Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

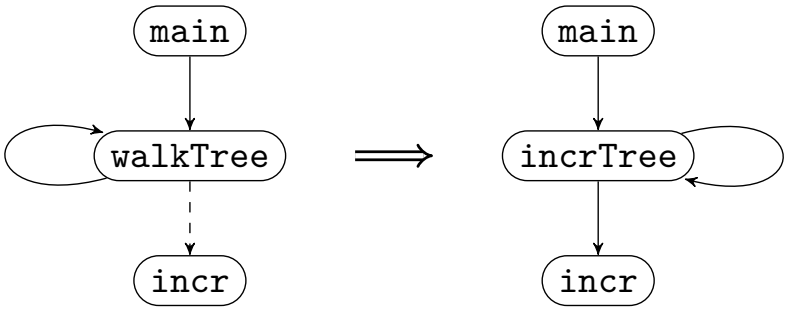
Lehrstuhl Prof. Dr.-Ing. Snelting

Procedure Cloning im Kontext SSA-basierter Zwischensprachen

Bachelorarbeit von

Raphael von der Grün

an der Fakultät für Informatik



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

Bearbeitungszeit: 9. August 2016 – 8. Dezember 2016

Zusammenfassung

Funktionsaufrufe stellen in der Programm-Analyse eine natürliche Barriere dar. Die Optimierung *Procedure Cloning* eignet sich zur Überwindung dieser Barriere. Dieses Verfahren spezialisiert Prozeduren für im Programm vorhandene Aufrufe. Die resultierenden Spezialisierungen können von mehreren Aufrufern gemeinsam verwendet werden. Dadurch wächst das Programm nicht so schnell wie es beim *Inlining* der entsprechenden Aufrufe der Fall wäre. Wir stellen unsere Implementierung von Procedure Cloning in libFirm vor. libFirm ist eine Compiler-Bibliothek, welche mit einer graphbasierten Zwischensprache in SSA-Form arbeitet. Außerdem analysieren wir die Komplexität einer optimalen Klonauswahl. Abschließend werden die Auswirkungen von Procedure Cloning mit der SPEC CPU2006 Benchmark-Suite untersucht.

Inhaltsverzeichnis

1 Einführung

Funktionsaufrufe stellen in der Programmoptimierung eine besondere Herausforderung dar. Wissen über den Programmzustand an der Aufrufstelle kann nicht ohne Weiteres genutzt werden, um den Code der aufgerufenen Funktion zu optimieren, da diese im Allgemeinen von beliebig vielen weiteren Kontexten aus aufgerufen werden kann.

Ein einfacher und weit verbreiteter Ansatz zur Lösung dieses Problems ist *Inlining*. Der Programmcode der aufgerufenen Funktion wird an der Aufrufstelle eingefügt und ersetzt den ursprünglichen Funktionsaufruf. Hierdurch existiert an jeder Aufrufstelle eine eigene Kopie der ursprünglichen Funktion, welche unter Verwendung des jeweiligen lokalen Kontextes beliebig optimiert werden kann. Ein weiterer positiver Effekt dieses Vorgehens ist, dass der Overhead für den Funktionsaufruf selbst, etwa das Sichern von Registern oder die Vorbereitung des Stacks, wegfällt.

Ein großer Nachteil dieser Herangehensweise ist das starke Wachstum der Programmgröße. Selbst wenn eine Funktion von zehn Stellen im Programm aus mit identischem Kontext aufgerufen wird, muss beim Inlining für jede Stelle eine eigene Kopie der fraglichen Funktion erstellt werden. Zum einen muss nun jede Kopie der Funktion separat optimiert werden, zum anderen wächst das Programm um das Zehnfache der optimierten Funktionsgröße.

Eine Lösung für das eingangs beschriebene Problem, welche die oben genannten Nachteile des Inlining vermeidet, ist das sogenannte *Procedure Cloning*. Anstatt, wie beim Inlining das vom Entwickler mühsam betriebene Ausfaktorisieren der Funktionen aus dem aufrufenden Code rückgängig zu machen, werden Kopien der aufgerufenen Funktionen erzeugt, welche unter Berücksichtigung des jeweiligen Aufrufkontextes optimiert werden. Diese Spezialisierungen werden dann anstelle des ursprünglichen Ziels aufgerufen. Im Gegensatz zum Inlining kann eine einzige spezialisierte Prozedur an allen Stellen mit gleichem Kontext verwendet werden. Dafür müssen zwar die Aufrufe erhalten bleiben und folglich deren Laufzeitkosten bezahlt werden. Der Aufwand für Funktionsaufrufe ist in der Praxis jedoch meist nicht relevant [1].

Um Procedure Cloning sinnvoll anwenden zu können, sind jedoch umfangreichere Analysen notwendig als beim Inlining. Diese können dann aber auch verwendet werden, um die ertragreichsten Stellen zum Klonen zu finden. Anstatt also einfach

auf gut Glück alle Funktionen zu klonen, welche häufig mit ähnlichen Parametern aufgerufen werden, können etwa gezielt Funktionen ausgewählt werden, bei denen durch die Spezialisierung ein bedingter Sprung wegfällt.

Abbildung 1.1a zeigt ein einfaches Programm zur Baum-Traversierung. Betrachtet man die Prozedur `walk`, lässt sich erhebliches Optimierungspotential vermuten: Die Parameter `pre` und `post` sind in jedem vorhandenen Aufruf der Prozedur gleich. Somit können alle Aufrufe statt der originalen Prozedur die Spezialisierung `incrTree` aufrufen. Bei jedem Aufruf von `incrTree` müssen im Vergleich zu `walk` zwei bedingte Sprünge weniger ausgeführt werden. Außerdem kann der indirekte Aufruf von `incr` über `pre` durch einen direkten Aufruf ersetzt werden.

```
1  def incr(n):
2    n.count += 1
3
4  def walk(n, pre, post):
5    if not n: return
6    if pre: pre(n)
7    walk(n.left, pre, post)
8    walk(n.right, pre, post)
9    if post: post(n)
10
11 def main():
12   root = randomTree()
13   walk(root, incr, None)
```

(a) Ausgangsprogramm zur Baum-Traversierung

```
1  def incr(n):
2    n.count += 1
3
4  def incrTree(n):
5    if not n: return
6    incr(n)
7    incrTree(n.left)
8    incrTree(n.right)
9
10
11 def main():
12   root = randomTree()
13   incrTree(root)
```

(b) Ergebnis der Anwendung von Procedure Cloning auf Abbildung 1.1a

Abbildung 1.1: Anwendung von Procedure Cloning auf einen einfachen Tree-Walker.

Beide Optimierungen bieten nicht nur zur Laufzeit Vorteile, sondern verfeinern auch die Aussagen die durch weitere Programm-Analysen getroffen werden können und begünstigen so nachfolgende Optimierungen. Beispielsweise kann nun Inlining verwendet werden, um den Aufruf von `incr` in `incrTree` zu eliminieren. Dies zeigt auch, dass sich Procedure Cloning und Inlining nicht ausschließen müssen. Im besten Fall werden beide Verfahren gemeinsam und ihren jeweiligen Stärken entsprechend eingesetzt.

2 Grundlagen

Der erste Teil dieses Kapitels beschreibt die in dieser Arbeit verwendeten mathematischen Konzepte und führt die verwendete Notation ein. Der zweite Teil besteht aus einer kurzen Einführung in die Art der Programmrepräsentation auf der unsere Algorithmen arbeiten.

2.1 Mathematische Grundlagen

Die vorliegende Arbeit setzt Grundlagenwissen aus dem Bachelor-Studium Informatik voraus. Nichtsdestotrotz werden wir im Folgenden die wichtigsten verwendeten Konzepte knapp definieren, auch um Klarheit bei der Notation zu schaffen.

Viele der hier verwendeten grundlegenden Definitionen basieren auf Hromkovičs Buch über Algorithmen für schwere Probleme [2].

Wir werden fortan folgende grundlegende Mengen verwenden:

$\mathbb{N}$	$= \{1, 2, 3, \ldots\}$	Die Menge der natürlichen Zahlen,
$\mathbb{N}_0$	$= \mathbb{N} \cup \{0\}$	Die Menge der natürlichen Zahlen mit Null,
$\mathbb{R}$		Die Menge der reellen Zahlen,
$\mathbb{R}_+$	$= \{n \in \mathbb{R} \mid n > 0\}$	Die Menge der positiven reellen Zahlen,
$\mathbb{R}_{\geq 0}$	$= \mathbb{R}_+ \cup \{0\}$	Die Menge der nicht-negativen reellen Zahlen,

Für eine beliebige Menge M, bezeichnet $\mathcal{P}(M)$ die *Potenzmenge* von M:

$$\mathcal{P}(M) = \{A \mid A \subseteq M\}.$$

Für eine beliebige Menge M und $n \in \mathbb{N}$ definieren wir außerdem die n-te Potenz als das Ergebnis der n-maligen Anwendung des Kreuzproduktes auf sich selbst:

$$M^n = \underbrace{M \times \cdots \times M}_{n \text{ Operanden}}.$$

Wir definieren die Vereinigung zweier *disjunkter* Mengen A und B:

$$A \cap B = \varnothing \;\Rightarrow\; A \dot\cup B = A \cup B.$$

Diesen Operator werden wir primär verwenden um auszudrücken dass seine Operanden disjunkt sind.

Wir erweitern Funktionen gelegentlich implizit auf die Potenzmengen ihrer Definitions- und Zielmengen. Gegeben $f\colon A \to B$ und $X \subseteq A$ gilt dann

$$f(X) = \{f(x) \mid x \in X\}.$$

2.1.1 Graphen

Definition 1 (Gerichteter Graph)**.** Ein *(gerichteter) Graph* G ist definiert durch ein Zweitupel (V, E), wobei

i) V die endliche Menge der *Knoten* von G,
ii) $E \subseteq V^2$ die Menge der *Kanten* von G

darstellen. Wir werden die Knoten auch manchmal als $V(G) = V$ und die Kanten als $E(G) = E$ bezeichnen. Die Kante $(u, v) \in E$ heißt *inzident* zu den Knoten u und v.

Da wir im Folgenden ausschließlich gerichtete Graphen betrachten werden, sprechen wir oft auch einfach von Graphen.

Definition 2 (Teilgraph)**.** Seien $G = (V, E)$ und $\widehat{G} = (\widehat{V}, \widehat{E})$ Graphen. Dann ist $\widehat{G}$ ein Teilgraph von G wenn Knoten und Kantenmengen der beiden Graphen in einer Teilmengenbeziehung stehen. Das heißt

$$\widehat{G} \leq G :\Leftrightarrow\; \widehat{V} \subseteq V \wedge \widehat{E} \subseteq E.$$

Wobei $\leq$ die Teilgraph-Relation darstellt.

Definition 3 (Induzierter Teilgraph)**.** Sei $G = (V, E)$ ein Graph und $\widehat{V} \subseteq V$. Dann ist $G[\widehat{V}]$ der induzierte Teilgraph von G, der alle Knoten aus $\widehat{V}$ und die zu ihnen inzidenten Kanten aus E enthält. Das heißt

$$G[\widehat{V}] := (\widehat{V}, E \cap \widehat{V}^2).$$

Offensichtlich gilt $G[\widehat{V}] \leq G$.

Definition 4 (Weg)**.** Sei $G = (V, E)$ ein Graph. Ein *Weg* W in G ist ein Folge von durch Kanten verbundenen Knoten. Das heißt

$$\begin{aligned} W &= v_1, v_2, \ldots, v_n \\ &\text{mit } v_i \in V && \text{für } i = 1, \ldots, n \\ &\text{und } (v_i, v_{i+1}) \in E && \text{für } i = 1, \ldots, n-1. \end{aligned}$$

Genauer ist W ein Weg von v_1 nach v_n. Gilt $v_1 = v_n$ mit $n \geq 1$ dann ist W ein *Kreis* oder *Zyklus*. Graphen welche keinen Kreis enthalten, heißen *azyklisch*.

Definition 5 (Wurzeln)**.** Sei $G = (V, E)$ ein Graph und $r \in V$. Dann nennen wir r eine *Wurzel* von G genau dann wenn es in G für jeden Knoten $v \in V$ einen Weg von r nach v gibt.

Definition 6 (Vorgänger und Nachfolger)**.** Sei $G = (V, E)$ ein Graph und $v \in V$. Dann beschreibt $N_G^-(v)$ die Menge der *Vorgänger* und $N_G^+(v)$ die Menge der *Nachfolger* von v.

$$\begin{aligned} N_G^-(v) &= \{u \in V \mid (u, v) \in E\} \\ N_G^+(v) &= \{u \in V \mid (v, u) \in E\} \end{aligned}$$

Definition 7 (Grad)**.** Sei $G = (V, E)$ ein Graph und $v \in V$. Dann beschreibt der *Eingangsgrad* $d_G^-(v)$ die Anzahl der in v eingehenden und der *Ausgangsgrad* $d_G^+(v)$ die Anzahl der von v ausgehenden Kanten.

$$\begin{aligned} d_G^-(v) &= |N_G^-(v)| \\ d_G^+(v) &= |N_G^+(v)| \end{aligned}$$

Definition 8 (Bäume und Wälder)**.** Sei $G = (V, E)$ ein azyklischer Graph. G heißt *Wald* wenn der maximale Eingangsgrad aller Knoten höchstens 1 ist. Hat G außerdem eine Wurzel, so ist G ein *Baum*.

Definition 9 (Spannwald und Spannbaum)**.** Sei $G = (V, E)$ ein Graph. $F = (V, E_F)$ mit $E_F \subseteq E$ heißt *Spannwald* von G wenn F ein Wald ist. Ist F ein Baum, heißt F auch *Spannbaum* von G. Wird F durch eine Tiefensuche auf G erzeugt, so heißt F *Tiefensuchwald* bzw. *Tiefensuchbaum*.

Definition 10 (Topologische Sortierung)**.** Sei $G = (V, E)$ ein azyklischer Graph. Eine Ordnungsrelation $\leq$ auf V ist eine topologische Sortierung von G wenn für alle $(u, v) \in E$ gilt dass $u \leq v$. Eine topologische Sortierung der Knoten existiert genau dann wenn ein Graph azyklisch ist.

Des Weiteren verwenden wir in einigen Algorithmen die *Reverse Postorder* als Abarbeitungsreihenfolge der Knoten eines Graphen. Sie stellt gewissermaßen eine

Approximation von einer topologischen Ordnung für beliebige Graphen dar. Auf azyklischen Graphen ist sie eine topologische Ordnung. Ein einfacher Algorithmus zur Erzeugung einer Reverse Postorder, sowie eine ausführliche formale Behandlung ihrer Eigenschaften finden sich bei Nielsen [3, S. 421–426]

2.1.2 Entscheidungs- und Optimierungsprobleme

Die folgenden Definitionen bilden die Grundlage für die Behandlung von Entscheidungs- und Optimierungsproblemen.

Definition 11 (Alphabet)**.** Wir nennen eine endliche, nicht-leere Menge Σ *Alphabet*. Die Elemente des *Alphabets* werden Symbole genannt. Für uns wird das *binäre Alphabet* $\Sigma_{bin} = \{0, 1\}$ von besonderem Interesse sein.

Definition 12 (Wörter)**.** Sei Σ ein Alphabet. Eine endliche Folge w von Symbolen aus Σ nennen wir *Wort über* Σ. $|w|$ bezeichnet die Länge dieser Folge. Das Wort ε mit $|\varepsilon| = 0$ nennen wir *leeres Wort*. Σ^* bezeichnet die Menge aller Wörter über dem Alphabet Σ.

Definition 13 (Sprache)**.** Sei Σ ein Alphabet. Dann heißt jedes $L \subseteq \Sigma^*$ *Sprache über* Σ. Σ^*_{bin} nennen wir die *binäre Sprache*.

Definition 14 (Entscheidungsproblem)**.** Wir definieren ein *Entscheidungsproblem* L als eine Sprache über Σ^*_{bin}. Ein Algorithmus *entscheidet* L, wenn er für jedes $w \in \Sigma^*_{bin}$ in endlicher Zeit feststellt ob $w \in L$ oder $w \notin L$ gilt.

Definition 15 (Binärrepräsentation)**.** Für eine beliebige Struktur X stellt $\langle X \rangle \in \Sigma^*_{bin}$ eine kompakte[1] *Binärrepräsentation* von X dar.

Betrachten wir dazu zwei Beispiele. Seien dazu $G = (V, E)$ ein Graph und $s, t \in V$ Knoten in G. Dann lässt sich das Entscheidungsproblem über die Existenz eines Weges von s nach t in G formulieren als

$$\textsc{Path} = \{\langle G, s, t\rangle \mid \text{Es existiert ein Weg von } s \text{ nach } t \text{ in } G\}.$$

Ein weiteres Beispiel ist das Problem, ob ein Graph eine gegebene Wurzel besitzt:

$$\textsc{Root} = \{\langle G, r\rangle \mid G \text{ ist gerichteter Graph mit Wurzel } r\}.$$

[1] Mit „kompakt“ ist gemeint dass die Repräsentation nicht verschwenderisch sein sollte. Eine unäre Kodierung einer Zahl wäre beispielsweise nicht kompakt.

Definition 16 (Optimierungsproblem)**.** Ein *Optimierungsproblem* ist ein Tupel $U = (I, \mathcal{M}, w_I, g)$ mit folgenden Bestandteilen

i) $I \subseteq \Sigma_{bin}^*$ ist die Menge der *Probleminstanzen*,
ii) $\mathcal{M}\colon I \to \mathcal{P}(\Sigma_{bin}^*)$ bildet eine Instanz auf ihre *zulässige Lösungsmenge* ab,
iii) $w_I = \{w_x\colon \mathcal{M}(x) \to \mathbb{R} \mid x \in I\}$ ist die Menge der *Bewertungsfunktionen*,
iv) $g \in \{\min, \max\}$ die *Optimierungsrichtung*.

Für $x \in I$ heißt eine zulässige Lösung $y \in \mathcal{M}(x)$ *optimal*, genau dann wenn

$$w_x(y) = g\left\{w_x(z) \mid z \in \mathcal{M}(x)\right\}.$$

Den Wert einer optimalen Lösung bezeichnen wir mit $Opt_U(x)$. $OptSols_U(x) \subseteq \mathcal{M}(x)$ bezeichnet die Menge aller optimalen Lösungen.

2.2 Repräsentation von Programmen

2.2.1 SSA und FIRM

Moderne Compiler führen eine Vielzahl von Analysen und Optimierungen auf ihren Eingabeprogrammen aus. Im Allgemeinen eignet sich die Struktur der Eingabeprogramme nicht besonders gut dafür. Deshalb überführen moderne Compiler die verarbeiteten Programme für die Durchführung von Analysen und Optimierungen in eine Zwischensprache, die sogenannte *Intermediate Representation* [4].

Heute basieren diese Zwischenrepräsentationen häufig auf einer *Static Single Assignment*-Form (SSA-Form) des Eingabeprogramms. Ein Programm liegt genau dann in SSA-Form vor, wenn für jede Variable exakt eine Zuweisung im Programm existiert [5]. Die SSA-Form vereinfacht durch ihre Eigenschaften diverse Analysen und Optimierungen [6]. Natürlich liegen Eingabeprogramme normalerweise nicht schon in dieser Form vor. Eine effiziente Überführung in SSA-Form ist aber, etwa mit dem Verfahren von Braun et al. [7], möglich.

Ein einfacher erster Schritt der SSA-Konstruktion ist es, die verwendeten Variablen fortlaufend zu nummerieren. Dadurch existiert nur noch eine Zuweisung pro Variable. Das Problem dabei ist, dass der zugewiesene Wert im Allgemeinen nicht statisch berechenbar ist, wenn er vom Kontrollfluss des Programms abhängt.

Betrachten wir dazu Abbildung 2.1a. Hier ist nicht klar, ob der Aufruf in Zeile 6 für `x` die Definition aus Zeile 2 oder 4 verwendet. Um die Abhängigkeit vom Kontrollfluss abzubilden, werden sogenannte ϕ-Funktionen eingesetzt. In Zeile 5 von

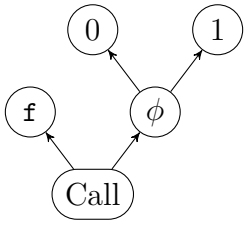
Abbildung 2.1a wird deshalb eine neue Variable `x3` eingeführt, welche abhängig von der Programmausführung entweder den Wert von `x1` oder `x2` enthält. Diese kann nun unabhängig vom tatsächlichen Kontrollfluss als Argument für den Aufruf an `f` in Zeile 6 verwendet werden.

```
1 if ... :
2   x = 0
3 else:
4   x = 1
5
6 f(x)
```

(a) Eingabeprogramm

```
1 if ... :
2   x1 = 0
3 else:
4   x2 = 1
5 x3 = φ(x1, x2)
6 f(x3)
```

(b) Programm in SSA-Form



(c) Vereinfachter Graph des Programms

Abbildung 2.1: SSA-Transformation für ein einfaches Programm

Liegt ein Programm in SSA-Form vor, so kann der Datenfluss als Abhängigkeitsgraph dargestellt werden. Ein vereinfachtes Beispiel eines Abhängigkeitsgraphen für das Programm aus Abbildung 2.1b ist in Abbildung 2.1c gegeben. Hier kann man gut erkennen, dass Variablen nicht mehr notwendig sind. Um vollständige Programme abbilden zu können, fehlen allerdings noch Konzepte für die Darstellung von Kontrollfluss und Nebeneffekten durch Speicherzugriffe.

Die im Folgenden beschriebenen Analysen und Optimierungen wurden im Rahmen von libFirm [8] implementiert. libFirm ist eine C-Bibliothek für Compiler-Optimierungen, welche die Zwischensprache Firm [9] implementiert. Firm stellt eine Variante des „Sea of nodes“-Konzeptes nach Click [10] dar, welches neben Datenfluss auch Kontrollfluss und Nebeneffekte abbilden kann. Auf die genaue Ausprägung werden wir hier nicht weiter eingehen, da wir im Folgenden nur auf dem Datenflussgraphen arbeiten werden.

2.2.2 Formale Definition von Programmen

Vorweg eine kurze Anmerkung zur verwendeten Terminologie. Da sich die vorliegende Arbeit mit der Spezialisierung von Unterprogrammen beschäftigt, werden wir diese sehr häufig referenzieren. Wir werden hier, vor allem aufgrund des etablierten Namens der implementierten Optimierung, von *Prozeduren* reden. Dieser Name wird unabhängig von der Tatsache verwendet, ob das Unterprogramm Nebeneffekte hat oder einen Wert zurück liefert.

Bevor wir die verwendeten Algorithmen im Detail betrachten können, müssen wir zunächst die Strukturen definieren auf denen sie arbeiten. Wir analysieren und optimieren ein gegebenes Programm Λ. Für uns ist ein Programm schlicht eine endliche

Menge an Prozeduren. Wenn also **Procs** die Menge aller möglichen Prozeduren bezeichnet, dann gilt $\Lambda \in \mathcal{P}(\mathbf{Procs})$. Wir bezeichnen $\mathcal{P}(\mathbf{Procs})$ auch als **Progs**. Ein Programm muss weder eigenständig sein, noch einen einzelnen bekannten Einstiegspunkt haben. Es könnte sich also genau so gut um eine Bibliothek oder eine Compilation Unit handeln.

Seien im Folgenden $p, q \in \mathbf{Procs}$. Eine Prozedur $p = (\ell_p, G_p)$ wird durch ihren Namen ℓ_p eindeutig identifiziert. Ihr Verhalten wird durch ihren Prozedur-Graphen $G_p = (\mathbf{Ops}_p, \mathbf{Deps}_p)$ bestimmt. Dieser liegt als Abhängigkeitsgraph von Ausdrücken in SSA-Form vor. Hierbei stellen die Operationen $\mathbf{Ops}_p$ die Knoten des Graphen und die Kanten $\mathbf{Deps}_p \subseteq \mathbf{Ops}_p^2$ die Abhängigkeiten zwischen ihnen folgendermaßen dar: Seien $o_1, o_2 \in \mathbf{Ops}_p$, dann gilt $(o_1, o_2) \in \mathbf{Deps}_p$ genau dann wenn o_1 das Ergebnis von o_2 verwendet. Für eine Operation $o \in \mathbf{Ops}_p$ bezeichnen wir die von o verwendeten Operationen mit $opr_p(o) = N^+_{G_p}(o)$, die Operationen die o verwenden mit $usr_p(o) = N^-_{G_p}(o)$.

Entfernen wir von einer normalerweise auf eine Prozedur p beschränkten Menge A_p ihren Index, so meinen wir damit das zugrundeliegende Universum, also $A = \bigcup_{p \in \mathbf{Procs}} A_p$. Ist der Index eine Menge von Prozeduren beziehungsweise ein Programm $\hat{\Lambda}$, dann gilt $A_{\hat{\Lambda}} = \bigcup_{p \in \hat{\Lambda}} A_p$.

Die Menge der Operationen lässt sich nach dem Typ der enthaltenen Knoten folgendermaßen partitionieren:

$$\mathbf{Ops} = \Phi \dot\cup \mathbf{Consts} \dot\cup \mathbf{Params} \dot\cup \mathbf{Calls} \dot\cup \cdots .$$

Hierbei ist Φ die Menge der ϕ-Knoten, **Consts** die Menge der Konstanten und **Params** die Menge der Prozedur-Parameter. **Calls** beschreibt die Menge der Prozedur-Aufrufe, für die wir statisch das Aufrufziel bestimmen können.

Elemente aus **Consts** oder **Params** haben niemals Abhängigkeiten zu anderen Knoten. Es gilt also

$$\forall p \in \mathbf{Procs} : \bigcup opr_p(\mathbf{Consts}_p \cup \mathbf{Params}_p) = \varnothing.$$

Für $\hat{\Lambda} \in \mathbf{Progs}$ und $c \in \mathbf{Calls}_{\hat{\Lambda}}$ ist $p = clr_{\hat{\Lambda}}(c) \in \mathbf{Procs}$ die aufrufende, $q = cle_{\hat{\Lambda}}(c) \in \mathbf{Procs}$ die aufgerufene Prozedur. Um in Zusammenhang mit Rekursion keine zyklischen Strukturen zu erzeugen, beinhaltet c mit ℓ_p und ℓ_q nur die *Namen* der Prozeduren p und q. Deshalb benötigen *clr* und *cle* einen Verweis auf das zugehörige Programm, um die gefragte Prozedur inklusive ihres Graphen zurückgeben zu können. Aufrufziele außerhalb des verfügbaren Programms lassen wir hier außer Acht, da ohne Kenntnis über ihren Graphen auch keine Spezialisierung stattfinden kann.

Da es häufig nötig sein wird Aufrufe nach Ziel und Ursprung zu gruppieren, definieren wir für $A, B \subseteq \hat{\Lambda}$

$$\mathbf{Calls}_{A \to B} := \{c \in \mathbf{Calls} \mid clr_{\hat{\Lambda}}(c) \in A \wedge cle_{\hat{\Lambda}}(c) \in B\}\,.$$

In unseren Anwendungen wird immer mindestens eine der beiden Mengen A und B einelementig sein. Deshalb definieren wir $\mathbf{Calls}_{A \to B}$ auch für einzelne Prozeduren. So gilt etwa $\mathbf{Calls}_{p \to q} = \mathbf{Calls}_{\{p\} \to \{q\}}$.

Für interprozedurale Analysen des Programms bietet sich die Darstellung der Prozedur-Aufrufe in Form eines Aufrufgraphen oder Call Graph an.

Definition 17 (Aufrufgraph)**.** Sei $A \in \mathbf{Progs}$ ein Programm. Dann ist der Aufrufgraph des Programms gegeben durch $G_A = (A, E)$ mit

$$E = \left\{(p, q) \in A^2 \mid \mathbf{Calls}_{p \to q} \neq \varnothing\right\}.$$

Außerdem muss für einen Aufruf festgestellt werden können, mit welchen konkreten Argumenten die formalen Parameter der aufgerufenen Prozedur belegt werden.

Definition 18 (Argumente eines Aufrufs)**.** Seien $p, q \in \mathbf{Procs}$, $c \in \mathbf{Calls}_{p \to q}$. Dann repräsentiert $arg_c^p \colon \mathbf{Params}_q \to \mathbf{Ops}$ die Belegung der Parameter von q mit Argumenten an Aufrufstelle c. Für $v \in \mathbf{Params}_q$ ist $arg_c^p(v)$ das Argument, welches dem Parameter v an Aufrufstelle c übergeben wird.

3 Entwurf und Implementierung

Im Folgenden werden wir die von uns verwendeten Algorithmen und besondere Herausforderungen bei ihrer Implementierung genauer betrachten. Wir klonen zielgetrieben – wie von Hall, Cooper und Kennedy in [11] und [1] beschrieben. Es wird also nicht jeder Aufruf spezialisiert, bei dem statisch bekannte Konstanten übergeben werden. Stattdessen versuchen wir nur dann zu klonen, wenn wir damit andere Optimierungen begünstigen können. Der Prozess gliedert sich in drei große Teilabschnitte.

Im ersten Schritt analysieren wir die Wichtigkeit der *formalen Parameter* aller im Programm vorhandenen Prozeduren. Hier wird – unabhängig von konkreten Prozedur-Aufrufen im Programm – untersucht welche Parameter besonders große Auswirkungen auf die Optimierbarkeit der zugehörigen Prozedur haben. Diese Parameter werden als *wichtig* vermerkt.

Im zweiten Schritt werden die tatsächlich im Programm vorhandenen Aufrufe untersucht. Werden hierbei Belegungen von wichtigen formalen Parametern mit konstanten Argumenten gefunden, wird an dieser Stelle die Möglichkeit der Erzeugung einer Spezialisierung der aufgerufenen Funktion vermerkt. Das Erstellen eines Klons kann durch die resultierende Verfeinerung der vorhandenen Information weitere Klonmöglichkeiten innerhalb des Klons offenbaren. Da hier in pathologischen Fällen exponentiell viele Klone entstehen können, ist die Beschränkung auf die wichtigen Parameter aus Schritt Eins von besonderer Bedeutung.

Der letzte Schritt besteht aus der Erstellung einer sinnvollen Auswahl von Prozedur-Spezialisierungen aus der Menge aller zuvor ermittelten Möglichkeiten. Einerseits ist übermäßiges Programmwachstum zu vermeiden. Andererseits stellen nicht alle Spezialisierungen gleich viel Performance-Steigerung in Aussicht. Deshalb untersuchen wir Bewertungsmöglichkeiten für Klone sowie die Bestimmung einer optimalen Auswahl, welche sich aber als schwierig herausstellen wird.

3.1 Identifikation wichtiger Parameter

Wie Eingangs bereits erwähnt, dient unsere erste Analyse der Einschränkung des Suchraums der folgenden Analysen. Seien hierzu $\mathbf{Ops}^*_\Lambda \subseteq \mathbf{Ops}_\Lambda$ die Knoten des Programms, denen wir besondere Bedeutung zumessen. In Schritt Zwei wollen wir Klon-Möglichkeiten, welche für diese Knoten keine Verbesserungen in Aussicht stellen, ignorieren und somit die Anzahl der zu untersuchenden Klon-Möglichkeiten reduzieren.

In unserer Implementierung besteht $\mathbf{Ops}^*_\Lambda$ aus den Operanden von bedingten Sprüngen, Zieladressen von Prozeduraufrufen und Zieladressen von Speicherzugriffen. Wir vermuten, dass durch zusätzliche Information an diesen Programmstellen viel zusätzliches Optimierungspotential entstehen kann. Im Gegensatz dazu lohnt es sich vermutlich nicht eine Prozedur zu spezialisieren, nur um eine Addition einzusparen.

3.1.1 Formale Grundlagen

Zunächst versuchen wir also herauszufinden, welche Parameter Einfluss auf die wichtigen Programmstellen $\mathbf{Ops}^*_\Lambda$ einer Prozedur $p \in \Lambda$ haben. Sei für alle $o \in \mathbf{Ops}_p$ zunächst

$$D^+_o := \{P \in \mathcal{P}(\mathbf{Params}_p) \mid P \subseteq \mathbf{Consts} \;\Rightarrow\; o \in \mathbf{Consts}\} \tag{3.1}$$

die Menge aller Parameterkombinationen, welche bei Belegung mit Konstanten den Wert von o bestimmen. Hierbei stellt $eval_p(o)$ die Auswertung von o unter Verwendung der transitiven Hülle von $opr_p(o)$ dar. Sei außerdem

$$D^*_o := \left\{P \in D^+_o \mid \nexists Q \subset P : Q \in D^+_o\right\} \tag{3.2}$$

die Menge aller *minimalen* Parameterkombinationen, welche den Wert von o bestimmen. Dann beschreibt

$$P^*_p := \bigcup_{o \in \mathbf{Ops}^*_p} D^*_o \tag{3.3}$$

alle Kombinationen von Parametern, welche den Wert wichtiger Programmstellen von p eindeutig bestimmen.

Allerdings sind diese Mengen recht unhandlich. Wenn $n = |\mathbf{Params}_p|$ die Anzahl der Parameter der untersuchten Funktion ist, so gibt es 2^{2^n} verschiedene Werte, die P^*_p annehmen kann. Bei vier Parametern ergeben sich bereits $2^{16} = 65536$ verschiedene Möglichkeiten.

Da wir hier nur eine grobe Vorauswahl der näher zu untersuchenden Aufrufkombinationen erreichen wollen, begnügen wir uns mit etwas ungenauerer Information. Betrachten wir im Folgenden also die jeweilige Vereinigung der Elemente von D_o^* und P_p^*:

$$D_o := \bigcup D_o^* \qquad\qquad P_p := \bigcup P_p^*. \tag{3.4}$$

D_o beschreibt also die Menge der Parameter welche Einfluss auf o haben können, P_p die Menge von Parametern, welche den Wert wichtiger Programmstellen von p bestimmen können.

Wir wollen sicherstellen, dass wir keine Möglichkeit zur Verbesserung der Information an wichtigen Programmstellen verpassen, wenn wir ausschließlich an Parametern in P_p klonen. Es soll also gelten, dass $\nexists X \in P_p^* : X \not\subseteq P_p$.

Mit der folgenden Äquivalenz

$$\begin{aligned} & \nexists X \in P_p^* : X \not\subseteq P_p \\ \Leftrightarrow\ & \forall X \in P_p^* : X \subseteq P_p \\ \Leftrightarrow\ & \bigcup P_p^* \subseteq P_p \end{aligned} \tag{3.5}$$

und der Definition von P_p in Gleichung (3.4) ist diese Eigenschaft nachgewiesen.

Wir wollen außerdem zeigen, dass wir, wie zuvor P_p^* aus D_o^*, auch P_p aus D_o berechnen können. Mit den Definitionen aus Gleichungen (3.3) und (3.4) sowie der Assoziativität und Kommutativität der Vereinigung von Mengen gilt:

$$P_p = \bigcup P_p^* = \bigcup\Big(\bigcup_{o \in \mathbf{Ops}_p^*} D_o^* \Big) = \bigcup_{o \in \mathbf{Ops}_p^*} \Big(\bigcup D_o^* \Big) = \bigcup_{o \in \mathbf{Ops}_p^*} D_o. \tag{3.6}$$

Nach dem Satz von Rice können wir jedoch im Allgemeinen auch diese einfacheren Mengen statisch nicht exakt berechnen. Wir müssen uns also mit einer Approximation zufrieden geben. Seien hierfür $\tilde{P}_p$ sowie $\tilde{D}_o$ die Approximationen für P_p und D_o.

Wollen wir weiterhin sicherstellen, dass wir auch bei der Verwendung von $\tilde{P}_p$ keine Möglichkeit zur Verbesserung der Information an wichtigen Programmstellen verpassen, muss analog zu Gleichung (3.5) gelten:

$$\bigcup P_p^* \subseteq P_p \subseteq \tilde{P}_p. \tag{3.7}$$

P_p muss also durch $\tilde{P}_p$ *überapproximiert* werden.

Analog zur Beziehung von P_p und D_o aus Gleichung (3.6) soll sich auch $\tilde{P}_p$ wieder aus $\tilde{D}_o$ berechnen lassen. Es muss also gelten:

$$\tilde{P}_p = \bigcup_{o \in \mathbf{Ops}_p^*} \tilde{D}_o. \tag{3.8}$$

Damit lässt sich Gleichung (3.7) darstellen als

$$P_p = \bigcup_{o \in \mathbf{Ops}_p^*} D_o \subseteq \bigcup_{o \in \mathbf{Ops}_p^*} \tilde{D}_o = \tilde{P}_p, \tag{3.9}$$

was genau dann der Fall ist, wenn auch $\tilde{D}_o$ eine Überapproximation von D_o ist:

$$D_o \subseteq \tilde{D}_o. \tag{3.10}$$

3.1.2 Algorithmen

Im Folgenden wollen wir einen konkreten Algorithmus vorstellen, um $\tilde{P}_p$ zu berechnen. Im Anschluss zeigen wir, wie man die bislang lokale Analyse leicht um interprozedurale Zusammenhänge erweitern kann.

Die hier vorgestellten Algorithmen entsprechen im Wesentlichen dem Vorgehen zur Berechnung von *CloningVars* nach Hall [11].

Es wurden auch Formulierungen unter Verwendung des Konzepts monotoner Frameworks erstellt. Diese wurden jedoch verworfen, da für deren Berechnung in libFirm kein Mechanismus existiert und außer der theoretischen Betrachtung wenig Mehrwert zu erkennen war.

Lokale Analyse

Algorithmus 1 berechnet $\tilde{D}_o$[1] für eine gegebene Prozedur p indem die Lösung des folgenden Gleichungssystems durch eine Fixpunktiteration ermittelt wird:

$$\tilde{D}_o = \begin{cases} \varnothing & \text{für } o \in \mathbf{Consts}, \\ \{o\} & \text{für } o \in \mathbf{Params}, \\ \bigcup\limits_{o' \in opr_p(o)} \tilde{D}_{o'} & \text{sonst.} \end{cases} \qquad \text{für alle } o \in \mathbf{Ops}_p. \tag{3.11}$$

Da wir über alle Operanden vereinigen, können wir keinen Ausführungspfad verpassen. Somit muss Gleichung (3.10) gelten.

[1] Die Berechnung ist beschränkt auf Datenflussabhängigkeiten. Die Abhängigkeiten des Werts eines Knotens vom Kontrollfluss geht nicht mit in die Berechnung ein. Beispielsweise hängt der Wert von ϕ in Abbildung 3.1b über den Kontrollfluss auch von `x` ab. Da wir aber am Ende an $\tilde{P}_p$ interessiert sind, und hier ohnehin der Wert von $\tilde{D}_o$ für alle den Kontrollfluss beeinflussenden Knoten eingeht, macht diese Vereinfachung in unserem Fall keinen Unterschied. Außerdem sind die Kontrollflussabhängigkeiten in Firm, anders als in unserem vereinfachten Graphen, nicht direkt abgebildet. Statt dessen müssen sie erst über eine gesonderte Analyse in Erfahrung gebracht werden.

Algorithm 1 Berechnung von Parameter-Abhängigkeiten

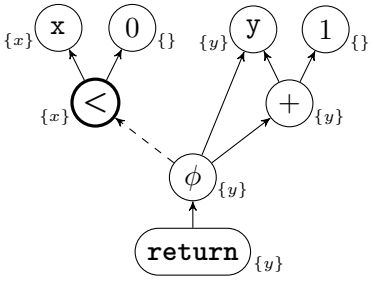
1: **procedure** PARAMDEPS(p)
2: $W \leftarrow usr_p(\mathbf{Params}_p)$ ▷ Set of nodes yet to process
3: $V \leftarrow \mathbf{Params}_p \cup \mathbf{Consts}_p$ ▷ Visited nodes
4: $\forall o \in \mathbf{Ops}_p : \tilde{D}_o \leftarrow \begin{cases} \{o\} & \text{if } o \in \mathbf{Params}_p \\ \varnothing & \text{otherwise} \end{cases}$
5: **while** $\exists o \in W$ **do**
6: $W \leftarrow W \setminus \{o\}$
7: **if** $o \in \Phi \vee opr_p(o) \subseteq V$ **then**
8: $V \leftarrow V \cup \{o\}$
9: $\tilde{D}_o \leftarrow \bigcup_{o' \in opr_p(o)} \tilde{D}_{o'}$
10: **if** $\tilde{D}_o$ did change **then**
11: $W \leftarrow W \cup usr_p(o)$
12: **return** $\tilde{D}$ ▷ return the mapping for all $o \in \mathbf{Ops}_p$

Nach der Berechnung von $\tilde{D}_o$ durch Algorithmus 1 kann $\tilde{P}_p$ einfach gemäß Gleichung (3.8) berechnet werden. Natürlich ist es auch möglich dies direkt während der Berechnung von $\tilde{D}_o$ in einem Durchlauf durchzuführen. Der ursprüngliche Algorithmus von Hall sowie unsere Implementierung tut genau dies. Hier wurde jedoch zur klareren Hervorhebung der einzelnen Analysen die getrennte Darstellung gewählt.

Abbildung 3.1 zeigt beispielhaft die Analyse einer einfachen Prozedur.

```
1 def f(x, y):
2   if x < 0:
3     return y
4   else:
5     return y + 1
```

(a) Beispiel-Prozedur p für die Berechnung von $\tilde{P}_p$



(b) Vereinfachter Graph von p. Die gestrichelte Kante deutet eine Kontrollflussabhängigkeit an. Knoten mit fetter Umrandung sind wichtig.

Abbildung 3.1: Beispiel-Berechnung von $\tilde{P}_p$ für eine einfache Prozedur. Im Graph sind neben den Knoten die zugehörigen Werte von $\tilde{D}_o$ nach Abschluss der Analyse zu sehen. Der endgültige Wert von $\tilde{P}_p$ ist $\{y\}$, da hier nur der Knoten $<$ zu den wichtigen Ausdrücken gehört.

Interprozedurale Verfeinerung

Um die Genauigkeit zu erhöhen werden die Ergebnisse der Analyse anschließend noch interprozedural propagiert. Das heißt, dass Argumente für wichtige Prozedur-Parameter nun ebenfalls Teil der wichtigen Ausdrücke werden. Dadurch kann die Menge der wichtigen Parameter der aufrufenden Prozedur weiter wachsen. Diese Propagation geschieht, wie in Algorithmus 2 gezeigt, in einem einzelnen Durchgang durch den Call-Graph des Programms.

Algorithm 2 Interprozedurale Berechnung wichtiger Parameter

```
1: procedure ImportantParams(A)                          ▷ A ∈ Progs
2:     for all p ∈ G_A in postorder do
3:         D̃ ← ParamDeps(p)
4:         P̃_p ← ⋃_{o∈Ops*_p} D̃_o                        ▷ gemäß Gleichung (3.8)
5:         for all c ∈ Calls_{p→A} do
6:             for all v ∈ P̃_{cle_A(c)} do
7:                 P̃_p ← P̃_p ∪ D̃_{arg^p_c(v)}
8:     return P̃
```

Ist der Call Graph azyklisch, entspricht die hier verwendete postorder einer umgekehrt topologischen Ordnung. In diesem Fall sind die interprozedural wichtigen Parameter einer jeden Prozedur bereits vollständig berechnet, wenn sie von den Aufrufern der Prozedur in Zeile 6 genutzt werden.

Ist eine Prozedur Teil eines rekursiven Zyklus, so kann es vorkommen, dass nicht alle wichtigen Parameter gefunden werden. Dies kann aber nur vorkommen, wenn die Belegung der Parameter innerhalb des Zyklus permutiert wird. Ein solcher Fall ist in Abbildung 3.2 dargestellt.

```
1 def f(x, y, z):
2   if x:
3     ...
4   f(y, z, x)
```

Abbildung 3.2: Beispiel für eine rekursive Prozedur, bei der nicht alle wichtigen Parameter gefunden werden. `x` ist schon nach der lokalen Analyse als wichtig markiert, `y` kommt in der interprozeduralen Verfeinerung hinzu. `z` wird hingegen nicht als wichtig erkannt.

In diesem Spezialfall wurde ein Verlust an Genauigkeit in Kauf genommen, da wir ihn nicht als besonders praxisrelevant empfinden. Das Problem ließe sich aber leicht mit einer Fixpunktiteration lösen. Die oben bereits erwähnte Formulierung als Datenflussproblem für monotone Frameworks etwa, würde die gesuchten Mengen auch für rekursive Prozeduren korrekt berechnen.

3.2 Ermitteln von Klon-Möglichkeiten

Wir wissen nun für welche Parameter zusätzliche Information besonders zuträglich für weitere Optimierungen wäre. Im nächsten Schritt wollen wir herausfinden, welche dieser Parameter in unserem Eingabeprogramm tatsächlich mit konstanten Argumenten belegt werden.

3.2.1 Cloning Vectors

Wir werden dazu das Konzept eines *Cloning Vectors* benötigen. Ein Cloning Vector ist eine Teilbelegung der Parameter einer Prozedur mit Konstanten.

Definition 19 (Cloning Vectors)**.** Sei $q \in \mathbf{Procs}$. Dann ist die Menge aller möglichen Cloning Vectors von q gegeben durch

$$\mathbf{CV}_q := \{v \colon Q \to \mathbf{Consts} \mid Q \subseteq \mathbf{Params}_q\}\,.$$

Jedes Element von $\mathbf{CV}_q$ lässt sich direkt auf eine Spezialisierung von q abbilden, wie Algorithmus 3 skizzenhaft zeigt. Da die konkrete Implementierung naturgemäß sehr stark von dem Compiler abhängt in dem die Optimierung umgesetzt ist, wird hier nicht genauer darauf eingegangen.

Algorithm 3 Klonen von Prozeduren

```
1: procedure CLONE(p, v)
2:     p' ← clone of p
3:     for all (x, y) ∈ v do
4:         Replace all occurrences of x in G_p with y
5:     Simplify G_p' with the new information (using constant propagation etc.)
```

Für uns sind jedoch nur die Cloning Vectors von Interesse, die auch von Prozeduraufrufen im zu optimierenden Programm erzeugt werden.

Definition 20 (Induzierter Cloning Vector)**.** Seien $p, q \in \mathbf{Procs}$, $c \in \mathbf{Calls}_{p \to q}$ und $Q \subseteq \mathbf{Params}_q$. Dann ist der von c induzierte und auf Q beschränkte Cloning Vector $\nu(p, c, Q) \in \mathbf{CV}_q$ gegeben durch

$$\nu(p, c, Q) := arg_c^p \cap (Q \times \mathbf{Consts}).$$

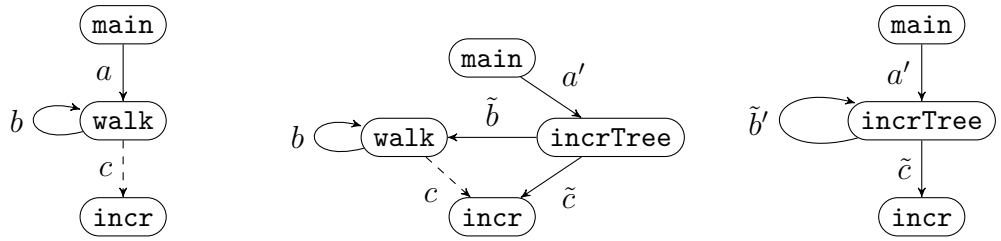
Q beschreibt dabei die Menge der zu betrachtenden Parameter.

Für uns wird im Folgenden immer $Q = \tilde{P}_q$ gelten. Jedes Element von $\nu(p, c, \tilde{P}_q)$ entspricht also einem wichtigen Parameter von q, welcher in c mit einem konstanten Argument belegt wird.

3.2.2 Der Algorithmus

Die Herausforderung beim Erzeugen der Klone für das gesamte Programm liegt in der Abhängigkeit der Klonvektoren untereinander. Schließlich kann das Klonen einer Prozedur p dazu führen, dass Argumente für Funktionsaufrufe aus p konstant werden. Demnach wäre es optimal zuerst alle Aufrufer einer Prozedur zu klonen, bevor sie selbst als Klonkandidat in Betracht gezogen wird.

Deshalb untersuchen wir die Klonmöglichkeiten im Call Graph in reverse postorder. Somit werden in Programmen ohne Rekursion alle Klonmöglichkeiten identifiziert. Auch direkte Rekursion stellt für den Algorithmus kein Hindernis dar. Ein Beispiel hierfür ist in Abbildung 3.3 gegeben.



(a) Der Graph des Eingabeprogramms.

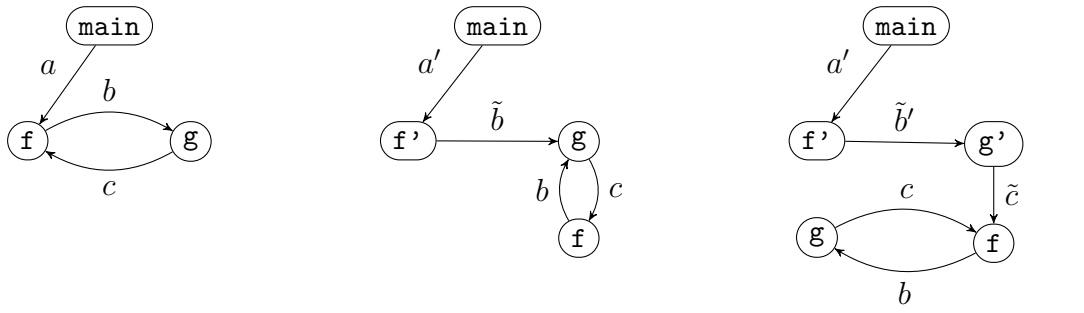
(b) `incrTree` wurde als Klon von `walk` erstellt.

(c) Der Aufruf von `incrTree` nach `walk` wurde durch eine direkte Rekursion ersetzt.

Abbildung 3.3: Anwendung von Procedure Cloning auf das Programm aus Abbildung 1.1. Die gestrichelten Kanten repräsentieren Aufrufe, bei denen das Ziel über eine Variable bestimmt wird. Die verwendete Abarbeitungsreihenfolge gemäß Algorithmus 4 ist $(\texttt{main}, \texttt{walk}, \texttt{incr})$ für die Prozeduren und $(b, \underline{a}, \underline{\tilde{b}}, \tilde{c})$ für die Aufrufe. Die Aufrufe, deren Ziel auf eine passende Spezialisierung geändert wurde, sind unterstrichen.

Indirekte Rekursionszyklen werden hingegen nur unvollständig geklont. Wie bereits zuvor erwähnt, messen wir indirekter Rekursion keine große Bedeutung bei. Demnach wird dieses Verhalten als akzeptabel angesehen. Ein beispielhafte Anwendung von Algorithmus 4 auf ein Eingabeprogramm mit indirekter Rekursion ist in Abbildung 3.4 dargestellt.

Algorithmus 4 beschreibt das von uns verwendete Verfahren im Detail. Das Ergebnis



(a) Eingabeprogramm **(b)** Klon von `f` wurde erstellt. **(c)** Klon von `g` wurde erstellt.

Abbildung 3.4: Anwendung von Procedure Cloning auf ein Programm mit indirekter Rekursion. Die verwendete Abarbeitungsreihenfolge gemäß Algorithmus 4 ist $(\texttt{main}, \texttt{f}, \texttt{g})$ für die Prozeduren und $(\underline{a}, c, b, \underline{\tilde{b}})$ für die Aufrufe. Der in (c) neu hinzu gekommene Aufruf $\tilde{c}$, welcher durch das Klonen von `g` entsteht, wird nicht mehr untersucht, da zu diesem Zeitpunkt `f` bereits abgearbeitet wurde. Die wünschenswerte Änderung seines Ziels auf `f'` entfällt also.

ist ein maximal geklontes Programm, sowie eine Zuordnung von Klonen zu ihren Originalen.

Beim Erzeugen der Klone ist in pathologischen Fällen theoretisch exponentielles Programmwachstum möglich. Eine genauere Betrachtung hierzu findet sich bei Cooper [1]. Wir vermuten jedoch, dass dies in der Praxis kein Problem darstellt, da die Anzahl der möglichen Klone schon durch die Vorauswahl stark eingeschränkt wird. Eine einfache, wenn auch nicht elegante, Methode dies zu verhindern wäre etwa eine Begrenzung der maximal zulässigen Klonanzahl.

Ein in der Praxis weitaus relevanteres Problem stellt das Ausrollen von Rekursion dar. In der vorliegenden Version kann es vorkommen, dass alle Aufrufkonfigurationen einer direkt rekursiven Funktion spezialisiert werden. Ein Beispiel hierfür findet sich in Abbildung 3.5.

Eine krude Möglichkeit die Auswirkungen dieses Problems einzugrenzen, wäre eine Beschränkung der maximalen Klone pro Prozedur. Eleganter[2] wäre es hingegen, genau so lange innerhalb eines Zyklus zu klonen, wie sich die erzeugten Klone im Vergleich zum jeweils vorherigen verbessern – eine Art Fixpunktiteration also. Wie genau die Qualität eines Klons gemessen werden kann, behandeln wir in Abschnitt 3.3.

[2]Wenn später, wie in Abschnitt 3.3 beschrieben, eine optimale Auswahl von Klonen getroffen wird, würden diese nutzlosen Klone natürlich auch wieder verschwinden. Tatsächlich könnte man durch eine Heuristik zum Vermeiden derartiger Klone auch ertragreiche Spezialisierungen am Ende einer solchen Kette verpassen. Hier wäre also eine gründlichere Untersuchung dieses Aspekts

Algorithm 4 Ermitteln von Klon-Möglichkeiten

procedure CLONES(A) ▷ $A \in$ **Progs**
  $\hat{A} \leftarrow A$
  $\tilde{P} \leftarrow$ IMPORTANTPARAMS(A)
  $\forall p \in A, v \in \mathbf{CV}_p : C_{p,v} \leftarrow \bot$ ▷ Maps p and v to the corresponding clone
  $\forall p \in A : orig(p) \leftarrow p$ ▷ Maps clones to their originals

  for all $p \in G_A$ in reverse postorder **do**
    for all $c \in \mathbf{Calls}_{\hat{A} \to p}$ **do** ▷ $clr_{\hat{A}}(c)$ might be a clone
      $v \leftarrow \nu(clr_{\hat{A}}(c), c, \tilde{P}_p)$
      next if $v = \varnothing$ ▷ no constant args, no cloning possibility

      if $C_{p,v} = \bot$ **then** ▷ we created no matching clone yet
        $C_{p,v} \leftarrow$ CLONE(p, v)
        $orig(C_{p,v}) \leftarrow p$
      Update c so that $cle_{\hat{A}}(c) = C_{p,v}$ ▷ Update call target
  return $(\hat{A}, orig)$ ▷ The input program with maximal cloning applied

3.2.3 Vergleich zu Procedure Cloning nach Cooper

Algorithmus 4 sowie das Konzept der Cloning Vectors ist inspiriert durch die Arbeit von Cooper et al. [1]. Unser Vorgehen unterscheidet sich insofern, dass wir jeden gefundenen Cloning Vector sofort in einen Klon umsetzen. Cooper et al. sammeln erst alle Cloning Vectors und kombinieren dann jene, die eine identische Auswirkung auf die Optimierung der wichtigen Programmstellen haben. Dieses Vorgehen erfordert jedoch diverse Analysen welche in libFirm so nicht vorhanden waren. Sie umzusetzen war im Rahmen dieser Arbeit nicht möglich.

Das Zusammenführen von Cloning Vectors ist bei Cooper auch der zentrale Mechanismus zur Kontrolle des Programmwachstums. Dahinter steht die Vermutung, dass sich durch die Vorauswahl und das spätere Zusammenführen die Anzahl der Klone so sehr reduziert, dass in den meisten Fällen keine Auswahl aus der verbleibenden Menge mehr nötig ist. Dementsprechend wird das Thema einer guten Auswahl bei zu vielen Klonen nur grob angerissen.

Unser Vorgehen verhindert diese Art der Verschmelzung, da wir die Knoten des ursprünglichen und des spezialisierten Graphen einer Prozedur nicht ohne Weiteres in Bezug zueinander setzen können. Andererseits haben wir durch das sofortige Klonen auch eine bessere Ausgangssituation zur Bewertung einzelner Klone. Deshalb werden wir nun die Möglichkeiten einer guten Klonauswahl genauer untersuchen.

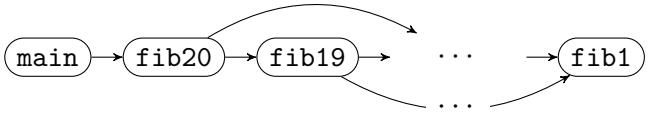
sinnvoll.

```
1 def main():
2   return fib(20)
3
4 def fib(n):
5   if (n <= 2):
6     return 1
7   else
8     return (fib(n-1)
9           + fib(n-2))
```

(a) Ausgangsprogramm

```
1 def main():
2   return fib20()
3
4 def fib20(n):
5   return (fib19()
6         + fib18())
7 # ...
8 def fib1(n):
9   return 1
```

(b) Ausgerollte Rekursion nach dem Klonen



(c) Graph des geklonten Programms

Abbildung 3.5: Beispiel für das unerwünschte Ausrollen einer direkten Rekursion.

3.3 Auswahl von Klonen

Durch das Erstellen von Klonen erwarten wir uns bessere Programmlaufzeiten. Das Spezialisieren einer Prozedur verursacht aber auch Kosten. Mit jeder Spezialisierung einer Prozedur steigt zum Beispiel die Größe des Ausgabeprogramms um die Ausgabegröße des Klons. Dies kann weitere negative Effekte nach sich ziehen, wie etwa reduzierte Code-Cache-Effizienz [12]. Deshalb ist es im Allgemeinen nicht sinnvoll alle Klon-Kandidaten tatsächlich zu erzeugen. Es bedarf also einer Strategie zur Auswahl der tatsächlich zu erstellenden Klone aus der Menge aller möglichen Klone.

3.3.1 Grundlegende Definitionen

Im Folgenden gehen wir davon aus, dass wir zusätzlich zu unserem Eingabeprogramm Λ, das Ergebnis $(\Lambda^+, \mathit{orig})$ des Aufrufs Clones(Λ) zur Verfügung haben. Dabei stellt Λ^+ das maximal geklonte Ausgangsprogramm dar, $\mathit{orig}\colon \mathbf{Procs} \to \mathbf{Procs}$ eine Funktion welche einen Klon auf sein Original abbildet.

Definieren wir vorerst abstrakt $\varrho(p)$ als den Ertrag und $\widetilde{\gamma}(p)$ als die Kosten für das Aufnehmen eines Klons p in das Programm. Des Weiteren soll $\pi(p)$ den Gewinn durch das Aufnehmen von p darstellen. Hierfür müssen Kosten und Ertrag in der selben Einheit vorliegen. Da dies im Allgemeinen nicht der Fall ist, benötigen wir

den Umrechnungsfaktor α von Kosten in Ertrag. Mit $\gamma(p)$ als umgerechnete Kosten können wir schließlich formal definieren:

$$\begin{aligned}
\varrho &\colon \mathbf{Procs} \to \mathbb{R}_+ \\
\widetilde{\gamma} &\colon \mathbf{Procs} \to \mathbb{R}_+ \\
\gamma &\colon \mathbf{Procs} \to \mathbb{R}_+, \quad p \mapsto \alpha\widetilde{\gamma}(p) \\
\pi &\colon \mathbf{Procs} \to \mathbb{R}, \quad\ p \mapsto \varrho(p) - \gamma(p).
\end{aligned}$$

Außerdem benötigen wir noch die entsprechenden Kennzahlen für ein ganzes Programm. In den meisten Fällen ergeben sich diese ganz natürlich. Im Fall des Gesamtertrags $\widehat{\varrho}$ müssen wir jedoch etwas genauer hinsehen. Wie oben bereits erwähnt, entspricht unser Ertrag positiven Laufzeiteigenschaften einer Spezialisierung. Diese kommen genau dann zum Tragen wenn die entsprechende Prozedur aufgerufen wird. Wir benötigen also einen Schätzer für die Anzahl der Aufrufe einer Prozedur.

Definition 21 (Aufrufhäufigkeit einer Prozedur)**.** Sei $A \in \mathbf{Progs}$ ein Programm. Dann ist $f_A : \mathbf{Procs} \to \mathbb{R}_{\geq 0}$ ein Schätzer für die erwartete Anzahl der Aufrufe einer Prozedur während eines Programmdurchlaufs von A.

Damit ergibt sich

$$\begin{aligned}
\widehat{\varrho} &\colon \mathbf{Progs} \to \mathbb{R}_+, \quad A \mapsto \sum_{p \in A} f_A(p)\varrho(p) \\
\widehat{\gamma} &\colon \mathbf{Progs} \to \mathbb{R}_+, \quad A \mapsto \sum_{p \in A} \gamma(p) \\
\widehat{\pi} &\colon \mathbf{Progs} \to \mathbb{R}, \quad\ A \mapsto \widehat{\varrho}(A) - \widehat{\gamma}(A).
\end{aligned}$$

3.3.2 Die Qualität eines Klons

Da wir bereits für die Auswahl der Klonkandidaten eine Charakterisierung wichtiger Programmstellen verwendet haben, bietet es sich an diese auch zur Feststellung der Güte eines Klons zu verwenden. Die von uns betrachteten, wichtigen Programmstellen repräsentieren besondere Komplexität im Programmablauf. Diese soll im Endresultat natürlich möglichst gering gehalten werden.

Eine zentrale Qualitätseigenschaft eines Klons p ist also seine Komplexität $\mathit{cmplx}(p)$. Die Reduktion der Komplexität im Vergleich zum im ursprünglichen Programm vorhandenen Original $\mathit{orig}(p)$ nennen wir

$$\mathit{cmplx}_\Delta(p) := \mathit{cmplx}(\mathit{orig}(p)) - \mathit{cmplx}(p).$$

Um $cmplx(p)$ konkret definieren zu können, benötigen wir außerdem noch verwandte Funktionen auf der Menge der Operationen. Sei hierzu $o \in \mathbf{Ops}_p$. Dann ist $cmplx'(o)$ die Komplexität von o und $f'_p(o)$ ein Schätzer für die mittlere Anzahl der Auswertungen von o während *eines* Durchlaufs von p.

$$cmplx(p) := \sum_{o \in \mathbf{Ops}^*_p} cmplx'(o) \cdot f'_p(o)$$

Wir definieren im Rahmen dieser Arbeit $cmplx' \equiv 1$. Es wäre aber durchaus denkbar hier weiter zu differenzieren, sodass etwa ein Speicherzugriff mehr Komplexität verursacht als ein bedingter Sprung. Dies soll jedoch nicht Gegenstand dieser Arbeit sein.

Wollen wir nun den Ertrag eines Klons durch die erzielte Komplexitätsreduktion und seine Kosten durch die Programmgröße definieren, so erreichen wir das durch folgende Definitionen:

$$\varrho(p) := cmplx_\Delta(p)$$
$$\widetilde{\gamma}(p) := \left|\mathbf{Ops}_p\right|.$$

Die Größenmessung könnte ebenfalls noch verfeinert werden. Mit Kenntnis der Zielarchitektur wäre es etwa möglich die Anzahl der zu erzeugenden Maschinenbefehle für die einzelnen Operationen mit einzubeziehen.

Die Wahl von α in $\gamma(p) = \alpha\widetilde{\gamma}(p)$ werden wir als Parameter der Optimierung offen lassen. Somit kann festgelegt werden wie viel Programmwachstum in Kauf genommen werden soll, um die Laufzeit zu verbessern.

3.3.3 Eine optimale Auswahl

Die wohl größte Schwierigkeit bei der Auswahl der Klone stellt ihre Abhängigkeit voneinander dar. Viele Klone können nur zum Einsatz kommen, wenn ihre Aufrufer bereits spezialisiert wurden und somit die interprozeduralen Fakten schärfer sind. Ein Klon kann also also nur Ertrag erwirtschaften, wenn mindestens einer seiner Aufrufer auch im Programm vorhanden ist.

Wir werden im Folgenden annehmen, dass alle im ursprünglichen Programm Λ vorhandenen Prozeduren nicht entfernt werden dürfen. Dies trifft in Wirklichkeit nur auf Programmeinstiegspunkte zu, vereinfacht im Weiteren aber die Optimierung, die sich auch mit dieser Einschränkung noch als komplex genug herausstellen wird.

Mit obiger Einschränkung können wir den Call-Graph G_{Λ^+} vereinfachen, indem wir alle Knoten aus Λ zu einem Knoten v_Λ verschmelzen. Alle aus der Partition Λ

ausgehenden Kanten gehen nun von v_Λ aus, alle in Λ eintretenden Kanten werden verworfen, da sie zur Optimierung nicht relevant sind, wie sich im Folgenden zeigen wird. Wir nennen diesen Graph $\tilde{G}_{\Lambda^+}$. Wie man leicht erkennt, ist v_Λ eine Wurzel von $\tilde{G}_{\Lambda^+}$. Wir definieren $\varrho(v_\Lambda) = \gamma(v_\Lambda) = 0$ und $orig(v_\Lambda) = v_\Lambda$.

Wir wollen nun eine Definition von f angeben, die dafür sorgt, dass ein Klon genau dann in $\widehat{\varrho}$ gewertet wird, wenn er von v_A aus erreichbar ist. Denn dies ist eine notwendige Bedingung dafür, dass er im erzeugten Programm überhaupt aufgerufen werden kann. Die folgende Definition erfüllt diese Forderung:

$$f_{A^+}(p) := \begin{cases} 1 & \text{wenn } \left\langle \tilde{G}_{A^+}, v_A, p \right\rangle \in \textsc{Path}, \\ 0 & \text{sonst.} \end{cases}$$

Unter Verwendung dieser Definition von f kann eine – unter den obigen Annahmen und Vereinfachungen – optimale Klonauswahl durch Maximierung von $\widehat{\pi}$ erreicht werden.

Da wir die Komplexität dieses Problems untersuchen wollen, formalisieren wir es als Optimierungsproblem gemäß Definition 16. Die Maximierung von $\widehat{\pi}$ ähnelt in ihrer Struktur einem Steinerbaumproblem mit Knotengewichten auf einem gerichteten Graphen. Um diese Ähnlichkeit mehr herauszustellen und das Problem besser zugänglich zu machen, werden wir das Problem etwas umformulieren.

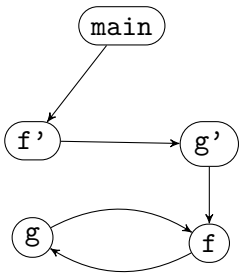
Es ergibt sich das Optimierungsproblem $\textsc{Opt-CS} = (I, \mathcal{M}, w_I, \min)$, wobei

$$\begin{aligned} I &= \{\langle G, r, \breve{\gamma}, \breve{\varrho}\rangle \mid \langle G, r\rangle \in \textsc{Root} \wedge \breve{\gamma}, \breve{\varrho} \colon V(G) \to \mathbb{R}_{\geq 0}\} \\ \mathcal{M}(\langle G, r, \breve{\gamma}, \breve{\varrho}\rangle) &= \{V' \subseteq V(G) \mid \langle G[V'], r\rangle \in \textsc{Root}\} \\ w_x(y) &= \sum_{v \in y} \breve{\gamma}(v) + \sum_{v \in V \setminus y} \breve{\varrho}(v) \quad \text{für jedes } x = \langle (V, E), r, \breve{\gamma}, \breve{\varrho}\rangle \in I. \end{aligned}$$

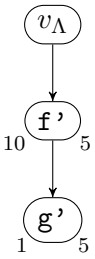
Gegeben ist ein Graph G mit Wurzel r, sowie je eine Kosten- und eine Ertragsfunktion auf den Knoten des Graphen. Gesucht ist eine Teilmenge V' der Knoten von G, so dass der von ihnen induzierte Teilgraph von G wiederum einen Graph mit Wurzel r bildet. Die zu minimierende Zielfunktion ist die Summe der Kosten der gewählten Knoten und des Ertrags der nicht gewählten Knoten.

Die Instanz von Opt-CS zur Bestimmung einer optimalen Klonauswahl hinsichtlich unserer bisherigen Kriterien ist dann gegeben durch $x = \left\langle \tilde{G}_{\Lambda^+}, v_\Lambda, \gamma, \varrho \right\rangle$. Ein Beispiel findet sich in Abbildung 3.6.

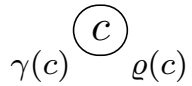
Es mag nicht sofort ersichtlich sein, dass die Minimierung von w_x zur selben Lösung führt wie die Maximierung von $\widehat{\pi}$. Betrachten wir deshalb die folgenden Zusammenhänge.



(a) G_{Λ^+}: Das geklonte Programm aus Abbildung 3.4.



(b) Die Instanz von OPT-CS: $\tilde{G}_{\Lambda^+}$ mit γ und ϱ.



(c) Notation der Knoten mit Kosten und Ertrag.

Abbildung 3.6: Illustration von OPT-CS. Das Optimum für die gegebene Instanz ist die leere Menge.

Seien $y \in \mathcal{M}(x)$, $\tilde{V}_{\Lambda^+} = V(\tilde{G}_{\Lambda^+})$ und $\widehat{\varrho}_{\max} = \sum_{p \in V(\tilde{G}_{\Lambda^+})} \varrho(p)$. Wegen $\tilde{G}_{\Lambda^+}[y] = G_y = \tilde{G}_y$ und $\langle G_y, v_\Lambda \rangle \in \text{ROOT}$ gilt folgende Gleichheit[3]:

$$\widehat{\varrho}(y) = \sum_{p \in y} f_y(p)\varrho(p) = \sum_{p \in y} \varrho(p).$$

Damit folgt dann

$$\begin{aligned} w_x(y) &= \sum_{p \in y} \gamma(p) + \sum_{p \in \tilde{V}_{\Lambda^+} \setminus y} \varrho(p) \\ &= \widehat{\gamma}(y) \quad + \widehat{\varrho}_{\max} - \widehat{\varrho}(y) \\ \Leftrightarrow \; -w_x(y) &= \widehat{\varrho}(y) - \widehat{\gamma}(y) - \widehat{\varrho}_{\max} \\ &= \widehat{\pi}(y) \qquad\quad - \widehat{\varrho}_{\max} \end{aligned}$$

Da für beliebige $f : \mathbb{R} \to \mathbb{R}$ und endliche, nichtleere Mengen $M \subseteq \mathbb{R}$

$$\min\{f(x) \mid x \in M\} = -\max\{-f(x) \mid x \in M\}$$

gilt, folgt schließlich

$$\begin{aligned} \min\{w_x(y) \mid y \in \mathcal{M}(x)\} &= -\max\{-w_x(y) \mid y \in \mathcal{M}(x)\} \\ &= -\max\{\widehat{\pi}(y) - \widehat{\varrho}_{\max} \mid y \in \mathcal{M}(x)\} \\ &= -\max\{\widehat{\pi}(y) \mid y \in \mathcal{M}(x)\} - \widehat{\varrho}_{\max}. \end{aligned}$$

Somit haben die Optimierungsprobleme $\max \widehat{\pi}(y)$ und $\min w_x(y)$ die selben optimalen Lösungen.

[3]Wird v_Λ als Prozedur verwendet, hat diese genau einen Aufruf zu jedem Nachfolger von v_Λ in $\tilde{G}_{\Lambda^+}$.

3.3.4 Eine optimale Klonauswahl ist NP-schwer

Im Folgenden werden wir zeigen, dass es sich bei Opt-CS um ein NP-schweres Problem handelt. Da die Klassen P und NP nur für Entscheidungsprobleme definiert sind, führen wir zunächst das zu Opt-CS zugehörige Entscheidungsproblem ein.

Definition 22 (Entscheidungsproblem zu CS)**.** Sei Opt-CS $= (I, \mathcal{M}, w_I, g)$. Dann ist das zugehörige Entscheidungsproblem gegeben durch

$$\text{CS} := \{\langle x, k\rangle \in I \times \mathbb{R}_+ \mid \mathit{Opt}_{\text{Opt-CS}}(x) \leq k\}\,.$$

Man erkennt leicht, dass man mit einem Algorithmus für Opt-CS auch CS entscheiden kann. CS ist also nicht schwerer als Opt-CS. Können wir also zeigen, dass CS NP-schwer ist, so folgt dass auch Opt-CS NP-schwer ist.

Wir benötigen also ein bekanntes NP-vollständiges Problem, welches wir auf CS reduzieren können. Wir verwenden hierzu das Problem der Mengenüberdeckung.

Definition 23 (Set Cover Problem)**.** Die Optimierungsvariante des Mengenüberdeckungsproblems ist gegeben durch Opt-Set-Cover $= (I, \mathcal{M}, w_I, \min)$ mit

$$\begin{aligned} I &= \left\{\langle \mathcal{U}, \mathcal{S}\rangle \mid \mathcal{U} \text{ ist endliche Menge, } \mathcal{S} \subseteq \mathcal{P}(\mathcal{U}),\ \bigcup \mathcal{S} = \mathcal{U}\right\} \\ \mathcal{M}(\langle \mathcal{U}, \mathcal{S}\rangle) &= \left\{\mathcal{S}^* \subseteq \mathcal{S} \mid \bigcup \mathcal{S}^* = \mathcal{U}\right\} \\ w_x(y) &= |y| \quad \text{für jedes } x \in I. \end{aligned}$$

Das zugehörige Entscheidungsproblem ergibt sich dann wie zuvor durch

$$\text{Set-Cover} := \{\langle x, k\rangle \in I \times \mathbb{N}_0 \mid \mathit{Opt}_{\text{Opt-Set-Cover}}(x) \leq k\}\,.$$

Theorem 1. CS *ist NP-schwer.*

Beweis. Wir geben eine in polynomieller Zeit berechenbare Funktion $f\colon \Sigma^*_{bin} \to \Sigma^*_{bin}$

an, für die gilt $x \in$ Set-Cover $\Leftrightarrow$ $f(x) \in$ CS:

$$
\begin{aligned}
f(\langle \mathcal{U}, \mathcal{S}, k \rangle) &= \langle (V, E), r, \breve{\gamma}, \breve{\varrho}, k \rangle \\
\text{mit } V &= \{r\} \cup \varphi(\mathcal{S}) \cup \psi(\mathcal{U}), \\
E &= \{r\} \times \varphi(\mathcal{S}) \cup \left\{ \left(\varphi(S), \psi(u) \right) \mid u \in S \in \mathcal{S} \right\}, \\
\breve{\gamma}(v) &= \begin{cases} 1 & \text{falls } v \in \varphi(\mathcal{S}), \\ 0 & \text{sonst}, \end{cases} \\
\breve{\varrho}(v) &= \begin{cases} 2 & \text{falls } v \in \psi(\mathcal{U}), \\ 0 & \text{sonst}, \end{cases} \\
\text{wobei } \varphi &\colon \mathcal{S} \to V \text{ und } \psi \colon \mathcal{U} \to V \text{ bijektiv.}
\end{aligned}
$$

Es gilt also Set-Cover $\preceq_p$ CS, und da Set-Cover $\in$ NPC folgt die Behauptung. □

Abbildung 3.7 zeigt ein Beispiel für die Modellierung einer Instanz von Set-Cover als Instanz von CS

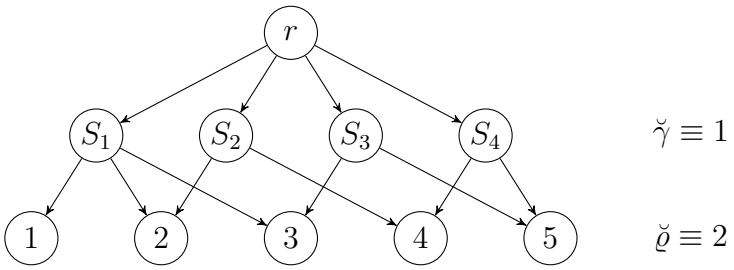


Abbildung 3.7: $f(\langle \mathcal{U}, \mathcal{S}, k \rangle)$ für $\mathcal{U} = \{1, 2, 3, 4, 5\}$ und $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$ mit $S_1 = \{1, 2, 3\}$, $S_2 = \{2, 4\}$, $S_3 = \{3, 5\}$ und $S_4 = \{4, 5\}$. Rechts sind Kosten und Ertrag der Knoten in der jeweiligen Ebene angegeben, soweit diese ungleich Null sind. Die optimale Lösung ist $\mathcal{S}^* = \{S_1, S_4\}$ mit $w(\mathcal{S}^*) = 2$.

3.3.5 Alternative Optimierungsansätze

Sei $T \in \mathbb{R}_+$. Dann ist eine andere Möglichkeit der Optimierung durch

$$
\begin{aligned}
\max \quad & \widehat{\varrho}(C) \\
\text{s.t.} \quad & \widehat{\gamma}(C) \leq T
\end{aligned}
$$

gegeben. Allerdings ist hier unklar wie T gewählt werden soll. Die Modellierung ist weniger natürlich als die vorhergehende. Außerdem ist diese Variante eine Verallgemeinerung des Rucksackproblems und damit auch NP-schwer. Konkret handelt

es sich um das Rucksackproblem mit Berücksichtigung von Abhängigkeiten welche durch einen Graphen, hier den Call-Graph, gegeben sind. Dieses Problem wird in [13] ausführlicher behandelt.

Eine einfache Heuristik auf Basis von CS ist durch Algorithmus 5 gegeben. Diese entfernt alle Klone, die isoliert betrachtet keinen Profit erzeugen und auch nicht nötig sind, um andere Klone zu erreichen.

Algorithm 5 Einfache Auswahl Heuristik

1: **while** $\exists p \in \tilde{G}_{\Lambda^+}$ mit $N^+(p) = \varnothing$ und $\varrho(p) - \gamma(p) < 0$ **do**
2: Entferne p aus $\tilde{G}_{\Lambda^+}$
3: Übernehme alle in $\tilde{G}_{\Lambda^+}$ verbliebenen Prozeduren in das endgültige Programm

Zum jetzigen Zeitpunkt enthält unsere Implementierung noch keine Auswahlstrategie. Hier ist noch eine genauere Untersuchung der Effekte von Procedure Cloning notwendig. Siehe dazu auch Kapitel 6.

4 Evaluation

Im Folgenden soll untersucht werden inwiefern die implementierte Optimierung den Erwartungen hinsichtlich verbesserter Laufzeit der optimierten Programme gerecht wird.

4.1 Testsystem

Das System auf dem die Benchmarks ausgeführt wurden, setzt sich aus folgender Hardware zusammen:

Prozessor	AMD Athlon II X4 635 @ 2,9 GHz
Chipsatz	AMD 880G/SB850
Hauptspeicher	12 GB DDR3-1600 CL9-9-9
Massenspeicher	500 GB Samsung SSD 850 EVO

Als Betriebssystem wurde eine frische Installation von Ubuntu 16.10 verwendet. Die Konfiguration entspricht weitestgehend den Voreinstellungen. Folgende Software-Versionen kamen zum Einsatz:

Linux	4.8.0-28-generic #30-Ubuntu SMP x86_64
cparser	1.22.1 (4ebb0b8...)
libfirm	1.22 (5674832...)
CPU2006	1.1

Alle Benchmarks wurden mit den Optionen `-O3 -m32` kompiliert.

4.2 Leistungsuntersuchung

Zur Untersuchung der Auswirkung der implementierten Optimierung auf die Laufzeit kompilierter Programme wurden alle C-Benchmarks der SPEC CPU2006 Benchmark-

Suite je 10 mal mit und ohne Procedure Cloning ausgeführt. Das System wurde während der gesamten Leistungsmessung nicht anderweitig genutzt.

Da wir vorerst den maximal erzielbaren Laufzeiteffekt untersuchen wollen, wird hier jede Klonmöglichkeit wahrgenommen. Das Ergebnis von Algorithmus 4 wird also nicht weiter eingeschränkt.

Bei den Benchmarks 429.mcf, 462.libquantum und 470.lbm wurden vom Algorithmus keine Prozeduren geklont. Deshalb sind diese Benchmarks nicht in Tabelle 4.1 und Tabelle 4.2 aufgeführt.

Benchmark	Base				Clone			
	$\lvert\Lambda\rvert$	S in KiB	T in s	$\frac{\sigma}{T}$	$\lvert\Lambda\rvert$	S in KiB	T in s	$\frac{\sigma}{T}$
400.perlbench	1.863	2.328	585	4‰	2.160	2.913	581	6‰
401.bzip2	100	85	870	4‰	102	90	874	2‰
403.gcc	5.569	6.324	543	6‰	6.711	8.871	544	6‰
433.milc	235	222	948	2‰	247	238	950	2‰
445.gobmk	2.679	4.234	645	1‰	2.828	4.478	648	1‰
456.hmmer	536	440	970	11‰	564	461	964	1‰
458.sjeng	144	243	757	4‰	151	272	763	2‰
464.h264ref	589	788	1.055	3‰	623	861	1.043	4‰
482.sphinx3	369	242	2.706	4‰	374	246	2.703	1‰

Tabelle 4.1: Ergebnisse von je zehn Durchläufen der SPEC CPU2006 Testsuite. Einmal ohne (Base) und einmal mit (Clone) aktiviertem Procedure Cloning. Dabei ist $\lvert\Lambda\rvert$ die Anzahl der Prozeduren im Programm[1], S die Größe des erzeugten Kompilats in Kibibyte, T die mittlere Ausführungszeit des erzeugten Programms in Sekunden und $\frac{\sigma}{T}$ die relative Standardabweichung der gemessenen Ausführungszeiten.

Wider Erwarten hält sich die Zunahme der Programmgröße selbst ohne jegliche Klon-Auswahl in Grenzen. Im Mittel wächst die Programmgröße nur um 12% an. Nimmt man die drei Programme hinzu, in denen überhaupt nicht geklont wurde, beträgt die Zunahme im Mittel sogar nur 9%.

Bei drei Benchmarks konnte eine Änderung der Laufzeit festgestellt werden, die sich deutlich von der Standardabweichung der Messungen abhebt. In zwei von drei Fällen verschlechterte sich die Laufzeit jedoch. Nur 464.h264ref profitiert von der Optimierung in Form einer um 12‰ kürzeren Laufzeit.

[1]Die Messung erfolgte indem die Anzahl der Prozeduren einmal vor und einmal nach der Durchführung von Procedure Cloning erfasst wurde. Das Ergebnis entspricht also nicht zwangsläufig der Anzahl der Prozeduren zum Zeitpunkt der Erzeugung des Maschinen-Codes.

Benchmark	$\frac{\Delta\lvert\Lambda\rvert}{\lvert\Lambda\rvert_B}$	$\frac{\Delta S}{S_B}$	$-\frac{\Delta T}{T_B}$	$-\frac{\Delta T}{T_B}\frac{S_B}{\Delta S}$	$\frac{\lvert\Delta T\rvert}{\sigma_{\max}}$
400.perlbench	16%	25%	7‰	3%	1,18
401.bzip2	2%	5%	−4‰		1,25
403.gcc	21%	40%	−2‰		0,32
433.milc	5%	7%	−2‰		0,99
445.gobmk	6%	6%	−4‰		**3,98**
456.hmmer	5%	5%	12‰	25%	0,94
458.sjeng	5%	12%	−8‰		**2,21**
464.h264ref	6%	9%	12‰	12%	**3,01**
482.sphinx3	1%	2%	1‰	7%	0,32
Geometrisches Mittel	7%	12%	1‰		

Tabelle 4.2: Statistiken zu den Messungen aus Tabelle 4.1. Für eine Messgröße X bezeichnet X_B die Messung aus dem Base-Lauf und X_C die Messung aus dem Clone-Lauf. Außerdem ist $\Delta X = X_C - X_B$. Die ersten drei Spalten der Tabelle zeigen also das Wachstum der Zahl an Prozeduren, das Wachstum der Programmgröße und die Reduktion der Laufzeit jeweils im Verhältnis zur jeweiligen Basisgröße. Die vierte Spalte zeigt mit dem Verhältnis von Laufzeitreduktion zu Programmwachstum die Effizienz der Optimierung an. Die letzte Spalte zeigt schließlich das Verhältnis von mittlerer Laufzeitdifferenz und der maximalen Standardabweichung der beiden Messungen. Sie gibt somit die Signifikanz der Laufzeitmessung an. Ist dieser Wert kleiner als 2 ist das Ergebnis nur bedingt aussagekräftig. Ist er kleiner als 1, sollten aus der Messung keine Schlüsse gezogen werden [14]. Von den drei Benchmarks mit signifikanten Laufzeitänderungen profitiert nur 464.h264ref von der Optimierung.

Problematisch ist hingegen die Entwicklung der Laufzeit. Auch wenn das Mittel aller Messungen auf den ersten Blick eine minimale Verbesserung vermuten lässt, gibt es doch in vielen Fällen eine Laufzeiterhöhung. Obwohl viele Laufzeit-Differenzen im Bereich der Mess-Ungenauigkeit liegen, muss der Anwender der Optimierung trotz allem das Programmwachstum in Kauf nehmen. Als Negativbeispiel ist das Programm 403.gcc aufzuführen, bei welchem trotz einer um 40% erhöhten Größe keine messbare Laufzeitverbesserung erzielt wurde.

Ein interessanter Effekt der Optimierung lässt sich für den Benchmark 456.hmmer beobachten. Wie aus Tabelle 4.1 zu entnehmen ist, gehört die Standardabweichung des optimierten Programms zu den niedrigsten Werten, während die Messung des Basis-Programms die höchste Standardabweichung aufweist. Auch nach mehreren Versuchen lag diese niemals unter 10‰. Die Anwendung von Procedure Cloning scheint bei diesem Programm also die Varianz der Laufzeit zu reduzieren.

5 Verwandte Arbeiten

Die dem Autor bekannten Arbeiten die sich am ausführlichsten mit Procedure Cloning beschäftigen sind die von Cooper, Hall und Kennedy [1, 11]. Diese wurden bereits ausführlich beschrieben. Sie bilden die theoretische Grundlage für diese Arbeit.

Ayers [12] beschreibt ein Verfahren, welches wiederholt Inlining und Cloning ausführt, um ein Programm zu optimieren. Dabei wird die Auswahl der Stellen, bei denen Klonen bzw. Inlining angewendet wird, von Profiling-Informationen geleitet. Die Analyse zur Identifikation geeigneter Kandidaten zum Klonen ist einfacher gehalten als bei Cooper et al. – sie arbeitet beispielsweise nicht interprozedural. Obwohl Ayers beachtliche Performance-Steigerungen bis zu einem Speedup vom Faktor zwei beschreibt, schien auch hier Procedure Cloning allein die Leistung nicht entscheidend zu beeinflussen:

> For reasons we do not yet completely understand, we have found our implementation of cloning to be relatively ineffective in boosting performance.

Eine weitere Arbeit, die Inlining und Cloning als ein gemeinsames Problem modelliert, ist die von Das [15]. Hier wird die Auswahl passender Stellen zum Inlining und Klonen als gemeinsames Rucksackproblem modelliert und ein zugehöriger Greedy-Algorithmus vorgestellt. Diese Arbeit baut auf dem selben Compiler auf wie die von Ayers.

Ein weiterer interessanter Ansatz zur Auswahl von zu klonenden Prozeduren wird von Way und Pollock [16] beschreiben. Sie ermitteln die zu erstellenden Klone nicht anhand von statischer Programm-Analyse wie bei Cooper et al. [1], sondern durch den Vergleich der verschiedenen *Path Spectra* einer Prozedur. Ein *Path Spectrum* ist die Menge der ausgeführten Pfade während eines Programmlaufs mitsamt ihrer Ausführungshäufigkeit.

6 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde systematisch eine solide Implementierung von Procedure Cloning auf Basis wissenschaftlicher Veröffentlichungen erstellt. Die abstrakten Konzepte aus der Literatur wurden in die konkrete Umgebung von libFirm überführt und an die vorherrschenden Gegebenheiten angepasst.

Das Problem einer optimalen Klonauswahl zum Zweck der Vermeidung übermäßigen Programmwachstums wurde ausgiebig theoretisch betrachtet. Dabei erwiesen sich jedoch selbst stark vereinfachte Varianten dieses Optimierungsproblems als NP-schwer.

Die Evaluation gibt jedoch Anlass zu der Annahme, dass eine Auswahl der Klone nach ihrem Beitrag zum Programmwachstum nicht höchste Priorität haben sollte. Vielmehr stellt sich die Frage, welche Eigenschaften beim Spezialisieren zu einer Verschlechterung der Laufzeit führen können. Ist diese Frage geklärt, gilt es Strategien zu finden, welche die positiven Effekte der Optimierung möglichst erhalten und negative Effekte weitestgehend vermeiden.

Eine weitere Verbesserung der vorliegenden Implementierung wäre außerdem das Entfernen von Prozeduren, welche nach dem Klonen nicht mehr genutzt werden. Dieser Schritt konnte aus Zeitgründen nicht mehr umgesetzt werden, sollte nach Auffassung des Autors aber ohne größere Probleme in das bestehende Programm einzubinden sein.

Zeitgleich mit dieser Arbeit wurde von Mark Weinreuter Unterstützung für Link-Time-Optimization zu libFirm hinzugefügt [17]. Eine Evaluation der Auswirkung auf Procedure Cloning konnte aus Zeitgründen leider nicht mehr durchgeführt werden. Wir vermuten aber, dass unsere Analysen von globaler Information stark profitieren könnten. Eine Anpassung des vorhandenen Codes ist dafür vermutlich nicht nötig.

Ein weiterer Aspekt der möglicherweise Optimierungspotential bietet, ist die Ausführungsreihenfolge der Optimierungen im Compiler. Wie schon in der Einleitung erwähnt können sich Inlining und Procedure Cloning gegenseitig begünstigen. Momentan wird das Procedure Cloning genau einmal direkt nach dem Inlining ausgeführt. So kann der Inliner natürlich nicht von der durch das Klonen verbesserten Information

profitieren. Besonders die Beobachtungen von Ayers [12] legen aber nahe, dass gerade hierdurch hohe Performance-Zuwächse möglich sind.

Schließend lässt sich sagen, dass Procedure Cloning ein umfangreiches Thema mit viel Potential für weitere Forschung ist. Für diese wurde mit der vorliegenden Arbeit eine solide Grundlage geschaffen.

Literaturverzeichnis

[1] K. D. Cooper, M. W. Hall, and K. Kennedy, "A methodology for procedure cloning," *Computer Languages*, vol. 19, no. 2, pp. 105–117, 1993.

[2] J. Hromkovič, *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Texts in theoretical computer science, Springer, 2001.

[3] F. Nielson, H. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer Berlin Heidelberg, 2004.

[4] K. Cooper and L. Torczon, *Engineering a Compiler*. Elsevier Science, 2011.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pp. 1–11, ACM, 1988.

[6] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pp. 12–27, ACM, 1988.

[7] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form," in *Compiler Construction* (R. Jhala and K. Bosschere, eds.), vol. 7791 of *Lecture Notes in Computer Science*, pp. 102–122, Springer, 2013.

[8] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[9] M. Braun, S. Buchwald, and A. Zwinkau, "Firm—a graph-based intermediate representation," Tech. Rep. 35, Karlsruhe Institute of Technology, 2011.

[10] C. Click and M. Paleczny, "A simple graph-based intermediate representation," *SIGPLAN Not.*, vol. 30, pp. 35–49, Mar. 1995.

[11] M. W. Hall, *Managing interprocedural optimization*. PhD thesis, Rice University, 1991.

[12] A. Ayers, R. Schooler, and R. Gottlieb, "Aggressive inlining," *SIGPLAN Not.*, vol. 32, pp. 134–145, May 1997.

[13] G. Borradaile, B. Heeringa, and G. T. Wilfong, "Approximation algorithms for constrained knapsack problems," *CoRR*, vol. abs/0910.0777, 2009.

[14] G. Heiser, "Systems benchmarking crimes." `https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html`. Stand: 5. Dezember 2016.

[15] D. Das, "Function inlining versus function cloning," *SIGPLAN Not.*, vol. 38, pp. 23–29, June 2003.

[16] T. Way and L. Pollock, "Using path spectra to direct function cloning," in *In Workshop on Profile and Feedback-Directed Compilation, International Conference on Parallel Architectures and Compilation Techniques (PACT*, pp. 40–47, 1998.

[17] M. Weinreuter, "Implementation and evaluation of link time optimization with libFirm," Bachelor's thesis, Karlsruher Institut für Technologie (KIT), Nov. 2016.

Erklärung

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

Ort, Datum	Unterschrift