt diese Funktion, da Änderungen der Parameterwerte an dieser Stelle nicht möglich sind. Beispielhaft für die Prädikate stehen hier die Funktionen `p` und `q`.

Diese Teilgraphen werden, wie beim Inlining von Lambda-Funktionen, mithilfe eines rekursiven Matchers erkannt.

Solche endrekursiven Funktionen lassen sich nach einem festen Schema immer zu einer `while`-Schleife umformen. Die Beispielfunktion kann vom Optimierer in die folgende Form umgewandelt werden.

```
function fn(x) {
  x = start(x)
  while (true) {
    if (p(x)) {
      x = a(x)
      continue
    }
    if (q(x)) {
      x = b(x)
      break
    }
    ...
    x = c(x)
    break
  }
  return x
}
```

Dabei wird die Schleife unterbrochen (`break`), falls in der Ausgangsfunktion kein weiterer rekursiver Aufruf erfolgt und am Anfang fortgeführt (`continue`), falls ein rekursiver Aufruf erfolgt wäre. Das Schema ist auf beliebig viele weitere Bedingungen erweiterbar. Wird `x` als Tupel aufgefasst, ist das Schema auch auf beliebig viele Parameter erweiterbar.

Der häufig auftretende Spezialfall mit nur einer Bedingung und einem rekursiven Aufruf kann imperativ etwas kürzer implementiert werden:

```
function fn(x) {
  x = start(x)
  while (p(x)) {
    x = a(x)
  }
  return b(x)
}
```

In der Implementierung des Optimierers wird dazu ein spezieller Knoten `tailrec` eingeführt. Die Berechnungen der Prädikatfunktionen sowie die Berechnungen der Parameter für die rekursiven Aufrufe und Rückgabewerte werden in Lambda-Funktionen extrahiert. Für n verschachtelte `logic/mux`-Knoten werden auf diese Weise n Prädikat-Funktionen und $n+1$ Funktionen zur Berechnung der neuen Parameter- beziehungsweise Rückgabewerte generiert. Da es in Buggy keine Tupel gibt, nehmen alle generierten Lambda-Funktionen so viele Argumente entgegen, wie die ursprüngliche endrekursive Funktion hat, und geben jeweils neue Werte zurück.

Die Lambda-Funktionen werden über neue Eingangsports mit dem `tailrec`-Knoten verbunden. Ebenso werden Zweige, die die die initialen Werte für die Parameter berechnen, also die Vorgän-

gerknoten des Aufrufs der endrekursiven Funktion, werden ebenfalls mit dem `tailrec`-Knoten verbunden.

Zudem erhält der Knoten Informationen darüber, bei welchem Prädikat ein rekursiver Aufruf erfolgte und welcher Wert zurückgegeben wurde, sodass bei der Codegenerierung bekannt ist, ob die Schleife unterbrochen werden oder vom Anfang aus weiter ausgeführt werden muss.

Bei der Codegenerierung wird der `tailrec`-Knoten als Schleife implementiert. Die Implementierungen der Lambda-Funktionen können direkt eingefügt werden, sodass der Mehraufwand durch die Funktionsaufrufe entfällt.

Beispiel Die zuvor gezeigte Implementierung von `fac` lässt sich wie folgt in eine endrekursive Funktion umwandeln.

```
(import all)

(defco fac_tr [n acc]
  (logic/mux
     (fac_tr (- n 1) (* n acc))
     acc
     (< 1 n)
  )
)

(defco fac [n] (fac_tr n 1))
```

Die Abbildung 5.8 zeigt den resultierenden Graphen nach der Anwendung der zuvor beschriebenen Ersetzungsregel. In diesem Programm wird eine Zahl eingelesen, deren Fakultät mit der endrekursiven Funktion `fac_tr(n, acc)` berechnet und anschließend ausgegeben.

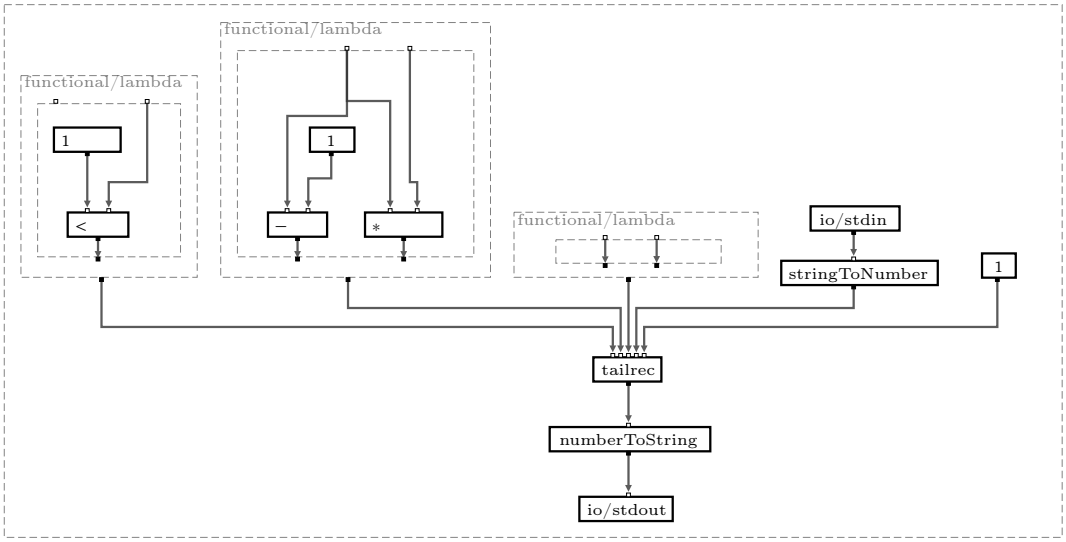


Abbildung 5.8: *Graph der umgeformten endrekursiven Implementierung der Fakultätsberechnung*

5.5.4 Umwandlung von rekursiven Funktionen zu endrekursiven Funktionen

Da der Optimierer nun endrekursive Funktionen zu Schleifen umwandeln kann, können weitere Performanzverbesserungen erzielt werden, indem einige rekursive Funktionen automatisch zu endrekursiven Funktionen umgeformt werden.

Eine solche Umwandlung ist zum Beispiel möglich, wenn eine rekursive Funktion die in Tabelle 5.2 aufgezeigte Struktur hat [Lan].

Lisgy	*Buggy*-Graph

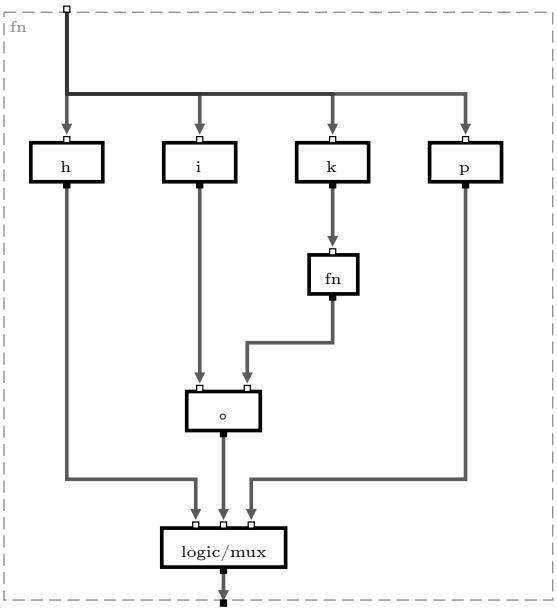
`(defco fn [x]`
`  (logic/mux`
`  (h x)`
`  (o (fn (k x)) (i x))`
`  (p x)`
`  )`
`)`
	

Tabelle 5.2: *Struktur der zu endrekursiven Funktionen umformbaren rekursiven Funktionen in Lisgy und in Buggy-Graphen*

Diese Struktur kann wie folgt etwas allgemeiner gefasst werden:

- Auf jeden rekursiven Aufruf folgt genau eine Operation ∘, deren Ergebnis zurückgegeben wird.
- Die Operation ∘ ist assoziativ.
- Es existiert ein neutrales Element e bezüglich ∘.

Mit diesen Eigenschaften können passende rekursive Funktionen im Kontrollflussgraphen erkannt werden.

Mithilfe einer Liste von assoziativen Operationen (unter anderem `math/multiply` und `array/concat`) und den zugehörigen neutralen Elementen kann die rekursive Funktion anschließend zu einer endrekursiven Funktion umgeformt werden. Es gibt dabei keine Beschränkung für die Anzahl der `logic/mux`-Knoten oder die Parameter der Funktion.

Zunächst beschränkt sich diese Ersetzungsregel auf Fälle mit nur einem `logic/mux`-Knoten. Zur Umwandlung in eine endrekursive Funktion wird der *Compound*-Knoten der Funktion um einen weiteren Eingangsport erweitert, der als Akkumulator dient. Dieser wird für den ersten Aufruf der rekursiven Funktion mit e belegt. Der zuvor rekursive Aufruf $fn(k(x))$ wird wie in [Lan] in einen endrekursiven Aufruf $fn(k(x), acc)$ umgeformt. Der neue Wert für den Akkumulator wird durch Anwendung von $\circ$ auf den aktuellen Akkumulator und $i(x)$ berechnet.

In Tabelle 5.3 sind die resultierende Funktion in *Lisgy* und der zugehörige Kontrollflussgraph angegeben. Die Funktion ist endrekursiv und kann mit der im vorherigen Abschnitt beschriebenen Regel in eine Schleife umgeformt werden.

Lisgy	*Buggy*-Graph

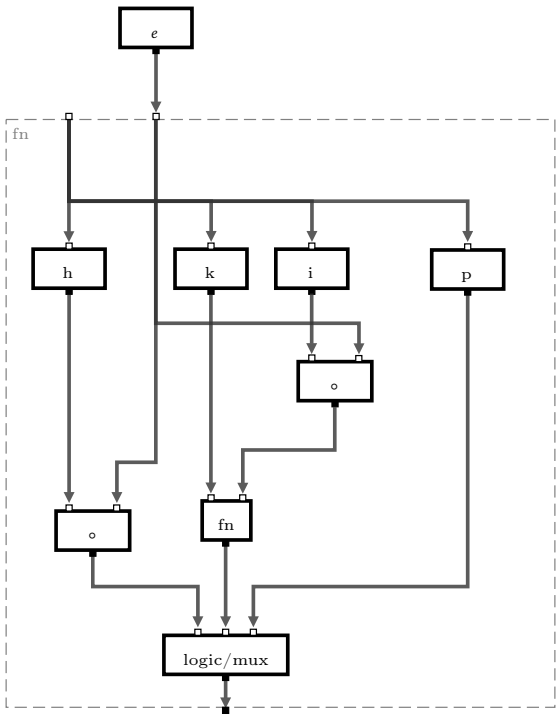
`(defco fn [x acc]`
`  (logic/mux`
`  (o (h x) acc)`
`  (fn (k x) (o (i x) acc))`
`  (p x)`
`  )`
`)`
	

Tabelle 5.3: *Struktur der umgeformten endrekursiven Funktionen in Lisgy und in Buggy-Graphen*

Mithilfe dieser Ersetzungsregel kann die eingangs angegebene rekursive Implementierung der Fakultätsfunktion bei der Optimierung automatisch in eine Schleife umgewandelt werden.

5.6 Optimierung abstrakter semantischer Strukturen

Bislang wurden Ersetzungsregeln definiert, die in erster Linie mit atomaren Knoten arbeiten und Optimierungen auf einer niedrigeren Abstraktionsebene durchführen. Die Semantik, die durch einige *Compound*-Knoten festgelegt wird, beispielsweise Sortierungen, wurde nicht betrachtet.

In diesem Abschnitt werden einige weitere Ersetzungsregeln eingeführt, die auf höherer, abstrakterer Ebene im Programmgraphen arbeiten. Da diese die *Compound*-Knoten benötigen, werden sie in der Phase *Optimierung I* verwendet.

5.6.1 Operationen auf Listen

In der Komponentenbibliothek von *Buggy* sind einige Operationen auf Listen hinterlegt. Die Semantik dieser Komponenten ist daher bekannt und kann im Optimierer anhand des Knotentyps erkannt werden. In diesem Abschnitt werden einige dieser Operationen betrachtet und Ersetzungsregeln entwickelt, die einige Fälle optimieren können.

Abbilden einer Liste

Mit dem *Compound*-Knoten `array/map` kann eine Liste auf eine neue Liste abgebildet werden. Dabei wird eine Lambda-Funktion auf jedes Listenelement angewandt, die den Wert für das Element in der neuen Liste zurückgibt. Falls unmittelbar nach einem `array/map`-Knoten ein Knoten folgt, in dem ein Teil der Liste ausgewählt wird, können die Knoten getauscht werden. Dabei wird die unnötige Berechnung von Werten eingespart, die nach der Abbildung verworfen werden. Stattdessen wird die Lambda-Funktion auf das gewählte Element aus der Liste angewandt.

Folgen mehrere `array/map`-Knoten aufeinander und wird nur das letzte Ergebnis genutzt, so können die Lambda-Funktionen verkettet werden und die Knoten zu einem einzigen `array/map`-Knoten zusammengefasst werden. Dadurch wird eine zweite Iteration über die Liste vermieden.

In *Lisgy* lassen sich diese Ersetzungen wie folgt darstellen:

```
(head (map list fn))  →  (fn (head list))
(map (map list a) b)  →  (map list (lambda [x] (b (a x))))
```

Sortieren

Wird eine Liste sortiert und anschließend das erste oder das letzte Element ausgewählt, kann die Sortierung, je nach der Sortierreihenfolge, durch eine effizientere Minimums- beziehungsweise Maximumssuche ersetzt werden.

Die Minimumssuche ist wie folgt implementiert:

```
(defco min_impl [list min] ; min ist das bislang kleinste Element der Liste
  (logic/if (array/empty list)
    min
    (min_impl (array/rest list)
      (logic/if (math/less (array/first list) min)
        (array/first list)
        min
      ) ; Berechnung des neuen kleinsten Elements
    )
  )
)

(defco min [list] (min_impl list (array/first list)))
```

Die Minimumssuche hat für eine n-elementige Liste eine Laufzeit von $O(n)$. Auf die gleiche Weise ist auch die Maximumssuche implementiert, der Vergleich wird dabei jedoch mit `math/greater` durchgeführt.

Die Ersetzungen lassen sich in *Lisgy* wie folgt darstellen:

Aufsteigende Sortierung:

```
(array/first (sort list))      →  (min list)
(array/last (sort list))       →  (max list)
```

Absteigende Sortierung:

```
(array/first (sortDesc list))  →  (max list)
(array/last (sortDesc list))   →  (min list)
```

Filtern

Mit dem `array/filter`-Knoten können Listen gefiltert werden. Dabei wird eine Prädikatfunktion in Form einer Lambda-Funktion für jedes Listenelement aufgerufen. Falls die Prädikatfunktion `true` zurückgibt, wird das jeweilige Element in die neue Liste übernommen. Diese Operation erhält die Reihenfolge der Elemente. Die Liste wird durch diese Operation häufig verkürzt.

Daher können `array/filter`-Knoten, die auf Operationen folgen, die die Elemente der Liste erhalten, vor diese Operationen verschoben werden. Bei einer Sortierung mit anschließender Filterung wird die Filterung beispielsweise vor die Sortierung verschoben. Dadurch wird die zu sortierende Liste in vielen Fällen verkleinert.

Wie zuvor bei der Abbildung von Listen können auch mehrere aufeinanderfolgende `array/filter`-Knoten zusammengefasst werden, falls nur das letzte Ergebnis genutzt wird. Die Lambda-Funktionen können hier jedoch nicht einfach verkettet werden, sondern ihre Ergebnisse müssen mit einer Konjunktion kombiniert werden. Die beiden Lambda-Funktionen können Seiteneffekte haben und die zweite Filterfunktion wird im ursprünglichen Programm nur für Elemente aufgerufen wird, die nicht von der ersten Filterfunktion gefiltert werden. Daher muss darauf geachtet werden, dass auch in der neuen Filterfunktion die Knoten der zweiten Filterfunktion nur ausgeführt werden, falls die erste Filterfunktion das Element nicht filtert. Dies kann mit einer Alternative erreicht werden, indem zwei Filterfunktion `(a x)` und `(b x)` zu `(logic/if (a x) (b x) false)` kombiniert werden.

In *Lisgy* lassen sich diese Ersetzungen wie folgt darstellen:

```
(filter (sort list) fn)     →  (sort (filter list fn))

                               (filter list
(filter (filter list a) b)  →   (lambda [x] (logic/if (a x) (b x) false))
                               )
```

Kapitel 6

Fazit

6.1 Ergebnisse

In diesem Abschnitt wird anhand einiger Beispiele demonstriert, welche Optimierungen mithilfe von Graphersetzungssystemen von dem in dieser Arbeit implementierten Optimierer durchgeführt werden können. Dabei wird insbesondere ersichtlich, dass sich erst durch die Kombination mehrerer Ersetzungsregeln komplexe Optimierungen ergeben. Da die Kontrollflussgraphen für einige Beispiele relativ groß und unübersichtlich sind, wird auf diese teilweise verzichtet. Stattdessen werden die Programme und ihre optimierten Versionen durch äquivalenten *Lisgy*-Code dargestellt.

6.1.1 Auswertung von Berechnungen

Im Abschnitt 5.2 wurden Ersetzungsregeln zur Auswertung von verschiedenen Funktionen definiert. Mit diesen ist es möglich, Programmteile während der Optimierung auszuwerten, sofern die dazu erforderlichen Werte bekannt sind.

Als Beispiel dient das folgende Programm:

```
(io/stdout
    (logic/if
        (logic/not
            (math/less
                (math/add 10 1)
                (math/multiply 10 7)
            )
        )
        "yes"
        "no"
    )
)
```

Alle Berechnungen in diesem Programm können aufgelöst werden, sodass das Programm wie folgt aussehen würde:

```
(io/stdout (logic/if false "yes" "no"))
```

Da auch `logic/if`-Knoten mit konstanter Bedingung aufgelöst werden können, sieht das vollständig optimierte Programm wie folgt aus:

```
(io/stdout "no")
```

Ein Entwickler würde derart triviale Berechnungen sicherlich bereits vorwegnehmen. Durch die Verwendung von konstanten Variablen und Komponenten fremder Entwickler können dennoch Berechnungen auftreten, die auf diese Weise entfernt werden können. Ein großer Gewinn wird dabei erzielt, falls sich durch die Berechnung einer Funktion Programmteile als unerreichbar oder nicht benötigt herausstellen und entfernt werden.

6.1.2 Optimierung von funktionalen Konzepten

Mit *Buggy* ist ein funktionaler Programmierstil möglich. Dies umfasst insbesondere die Nutzung von Lambda-Funktionen, die wie Werte verwendet werden können und das Binden von Parametern an Funktionen. Mit den in Abschnitt 5.4 eingeführten Ersetzungsregeln können einige Optimierungen durchgeführt werden. So ist es beispielsweise möglich, Funktionen mit gebundenen Parametern während der Optimierung zu berechnen. Weiterhin können statisch bekannte Lambda-Funktionen direkt in den Kontrollflussgraphen eingesetzt werden, wenn diese aufgerufen werden.

Das folgende Beispiel demonstriert, wie der Optimierer das Programm mit diesen Regeln umformt. Insgesamt werden dabei 16 Regeln angewandt, weshalb einige Zwischenschritte zusammengefasst werden. Gegeben sei der folgende *Lisgy*-Code. In diesem werden einige Parameter an Lambda-Funktionen gebunden und diese ausgewertet. Letztlich wird aber nur mit Konstanten gerechnet, sodass die Berechnungen vom Optimierer durchgeführt werden können.

```
1  (defco p [in] (partial
2      (lambda [fn] (functional/apply fn 42))
3      (partial (lambda [n m] (math/add n m)) in)
4    )
5  )
6
7  (io/stdout
8    (functional/apply
9      (lambda [x] (math/add (functional/call (p x)) 1))
10     8
11   )
12 )
```

Erläuterung zum Beispiel In der Komponente `p` wird eine Funktion erstellt, die die über den Port `in` übergebene Zahl mit `42` addiert. Dazu wird in Zeile 1 der erste Parameter der Lambda-Funktion in Zeile 2 an die Funktion gebunden, die in Zeile 3 erzeugt wird. Diese entsteht wiederum, indem der erste Parameter einer Lamba-Funktion, die ihre zwei Argumente addiert, an die über `in` übergebene Zahl gebunden wird. Die Funktion in Zeile 3 ist somit eine Funktion, die ein Argument entgegennimmt, dieses mit `in` addiert und das Ergebnis zurückgibt. Diese Funktion wird jedoch nicht direkt ausgewertet, sondern als erster Parameter an die Lambda-Funktion aus Zeile 2 gebunden. Diese Funktion wird in Zeile 9 erzeugt, wobei `x` durch `functional/apply` auf den Wert `8` gesetzt wird. Das Ergebnis der Funktion wird mit `1` addiert.

Der *Compound*-Knoten `functional/apply` ruft eine gegebene Funktion mit einem Parameter auf. Dazu nutzt der Knoten einen `functional/partial`-Knoten, um den Parameter zu binden und einen `functional/call`-Knoten, um die Funktion aufzurufen. Da die Lambda-Funktion hier bekannt ist, kann deren Implementierung direkt eingesetzt werden.

```
(defco p [in] (partial
    (lambda [fn] (functional/apply fn 42))
    (partial (lambda [n m] (math/add n m)) in)
  )
)

(io/stdout
  (math/add (functional/call (p 8)) 1)
)
```

Die Definition der Komponente `p` kann direkt an die Stelle, in der sie benutzt wird, eingesetzt werden. Der entsprechende Kontrollflussgraph ändert sich dadurch nicht, da `p` dort ohnehin über zwei Kanten mit dem restlichen Programm verbunden ist.

```
(io/stdout
  (math/add (functional/call
    (partial
      (lambda [fn] (functional/apply fn 42))
        (partial (lambda [n m] (math/add n m)) 8)
      )
    ) 1)
)
```

Im nächsten Schritt wird der Parameter `8` direkt in die Lambda-Funktion eingesetzt und die partielle Funktion somit berechnet.

```
(io/stdout
  (math/add (functional/call
    (partial
      (lambda [fn] (functional/apply fn 42))
        (lambda [m] (math/add 8 m))
      )
    ) 1)
)
```

Anschließend wird der letzte `functional/partial`-Knoten aufgelöst.

```
(io/stdout
  (math/add (functional/call
    (lambda [] (functional/apply (lambda [m] (math/add 8 m)) 42))
  ) 1)
)
```

Wie im ersten Schritt wird nun der Aufruf der Lambda-Funktion durch die Implementierung der Lambda-Funktion ersetzt. Auch der `functional/apply`-Knoten kann wie zuvor aufgelöst werden.

```
(io/stdout
  (math/add
    (math/add 8 42)
    1
  )
)
```

Im letzten Schritt können die beiden Additionen von Konstanten ersetzt werden, sodass das Programm letztlich zu `(io/stdout 51)` wird.

6.1.3 Optimierung rekursiver Funktionen

Rekursive Funktionen werden sind in *Buggy* besonders wichtig, da keine Komponenten für Schleifen existieren. Mit den in Abschnitt 5.5 eingeführten Ersetzungsregeln können einige rekursive Funktionen in endrekursive Funktionen und diese dann in Schleifen umgeformt werden.

Als Demonstration dient hier die Sortierfunktion *Quicksort* aus den Beispielen, die *Buggy* beiliegen:

```
(defco foldl [list fn init]
  (logic/if (array/empty list)
    init
    (foldl
      (array/rest list)
      fn
      (functional/apply (partial 1 fn (array/first list)) init)
    )
  ))

(defco filter [list fn]
  (foldl list (partial (lambda (fn acc cur)
    (logic/mux (array/append acc cur) acc (functional/apply fn cur))) fn) []))

(defco partition-left [list p]
  (filter list (partial (lambda (n m) (math/less m n)) p)))

(defco partition-right [list p]
  (filter list (partial (lambda (n m) (math/less n m)) p)))

; Quicksort, als Pivot-Element dient das erste Element der Liste
(defco quicksort [list]
  (logic/if (array/empty list)
    list
    (array/concat
      (array/append
        (quicksort
          (partition-left (array/rest list) (array/first list)
        ) ; sortieren der Elemente links vom Pivot-Element
        (array/first list)
      )
      (quicksort
        (partition-right (array/rest list) (array/first list))
      ) ; sortieren der Elemente rechts vom Pivot-Element
    )
  )
)
```

Dieses Programm nutzt den *foldl*-Operator zur Implementierung der Filterfunktion, der in [Hut99, S. 367] beschrieben ist. Dessen Implementierung ist endrekursiv und kann daher zu einer Schleife optimiert werden. Die Funktion `filter` erstellt eine neue Liste mit allen Elementen aus einer Liste, für die die angegebene Funktion `true` zurückgibt.

Im Folgenden wird die Ausführungsgeschwindigkeit des unoptimierten Programmes mit der des optimierten Programmes verglichen. Dazu werden beide Programme mit `buggy compile <dateiname> golang -s` zu Go-Programmen transpiliert und anschließend mit `go build <dateiname>` zu einer ausführbaren Anwendungen kompiliert. Um die optimierte Version zu generieren, wird das Programm mit `buggy compile <dateiname> golang -s -optimize` transpiliert.

Abbildung 6.1 zeigt die Laufzeiten des optimierten und des unoptimierten Programms für die Listen. Bei 1.000.000 Elementen funktioniert die unoptimierte Version nicht, da der Stack überläuft. Dies passiert bei der optimierten Version nicht, da die Rekursion in eine Iteration umgeformt wird. Die optimierte Version braucht außerdem nur etwa ein Drittel der Zeit, um eine Liste gleicher Länge zu sortieren.

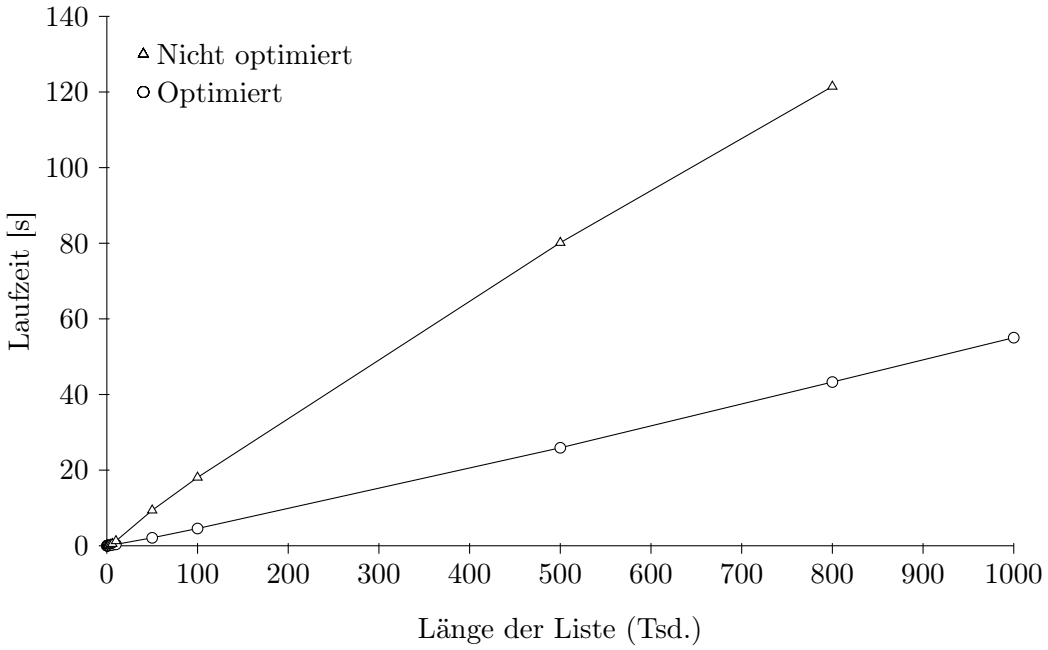


Abbildung 6.1: *Laufzeit des Quicksort-Programms bei verschiedenen Listengrößen*

6.1.4 Optimierung abstrakter semantischer Strukturen

Mit den in Abschnitt 5.6 definierten Regeln ist es möglich, Ersetzungen auf einer semantisch abstrakteren Ebene durchzuführen. Dadurch können Operationen getauscht, kombiniert oder durch effizientere Implementierungen ersetzt werden.

Das Programm in Abbildung 6.2 liest eine Liste von Zahlen ein und sortiert diese aufsteigend. Anschließend werden alle Elemente aus der Liste gefiltert, die kleiner oder gleich 10 sind. Zuletzt werden alle Elemente dieser Liste mit 2 multipliziert und das erste Element wird ausgegeben.

```
(io/stdout (array/first
  (map
    (filter
      (sort (io/stdin))
      (lambda [x] (math/less 10 x))
    )
    (lambda [x] (math/multiply x 2)))
))
```

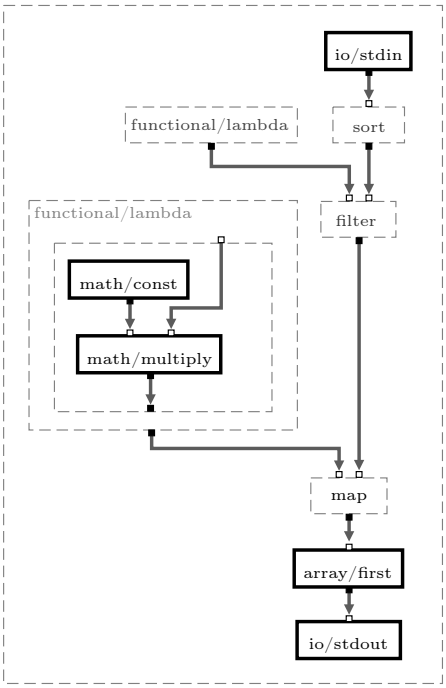
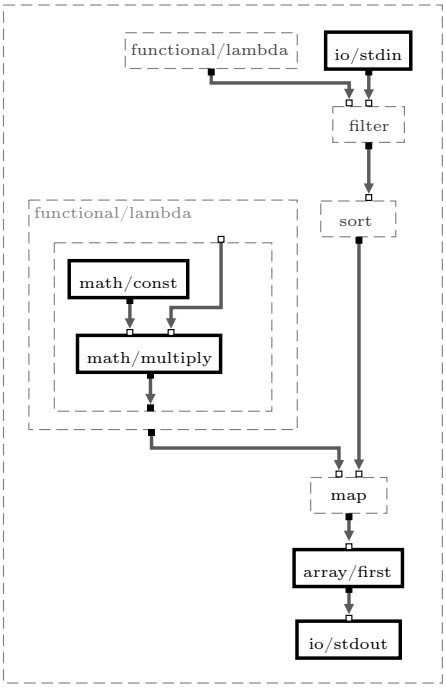


Abbildung 6.2: *Ein Beispielprogramm, das mit einer Liste arbeitet in Lisgy und der zugehörige vereinfachte Kontrollflussgraph*

Die Abbildung 6.3 zeigt die einzelnen Schritte der Optimierung. Das resultierende Programm iteriert nur noch einmal über die Liste. Außerdem ist die Minimumssuche endrekursiv und kann von weiteren Regeln zu einer Schleife transformiert werden.

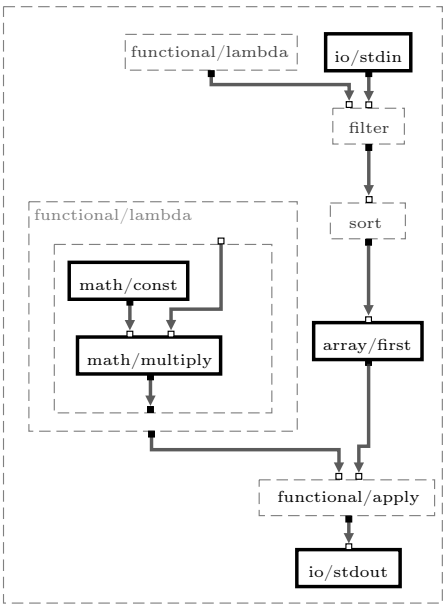
Zunächst werden die Sortierung und die Filterung getauscht. Dadurch wird die zu sortierende Liste häufig kleiner.

```
(io/stdout (array/first
  (map
    (sort
      (filter
        (io/stdin)
        (lambda [x] (math/less 10 x))
      )
    )
    (lambda [x] (math/multiply x 2)))
))
```



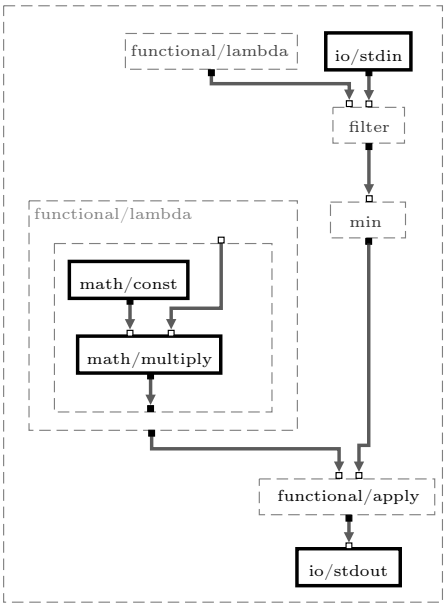
Da lediglich das erste Element der Liste verwendet wird, wird die Funktion, mit der die Elemente im `map`-Knoten abgebildet werden, nur auf das erste Element angewandt.

```
(io/stdout (functional/apply
  (lambda [x] (math/multiply x 2)))
  (array/first
    (sort
      (filter
        (io/stdin)
        (lambda [x] (math/less 10 x))
      )
    )
  )
))
```



Auf den `sort`-Knoten, der die Liste aufsteigend sortiert, folgt unmittelbar ein `array/first`-Knoten, der das erste Element der Liste auswählt. Diese beiden Operationen werden durch eine Minimumssuche ersetzt, die eine bessere Laufzeit hat.

```
(io/stdout (functional/apply
  (lambda [x] (math/multiply x 2)))
  (min
    (filter
      (io/stdin)
      (lambda [x] (math/less 10 x))
    )
  )
))
```



Im letzten Schritt kann die Implementierung der Lambda-Funktion direkt eingesetzt und der `functional/apply`-Knoten somit entfernt werden.

```
(io/stdout (math/multiply
  (min
    (filter
      (io/stdin)
      (lambda [x] (math/less 10 x))
    )
  )
  2
))
```

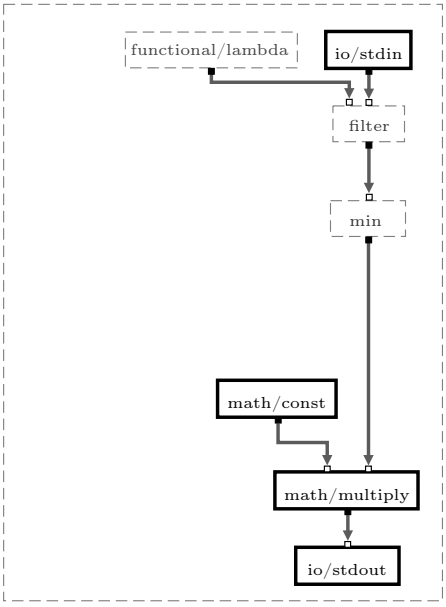


Abbildung 6.3: *Schrittweise Optimierung des Programms aus Abbildung 6.2*

6.2 Ausblick

Mit Abschluss dieser Arbeit ist die Entwicklung eines Optimierers für *Buggy*-Programme noch nicht abgeschlossen. Da sich *Buggy* derzeit schnell weiterentwickelt, werden immer mehr neue Optimierungen ermöglicht und bestehende Ersetzungsregeln müssen angepasst werden. Die implementierten Ersetzungsregeln bieten jedoch eine gute Grundlage. Insbesondere die Optimierung von Endrekursion ist wichtig, da einige Programme dadurch deutlich performanter sind oder ihre Ausführung mit größeren Eingabewerten andernfalls nicht möglich wäre.

Während der weiteren Entwicklung von *Buggy* wird sich zeigen, in welchen Bereichen Optimierungen nötig sind. Rekursive Funktionen werden häufig benutzt, insbesondere hier könnten weitere Fälle optimiert werden.

Optimierungen auf semantisch abstrakterer Ebene wurden in dieser Arbeit nur kurz thematisiert. Später wäre hierfür eine Datenbank denkbar, die äquivalente Implementierungen speichert, auf die der Optimierer dann für Ersetzungen zurückgreifen kann. In dieser Datenbank könnten auch semantische Eigenschaften von Komponenten gespeichert werden, wie beispielsweise neutrale Elemente und arithmetische Eigenschaften, die für einige Optimierungen benötigt werden.

Die Implementierung einiger Ersetzungsregeln ist teilweise sehr aufwendig. Zukünftig kann versucht werden, für diese eine höhere Abstraktionsebene zu finden. Mit den Matchern und den vorgegebenen Ersetzungsfunktionen wurde dafür bereits eine Grundlage geschaffen.

Wie auch alle anderen Programme der *Buggy*-Plattform wird auch der entwickelte Optimierer quelloffen im Internet zur Verfügung stehen. Somit kann der Optimierer von der Mitarbeit der anderen an *Buggy* beteiligten Personen und auch der Anwender profitieren.

Anhang

A.1 Referenz der Lisgy-Befehle

Viele Beispiele in dieser Arbeit nutzen *Lisgy*. Neben den *Buggy*-Komponenten bietet *Lisgy* auch einige weitere Befehle, die nachfolgend mit Beschreibungen aufgeführt sind.

`defco`	Erzeugt einen neuen *Compound*-Knoten, gegebenenfalls mit Eingangsports. `(defco cmp [x y] ...)` erzeugt einen *Compound*-Knoten mit dem Namen `cmp` mit den beiden Eingangsports `x` und `y`.
`lambda`	Erzeugt eine Lambda-Funktion. `(lambda [x y] ...)` erzeugt eine Lambda-Funktion mit den beiden Argumenten `x` und `y`, in dieser Reihenfolge.
`import`	Importiert *Lisgy*-Code aus vordefinierten Paketen. Mit `(import all)` ist unter anderem die Verwendung von $*$ als Alias für `math/multiply` möglich.
`partial`	Bindet einen Parameter an eine Lambda-Funktion. Dieser Befehl ist ein Alias für die *Buggy*-Komponente `functional/partial`. Mit `(partial fn 1 x)` wird die Variable `x` als zweiter Parameter (Index `1`) an die Lambda-Funktion `fn` gebunden. Wird kein Index angegeben, wird `0` als Index benutzt: `(partial fn x)` ist äquivalent zu `(partial fn 0 x)`.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[AK07] Oana Andrei and Hélène Kirchner. A Rewriting Calculus for Multigraphs with Ports. In *The Eighth International Workshop on Rule-Based Programming (RULE 2007)*, page 20, Paris, France, June 2007. ENTCS.

[Ass00] Uwe Assmann. Graph rewrite systems for program optimization. *ACM Trans. Program. Lang. Syst.*, 22(4):583–637, July 2000.

[BFG95] D. Blostein, H. Fahmy, and A. Grbavec. Practical use of graph rewriting. Technical Report 95-373, Queen's University, CDN, 1995.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[Dö95] Heiko Dörr. *Efficient Graph Rewriting and Its Implementation*. Springer Berlin Heidelberg, Secaucus, NJ, USA, 1995.

[FSFa] Free Software Foundation. Gnu compiler collection (gcc) internals. Dieses Dokument ist verfügbar unter `http://gcc.gnu.org/onlinedocs/gccint`. Abgerufen am 18. Juli 2016.

[FSFb] Free Software Foundation. Using the gnu compiler collection (gcc). Dieses Dokument ist verfügbar unter `http://gcc.gnu.org/onlinedocs/gcc`. Abgerufen am 18. Juli 2016.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, July 1999.

[Lan] Hans Werner Lang. Umwandlung von rekursion in endrekursion. Dieses Dokument ist verfügbar unter `http://www.inf.fh-flensburg.de/lang/prog/endrekursion-formal.htm`. Abgerufen am 2. August 2016.

[Nov06] Diego Novillo. Gcc–an architectural overview, current status, and future directions. In *Proceedings of the 2006 Linux Symposium, Volume Two*, pages 185–200, 2006.

[Plu99] Detlef Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, chapter 1, pages 3–61. World Scientific, River Edge, NJ, USA, 1999.

[Ris73] Tore Risch. *REMREC - A program for Automatic Recursion Removal in LISP*. Uppsala Univerity (Datalogilaboratoriet), 1973.

[SDF+09] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19:1–301, 8 2009.