Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 98

Prototypenentwicklung mit Bauhaus und SKilL

Matthias Harrer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dipl.-Inf. Timm Felden
Beginn am:	13. April 2016
Beendet am:	13. Oktober 2016
CR-Nummer:	D.3.3, E.2, F.3.2

Kurzfassung

Bauhaus ist ein Forschungsprojekt, das Werkzeuge zur Analyse von Software beinhaltet. Diese sollen das Verständnis der Programmarchitektur erhöhen und Wartungsarbeiten erleichtern. Als Alternative zum aktuellen, XML-basierten Austauschformat für Daten, wird das Serialisierungstool SKilL entwickelt. Dieses bietet neben anderen Vorteilen eine höhere (De-)Serialisierungsgeschwindigkeit und unterstützt eine Vielzahl an Programmiersprachen. Dadurch wird ermöglicht, dass Programmanalysen in der Sprache implementiert werden können, die dem Entwickler am besten bekannt ist.

In dieser Arbeit soll gezeigt werden, ob sich SKilL im Kontext von Bauhaus zur effizienten Prototypenentwicklung eignet. Dazu werden Prototypen der Zeigeranalysen nach Steensgaard, Das und Andersen entwickelt. Diese stellen anhand ihrer Art und Komplexität ein gutes Beispiel für Analysen dar, die in Zukunft mit SKilL entwickelt werden könnten. Die Prototypen werden automatisiert getestet und funktionieren für die verwendeten, kleinen Beispielprogramme korrekt. Zusätzlich wird die Plausibilität der Resultate für größere Programme durch den Vergleich der Ergebnisse von Steensgaards, Das' und Andersens Analyse gezeigt. Aus den Erfahrungen bei der Implementierung kann abgeleitet werden, dass sich SKilL im Kontext von Bauhaus gut zur Prototypenentwicklung eignet.

Inhaltsverzeichnis

Abkürzungsverzeichnis **7**

1. Einleitung **9**

2. Grundlagen **11**
2.1. Prototypen 11
2.2. Bauhaus 12
2.3. SKilL 12
2.4. Zeigeranalysen 13
2.4.1. Motivation für Zeigeranalysen 14
2.4.2. Klassifizierung von Zeigeranalysen 15
2.5. Union-Find 16
2.6. Zeigeranalyse nach Steensgaard 19
2.6.1. Zeigeranalyse als Typinferenz 19
2.6.2. Typen 20
2.6.3. Typisierungs-Regeln 20
2.6.4. Implementierung mit Union-Find 21
2.6.5. Pending Lists 22
2.6.6. Structs und Unions 22
2.7. Zeigeranalyse nach Das 23
2.7.1. Funktionsweise 24
2.7.2. Indirekte Funktionsaufrufe 24
2.8. Zeigeranalyse nach Andersen 25
2.8.1. Ablauf der Analyse 26
2.8.2. Effizientes Auflösen der Lösungsbedingungen 28

3. Prototypenentwicklung **29**
3.1. Architektur der entwickelten Prototypen 29
3.1.1. Reduktion des Zwischencodes 29
3.1.2. Ablauf der Analysen 32
3.2. Implementierung der Prototypen 32
3.2.1. SKilL-Funktionen 32
3.2.2. Erstellung der reduzierten Darstellung 34
3.2.3. Analyse nach Steensgaard 35
3.2.4. Analyse nach Das 38

3.2.5. Analyse nach Andersen . . . 39
3.3. Schwierigkeiten bei der Implementierung . . . 40
3.3.1. Variable Argumentenlisten . . . 41
3.3.2. Struct Aggregates . . . 42
3.3.3. Beschreibung der Algorithmen . . . 43

4. Evaluation **45**
4.1. Testplattform . . . 45
4.2. Automatisiertes Testen . . . 45
4.2.1. Motivation für automatisiertes Testen . . . 45
4.2.2. Implementierung der Tests für die Prototypen . . . 46
4.3. Analyse von realen Programmen . . . 47
4.3.1. Plausibilität der Analyseergebnisse . . . 48
4.3.2. Präzision der Analysen . . . 49
4.3.3. Laufzeitverhalten der Prototypen . . . 50
4.4. Umsetzung der Analysen mit SKilL . . . 51

5. Zusammenfassung und Ausblick **55**

A. Anhang **57**
A.1. Liste der Programme, die zur Analyse zur Verfügung gestellt wurden . . . 57
A.2. Durchschnittliche Anzahl der Elemente in den erzeugten Zeigerzielmengen . 59
A.3. Maximale Anzahl der Elemente in den erzeugten Zeigerzielmengen . . . 63
A.4. Ausführungszeiten der Analysen . . . 67

Literaturverzeichnis **71**

Abkürzungsverzeichnis

Abkürzung	Bedeutung	Erstes Vorkommen
AST	abstrakter Syntaxbaum	30
ECR	Equivalence Class Representatives	36
IML	InterMediate Language	12
RFG	Resource flow graph	12

1. Einleitung

Bauhaus ist ein Forschungsprojekt der Universitäten Stuttgart und Bremen [EKP+99; RVP06]. Es beinhaltet Werkzeuge zur Analyse von Software, mit den Zielen das Verständnis der Programmarchitektur zu erhöhen und Wartungsarbeiten zu erleichtern. Dabei müssen regelmäßig Daten zwischen unterschiedlichen Werkzeugen ausgetauscht werden. Diese werden dazu in einem auf XML basierten Austauschformat serialisiert und von anderen Programmen eingelesen.

Als Alternative zu diesem Ansatz wird das Serialisierungstool SKilL entwickelt [Fel14]. Es beinhaltet eine domänenspezifische Sprache zur Definition eines Modells der Daten, die gespeichert werden sollen. Aus dieser Spezifikation kann mithilfe des SKilL-Generators Code für verschiedene Programmiersprachen generiert werden, mithilfe dessen die Daten repräsentiert und bei Bedarf serialisiert werden können. Durch die Erstellung einer SKilL-Datenspezifikation für Bauhaus soll die Entwicklung von Prototypen im Umfeld von Bauhaus vereinfacht werden. Dabei soll jeder Mitarbeiter oder Student, der einen Prototypen erstellt, diesen nach Möglichkeit in der Programmiersprache seiner Wahl implementieren können.

Zielsetzung

Im Rahmen dieser Arbeit sollen die Prototypenentwicklungseigenschaften von SKilL mit Bauhaus untersucht werden. Dazu sollen die drei Zeigeranalysen nach Steensgaard, Das und Andersen implementiert werden und auf einer Auswahl, in SKilL-Darstellung vorliegender Programme, ausgeführt werden. Dabei soll es möglich sein, die Ergebnisse unabhängig davon, von welcher Analyse sie erzeugt wurden, zu untersuchen. Weiterhin soll ein Werkzeug geschaffen werden, mithilfe dessen die Ergebnismengen gegenseitig auf Inklusion untersucht werden können. Dies ermöglicht die Überprüfung der Analyseergebnisse auf Sinnhaftigkeit und Plausibilität.

Damit soll untersucht werden in wieweit die Ergebnisse der Analysen sinnvoll und plausibel sind.

Anhand der Entwicklung der Prototypen werden die Prototypenentwicklungseigenschaften von SKilL im Bauhaus-Kontext untersucht. Dabei werden die hilfreichen oder hinderlichen Funktionen und Eigenschaften von SKilL betrachtet und eine Einschätzung zu den Prototypenentwicklungseigenschaften abgegeben.

Gliederung

Anschließend an die Einleitung werden in Kapitel 2 die zum Verständnis notwendigen Grundlagen behandelt. Hier wird der Kontext der Arbeit erläutert und die Algorithmen der implementierten Zeigeranalysen beschrieben.

Anschließend wird in Kapitel 3 auf die entwickelten Prototypen eingegangen. Dabei wird insbesondere die Verwendung von SKilL bei der Implementierung betrachtet. Am Ende werden Hindernisse diskutiert, die während der Entwicklung der Prototypen aufgetreten sind.

In Kapitel 4 wird untersucht, ob die implementierten Analysen sinnvolle Ergebnisse liefern und die Laufzeiten innerhalb der vorgegebenen Schranken liegen. Anschließend wird evaluiert, welche Funktionen von SKilL für die Entwicklung der Prototypen hilfreich waren.

Zuletzt gibt Kapitel 5 eine kurze Zusammenfassung der Arbeit.

2. Grundlagen

In diesem Kapitel werden einige Grundlagen erläutert, die für das Verständnis der folgenden Kapitel notwendig sind.

2.1. Prototypen

In der Softwareentwicklung besteht eine große Herausforderung darin, die Anforderungen an ein Programm aufzustellen [Bro87]. Es können Missverständnisse zwischen Entwicklern und Kunden entstehen, die zur inkorrekten Umsetzung führen. Außerdem weiß der Kunde häufig selbst nicht exakt, welche Anforderungen er an eine Anwendung hat.

Das frühe Entwickeln einer Vorabversion kann helfen, Probleme früh zu erkennen und Anforderungen entsprechend den Vorstellungen des Kunden anzupassen. Eine solche, meist nur eingeschränkt funktionsfähige Vorabversion, wird *Prototyp* genannt.

Es existieren mehrere Möglichkeiten zum Einsatz von Prototypen. Sie können als Basis für die folgende Implementierung dienen und so ständig erweitert werden, bis der finale Funktionsumfang erreicht ist. Außerdem kann ein Prototyp lediglich zur Evaluation des Konzeptes sowie der Verfeinerung von Anforderungen eingesetzt und anschließend verworfen werden. Der Fokus liegt dann stärker auf der Umsetzbarkeit des grundlegenden Konzeptes und weniger auf der Robustheit des Prototyps.

Im wissenschaftlichen Umfeld ist oft letztere Variante vorzufinden. Da es häufig nicht um die Entwicklung eines Produktes geht, sondern um die Bestätigung einer Hypothese oder der Umsetzbarkeit einer entwickelten Lösung, reicht die Implementierung eines Prototyps meist aus. Die aus dem Prototyp gewonnenen Erkenntnisse können veröffentlicht werden und als Grundlage in die Entwicklung eines entsprechenden Produktes einfließen.

Um unterschiedliche Hypothesen und Ansätze möglichst effizient untersuchen zu können, ist es wichtig, dass die Entwicklung eines Prototyps mit geringem Aufwand und in möglichst kurzer Zeit möglich ist.

2.2. Bauhaus

Bauhaus ist ein Toolset, welches Werkzeuge zum Verstehen und zum Reverse Engineering von Programmen (in Ada, C, C++ und Java) beinhaltet [EKP+99; RVP06]. Es definiert dazu zwei Repräsentationen für Programme, auf deren Basis weitere Werkzeuge entwickelt werden können.

Die InterMediate Language (IML) beinhaltet systemnahe Informationen zur Syntax und Semantik der Programme. Sie ähnelt einer Zwischendarstellung, die in Compilern zum Einsatz kommen könnte. Sie ist allerdings darauf optimiert, Programmanalysen eine gute Basis zu bieten. Dafür werden beispielsweise konkrete Programmkonstrukte unter gemeinsamen Obertypen zusammengefasst. Unterschiede zwischen verschiedenen Sprachen werden dadurch genau abgebildet, während Analysen, für die diese Unterschiede irrelevant sind, auf den abstrakteren Obertypen arbeiten können. Ein Beispiel für solch eine Abstraktion sind die Klassen `C_For_Loop` und `While_Loop`, welche mit weiteren Klassen unter der gemeinsamen Oberklasse `Loop_Statement` zusammengefasst werden.

Die Darstellung durch einen Resource flow graph (RFG), stellt Beziehungen zwischen Programmkomponenten auf höherer Ebene dar. Mithilfe einer Graphstruktur, deren Knoten für die Architektur relevante Programmelemente wie zum Beispiel Funktionen darstellen, werden Abhängigkeiten zwischen diesen Programmkomponenten dargestellt. Diese Informationen über Beziehungen können aus Analysen auf IML-Ebene gewonnen werden. Ein Beispiel dafür sind Aufruf-Kanten aus dem Aufrufgraph.

2.3. SKilL

SKilL ist ein Tool, das es ermöglicht, Daten sprachunabhängig zu serialisieren und zu deserialisieren [Fel14]. Die Design-Ziele dabei sind neben der Sprachunabhängigkeit eine einfache Benutzbarkeit, Effizienz, sowie größtmögliche Abwärts- und Aufwärtskompatibilität der zu serialisierenden Datenmodelle.

Um Daten mit SKilL serialisieren zu können, gibt der Benutzer eine Spezifikation des Datenmodells in einer domänenspezifischen Sprache an. Diese erlaubt es, Klassen und Interfaces auf ähnliche Art wie in objektorientierten Programmiersprachen anzugeben (siehe Beispiel in Listing 2.1).

Ein Generator erzeugt aus dieser Spezifikation sprachspezifischen Code, der das Datenmodell abbildet und zusammen mit spezifikationsunabhängigem Code aus einer Bibliothek die Logik bereitstellt, um die Daten zu (de-)serialisieren.

Das Dateiformat für die Serialisierung stellt die Daten binär dar und beinhaltet das Typsystem, wodurch eine hohe Abwärts- und Aufwärtskompatibilität gewährleistet werden kann. Programme können Daten selbst dann lesen, wenn diese dem Programm unbekannte Felder

Listing 2.1 Beispiel für eine Spezifikation in der domänenspezifischen Sprache von SKilL

```
1 Object {
2       string description;
3       Point location;
4 }
5
6 Point {
7       i32 x;
8       i32 y;
9 }
```

enthalten oder dem Programm bekannte Felder nicht enthalten. Werkzeuge, die auf älteren Spezifikationen beruhen, können so oft ohne Anpassungen mit Daten umgehen, die einem erweiterten Datenmodell entsprechen.

2.4. Zeigeranalysen

Zeiger sind spezielle Variablen, die als Wert Speicheradressen annehmen können. Sie bilden damit die grundlegenden Fähigkeiten moderner Prozessoren ab, Daten von bestimmten Adressen zu laden oder Instruktionen an einer Speicherstelle auszuführen. In Programmiersprachen wie C [ISO11] besitzen diese Zeigervariablen einen Typ, der beschreibt, welche Art von Objekt sich an der Adresse befindet.

Dadurch wird es ermöglicht, bei einer *Dereferenzierung* des Zeigers, also dem Laden von Daten an der Adresse, auf die der Zeiger zeigt, die gewünschte Interpretation auf die Daten anzuwenden. Wird also ein Zeiger auf eine Zahl dereferenziert, kann das Ergebnis direkt als Zahl verwendet werden.

Zudem erlaubt es C beispielsweise Zeiger zu inkrementieren. Dabei wird die Adresse, die der Zeiger beschreibt jeweils um die Größe des referenzierten Datums erhöht. Die Anweisung `Zeiger++` lässt den Zeiger also auf die Zahl zeigen, die im Speicher hinter der zuvor referenzierten liegt. In anderen Programmiersprachen ist dies oft nicht zulässig. Sie nennen Variablen, die Speicheradressen beinhalten, oft Referenz statt Zeiger. In dieser Arbeit wird der Begriff Zeiger stellvertretend für sämtliche Konzepte des Speicherns von Adressen verwendet.

Neben Zeigern auf Daten ist es in C auch möglich Zeiger auf Funktionen zu erstellen. Diese können zum Beispiel in Tabellen geführt oder als Parameter übergeben werden, um Funktionen indirekt aufzurufen.

```
1 *x = a * b + c
2 *y = a * b + d
```

```
1 t = a * b
2 *x = t + c
3 *y = t + d
```

Abbildung 2.1.: Das Entfernen gemeinsamer Teilausdrücke. Das nicht optimierte Programmfragment ist *links* und das optimierte *rechts* abgebildet.

2.4.1. Motivation für Zeigeranalysen

Während Zeiger notwendig sind um dynamische Datenstrukturen und dynamisches Verhalten zu ermöglichen, entsteht durch sie ein Problem. Da ein Zeiger generell auf jede andere Variable zeigen kann, ist es oft schwer den Fluss von Daten nachzuvollziehen.

Analysen, die mögliche Ziele eines Zeigers einschränken, können einem Programmierer beim Verstehen eines Programms helfen. Diese *Zeigeranalysen* dienen außerdem als Grundlage für andere Algorithmen.

Um die Codequalität eines Softwareprodukts abzuschätzen, existieren verschiedene Metriken. Beispielsweise kann der Grad an Kopplung zwischen den Modulen einer Anwendung herangezogen werden [SAB+07]. Bei zu starker Kopplung wird ein Modul schlechter austausch- und anpassbar, ohne Änderungen in anderen Bereichen des Programms zu erfordern. Eine Form der Kopplung besteht darin, dass Module Daten über globale Variablen austauschen. Änderungen an einem Modul bedingen bei zu starker Kopplung eventuell Anpassungen an anderen Modulen.

Die Analyse, welches Modul auf welche globalen Datenstrukturen zugreift wird allerdings dadurch erschwert, dass neben dem direkten Zugriff auch ein Zugriff über Zeiger möglich ist. Um diese Fälle ebenfalls zu betrachten, muss eine Zeigeranalyse durchgeführt werden, die bestimmt, welche Zeiger möglicherweise globale Daten referenzieren.

Ein weiteres wichtiges Anwendungsfeld findet sich im Compilerbau. In der Gegenwart von Zeigern sind viele Optimierungen ohne Zeigeranalyse nicht möglich.

Beispielsweise ist eine Inline-Ersetzung von Funktionen, die indirekt über einen Funktionszeiger aufgerufen werden, nicht möglich, auch wenn tatsächlich immer dieselbe Funktion aufgerufen wird. Ein weiteres Beispiel für eine Optimierung, die Wissen über Zeigerziele benötigt, ist das *Entfernen gemeinsamer Teilausdrücke* [GH98].

Wie in Abbildung 2.1 angedeutet, ersetzt der Compiler hier (Teil-)Berechnungen die mehrfach ausgeführt werden müssten. Der Wert wird einmal berechnet und in einer temporären Variable zwischengespeichert. Statt der erneuten Berechnung wird lediglich der Wert aus der temporären Variable gelesen. Auf den ersten Blick wirkt das Beispiel (Abbildung 2.1) korrekt. Zeigt `x` allerdings auf `a` oder `b` müsste der Ausdruck `a * b` neu berechnet werden und die Verwendung des Wertes aus der temporären Variable ist nicht korrekt. Zeigeranalysen ermöglichen es Zeigerziele auszuschließen. In diesem Beispiel würde der Ausschluss von `a` und `b` als mögliche Ziele von `x` ausreichen um die Optimierung durchführen zu können.

Listing 2.2 Code-Beispiel zur Demonstration von Kontext-Sensitivität

```
1  void assign(int **r, int *s) {
2     *r = s;
3  }
4
5  int main(int argc, char *argv[])
6  {
7     int *x, *y, a, b;
8     assign(&x, &a);
9     assign(&y, &b);
10 }
```

2.4.2. Klassifizierung von Zeigeranalysen

Die statische Berechnung der tatsächlichen Zeigerziele ist, genau wie die Berechnung der daraus ableitbaren Alias-Informationen nicht entscheidbar [Lan92]. Aus diesem Grund ist es notwendig Analysen zu entwickeln, die Ungenauigkeiten zulassen und die Zeigerzielmengen überschätzen. Dabei ist ein Kompromiss zwischen Genauigkeit und Analysegeschwindigkeit nötig.

Im Folgenden werden die für diese Arbeit wichtigsten charakteristischen Eigenschaften erläutert, in denen sich die Zeigeranalysen unterscheiden und nach denen sie sich kategorisieren lassen.

Fluss-Sensitivität

Eine flusssensitive Analyse berücksichtigt den Kontrollfluss eines Programms [Har09]. Für jede Stelle im Programm wird eine separate Lösung ermittelt, welche die an dieser Stelle gültigen Informationen über Zeigerziele beinhaltet. Diese Ergebnisse sind deutlich genauer als die einer flussinsensitiven Analyse, die ein Ergebnis erzeugt, welches an jeder Programmstelle gültig sein muss. Flussinsensitive Analysen hingegen skalieren besser auf große Programme.

Kontext-Sensitivität

Die Kontext-Sensitivität beschreibt, ob eine Analyse unterscheidet aus welchem Kontext eine Funktion aufgerufen wird [Har09]. Eine kontextsensitive Analyse analysiert Funktionen in verschiedenen Aufrufkontexten getrennt, während eine kontextinsensitive Analyse alle Aufrufkontexte vereint und die Funktion nur einmal analysiert.

Listing 2.2 zeigt ein Programmbeispiel zur Veranschaulichung der Effekte. Eine kontextinsensitive Analyse mischt die beiden Aufrufe zusammen und kommt zu dem Ergebnis, dass `x` und `y` beide auf `a` und `b` zeigen könnten. Eine kontexsensitive Analyse dagegen trennt die beiden Aufrufe und kommt damit zu dem Ergebnis, dass `x` nur auf `a` und `y` nur auf `b` zeigen kann.

Feld-Sensitivität

Die Feld-Sensitivität beschreibt, wie Felder von Structs oder Klassen behandelt werden. Es gibt drei Alternativen [PKH07]. Im feldinsensitiven Ansatz werden alle Felder eines Structs durch das Struct selbst repräsentiert. Dieser Ansatz ist auch in der Behandlung von Arrays üblich, da dort die Unterscheidung der Elemente aufgrund dynamischer Zugriffe unmöglich ist.

Der feldbasierte Ansatz verhält sich, als ob es nur eine Instanz des Structs gäbe. Das Feld wird unabhängig von der konkreten Instanz als eine Variable modelliert.

Im feldsensitiven Ansatz werden die Felder aller Struct-Instanzen unterschieden und durch eigene Variablen modelliert. Dadurch wird die Menge an Variablen größer, allerdings kann das Ergebnis deutlich genauer werden.

Beachtung der Zuweisungsrichtung

Eine Analyse kann die Zuweisungsrichtung beachten, oder die Zuweisungen `x = y` und `y = x` gleich behandeln. Letzterer Ansatz ist deutlich ungenauer, ermöglicht allerdings unter Einsatz einer Union-Find Datenstruktur nahezu lineare Laufzeiten [Ste96b].

Intra- vs. Interprozedural

Intraprozedurale Analysen betrachten nicht das gesamte Programm, sondern lediglich das Programm innerhalb einer Prozedur. Interprozedurale Analysen hingegen betrachten die Weitergabe von Adressen über Prozedurgrenzen hinweg.

2.5. Union-Find

Der Union-Find Algorithmus erlaubt es disjunkte Mengen zu verwalten und diese zu vereinigen. Er findet in zwei der anschließend beschriebenen Zeigeranalysen Anwendung. Der im Folgenden vorgestellte Algorithmus entspricht dem Algorithmus zu *Disjoint sets*, der von Tarjan beschrieben wird [Tar83]. Die Operation $link$ wird hier $union$ genannt.

Jede Menge wird durch ein eindeutiges Element – das *kanonische Element* – repräsentiert. Das kanonische Element wird verwendet, um Mengen zu identifizieren und zu vergleichen. Die Mengen werden mit drei Operationen manipuliert.

$\boldsymbol{makeset(x)}$ **:** Erzeugt eine neue Menge, welche das Element x enthält. Da die Mengen disjunkt sein müssen, darf x in keiner anderen Menge enthalten sein.

$\boldsymbol{find(x)}$ **:** Gibt das kanonische Element zurück, das die Menge beschreibt, die x enthält.

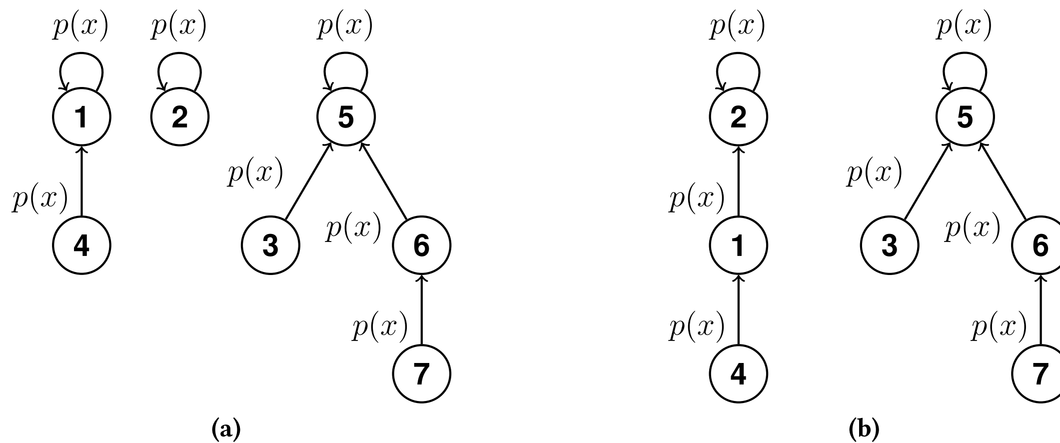


Abbildung 2.2.: Darstellung der disjunkten Mengen $\{1, 4\}$, $\{2\}$ und $\{3, 5, 6, 7\}$ durch Bäume, vor (a) und nach (b) dem Aufruf von $union(2, 4)$.

$\boldsymbol{union(x, y)}$ **:** Bildet die Vereinigung der beiden Mengen, die x beziehungsweise y enthalten. Das kanonische Element der neuen Menge wird zurückgegeben.

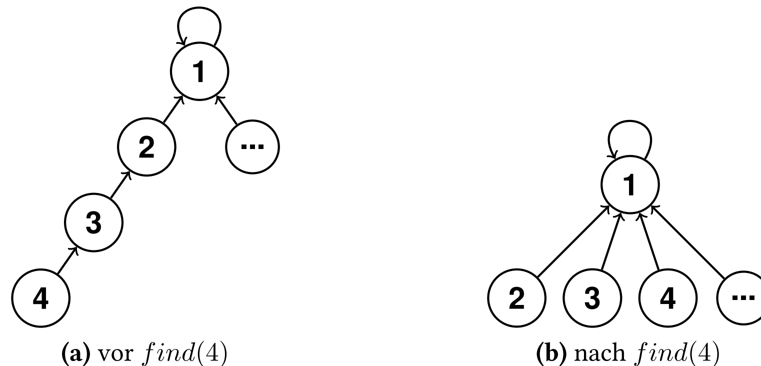
Um diese Operationen effizient zu implementieren, werden die Elemente der Mengen als Knoten in einer Baumstruktur abgelegt. Dies wird in Abbildung 2.2a verdeutlicht. Jeder Knoten besitzt eine Referenz $p(x)$ auf den Elternknoten, wobei die Wurzel das kanonische Element darstellt und auf sich selbst zeigt ($p(x) = x$).

Die Operation $makeset(x)$ besteht darin, x als Wurzel einer neuen Menge zu definieren und dafür $p(x)$ auf x zu setzen (siehe die Menge $\{2\}$ in Abbildung 2.2a).

Für $find(x)$ wird $p(x)$ gefolgt bis $p(x) == x$ gilt und somit die Wurzel gefunden ist. Diese wird als das kanonische Element zurückgegeben. Im Beispiel 2.2a gibt $find(6)$ ebenso wie $find(7)$ den Knoten 5 zurück.

Der Effekt von $union(x, y)$ ist in Abbildung 2.2b für die Mengen $\{1, 4\}$ und $\{2\}$ dargestellt. Der Baum der einen Menge wird unter die Wurzel des anderen Baumes gehängt. Das Ergebnis ist allerdings nicht optimal. Werden die Bäume willkürlich zusammengeführt, können sich, wie in der Abbildung 2.2b angedeutet, lange Ketten bilden. Im schlechtesten Fall benötigt die $find$-Operation auf solch einer Menge n Schritte um die Wurzel zu finden (wobei n die Anzahl der Elemente der Menge ist, die im schlechtesten Fall alle Elemente beinhaltet).

Um den Algorithmus effizient zu machen, sind zwei Optimierungen notwendig. Zum einen das Komprimieren der Pfade bei $find(x)$, sowie zum anderen das Vereinigen nach Rang bei $union(x, y)$.



(a) vor $find(4)$ **(b)** nach $find(4)$

Abbildung 2.3.: Beispiel für den Effekt von Pfad-Kompression beim Aufruf von $find(4)$.

Pfad-Kompression

Beim Aufruf von $find(x)$ wird $p(x)$ solange gefolgt, bis die Wurzel erreicht ist. Um die notwendigen Schritte für folgende Aufrufe zu minimieren, können alle Knoten auf dem Pfad von x zur Wurzel direkt unter die Wurzel gehängt werden. Abbildung 2.3 zeigt dieses Prinzip an einem Beispiel.

Da beim ersten Durchlaufen des Pfades die Wurzel noch nicht bekannt ist, wird ein zweiter Durchlauf benötigt, in dem $p(x)$ für jedes x auf dem Pfad entsprechend angepasst wird. Da $\mathcal{O}(2n) \in \mathcal{O}(n)$ verschlechtert sich die asymptotische Laufzeit dadurch nicht.

Vereinigung nach Rang

Idealerweise wird bei der Vereinigung der kleinere Baum unter der Wurzel des größeren Baumes eingefügt, um die Tiefe des resultierenden Baumes gering zu halten. Da es zu aufwändig ist die Tiefe der Bäume jeweils vor der Vereinigung dynamisch zu ermitteln und sie sich durch die Pfad-Kompression verändern kann, wird der *Rang* als Heuristik für die Tiefe des Baumes verwendet. Er stellt eine obere Schranke dar, die allerdings unterschritten werden kann. Ohne eine Verringerung der Tiefe durch die Pfad-Kompression würde er die tatsächliche Tiefe des Baumes darstellen.

Der Rang wird bei der Erzeugung einer Menge durch $makeset$ mit 0 initialisiert. Beim Aufruf von $union$ wird der Baum mit dem geringeren Rang unter dem mit dem höheren eingehängt. Haben beide denselben Rang, wird der Rang des Baumes inkrementiert, unter dessen Wurzel der andere Baum eingefügt wird.

Listing 2.3 Beispiel: Typinferenz mit Mengen von Integern als Werte

```
1 int a, b;
2 a = 2;
3 b = 3;
4 a = 4;
5
6 a = b;
```

Komplexität

Nach [Tar75] ist der Worst-Case für die amortisierten Kosten von m Aufrufen von $find$ auf n Elementen (= n $makeset$-Aufrufe) mit maximal $n - 1$ $union$-Aufrufen $\Theta(m\alpha(m,n)$. Dabei ist $\alpha(m,n)$ ein funktionales Inverses der Ackermannfunktion, das sehr langsam wächst.

2.6. Zeigeranalyse nach Steensgaard

Die 1996 von Bjarne Steensgaard veröffentlichte Analyse [Ste96b] ist fluss- und kontextinsensitiv und eine der am besten skalierenden interprozeduralen Zeigeranalysen. Die asymptotische Laufzeit ist durch die Verwendung einer Union-Find Datenstruktur nahezu linear ($\mathcal{O}(n\alpha(n,n))$), was die Analyse auf große Programme anwendbar macht. Die geringe Komplexität wird dadurch erreicht, dass Zeigerzielmengen als entweder gleich oder disjunkt betrachtet werden. Dies führt allerdings dazu, dass die Zuweisungsrichtung nicht beachtet werden kann.

Bei einer Zuweisung `a = b` muss die Zeigerzielmenge von `a` zumindest das aktuelle Ziel von `b` enthalten. Da die Zielmengen von `a` und `b` damit nicht mehr disjunkt sind, müssen sie in Steensgaards Modell gleich sein. Die Mengen werden deshalb vereinigt. Der Ansatz wird in der Literatur *equality-based* oder *unification-based* genannt [Das00; Fah00; Hin01].

2.6.1. Zeigeranalyse als Typinferenz

Steensgaard beschreibt seinen Algorithmus als Regeln zur Typinferenz. Typen sind dabei nicht die aus Programmiersprachen bekannten Typen wie `integer` oder `float`, sondern eine Menge von Werten, die potentiell zur Laufzeit angenommen werden. Im Rahmen der Zeigeranalyse stellen die Werte Adressen von abstrakten Speicherstellen dar. Im Folgenden wird das Prinzip mithilfe von Integern an der Stelle der Zeiger verdeutlicht.

Das Beispiel in Listing 2.3 zeigt einige Zuweisungen an die Integer-Variablen `a` und `b`. Bei Betrachtung der Anweisungen bis einschließlich Zeile 4, kann die flussinsensitive Abschätzung aufgestellt werden, dass `a` $\in \{2,4\}$ und `b` $\in \{3\}$ ist. Dabei können die Wertemengen $\{2,4\}$ und $\{3\}$ als Typen betrachtet werden.

Wird die Zuweisung in Zeile 6 hinzugezogen, sind mehrere Verarbeitungsweisen möglich. Lässt man Subtypisierung zu, kann abgeleitet werden, dass der Typ von a ein Supertyp von b sein muss. Dies führt zum genaustmöglichen Ergebnis für eine flussinsensitive Analyse von $type(a) = \{2, 3, 4\}$ und $type(b) = \{3\}$.

Wird Subtypisierung, wie im Fall von Steensgaards Regeln, nicht zugelassen, müssen die Typen von a und b gleich sein. Das führt zu der Abschätzung, dass $type(a) = type(b) = \{2, 3, 4\}$.

2.6.2. Typen

Die Typen zur Beschreibung von Zeigervariablen sind komplexer als die einfachen Wertemengen aus dem vorherigen Abschnitt. Der Grund hierfür liegt darin, dass Ziele von Zeigervariablen selbst Zeiger sein können. Dies wird von Steensgaard modelliert, indem Typen zwei Typkomponenten besitzen. Die eine Komponente beschreibt referenzierte Speicherstellen, während die andere referenzierte Funktionen beschreibt. Dafür wird folgende Produktionsregel gegeben:

$$\begin{aligned}
\alpha &:= \tau \times \lambda \\
\tau &:= \bot \mid ref(\alpha) \\
\lambda &:= \bot \mid lam(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{aligned}$$

Dabei beschreiben die α-Typen Werte, also die Inhalte, die an Speicherstellen stehen stehen. Diese können Adressen enthalten, die entweder andere Speicherstellen oder Funktionen referenzieren. Die Referenzen auf andere Speicherstellen, sowie die Speicherstellen z.B. in Form von Variablen selbst, werden durch die τ-Typen beschrieben. Funktionen bzw. Referenzen auf Funktionen werden durch die λ-Typen beschrieben. $\bot$ ist dabei der *Bottom Type*, der die leere Menge an referenzierten Speicherstellen oder Funktionen darstellt. Speicherstellen, die beispielsweise nur Integer enthalten, können durch den α-Typ $\bot \times \bot$ repräsentiert werden, da sie nichts referenzieren.

Typen sind nur gleich, wenn sie von derselben Typvariablen beschrieben werden oder $\bot$ sind. Das bedeutet unter anderem, dass zwei Variablen mit α-Typ $\bot \times \bot$ verschiedene Typen besitzen. Ein weiteres Beispiel dafür findet sich in Abbildung 2.5b. Die Variablen x und y werden durch zwei Typen (τ_3 und τ_4) beschrieben, die strukturell gleich sind, allerdings trotzdem verschiedene Typen darstellen.

2.6.3. Typisierungs-Regeln

Zur Definition der korrekten Vergabe von Typen führt Steensgaard Regeln auf. Diese beschreiben Bedingungen, die zur korrekten Typisierung erfüllt sein müssen, wenn das Programm eine entsprechende Zuweisung enthält. Einige Beispiele für diese Regeln zeigt Abbildung

$$\frac{A \vdash x : ref(\alpha) \quad A \vdash y : ref(\alpha)}{A \vdash welltyped(x = y)}$$
(a)

$$\frac{A \vdash x : ref(\tau \times _) \quad A \vdash y : \tau}{A \vdash welltyped(x = \&y)}$$
(b)

$$\frac{A \vdash x : ref(\alpha) \quad A \vdash y : ref(ref(\alpha) \times _)}{A \vdash welltyped(x = *y)}$$
(c)

Abbildung 2.4.: Eine Auswahl der Typisierungs-Regeln

2.4. A stellt dabei eine Typumgebung dar, die alle Typen enthält. Jede Variable ist mit einem solchen Typ assoziiert. Die hier dargestellten Regeln sind vereinfacht und verwenden nicht die *Pending-Lists* Optimierung, die in Abschnitt 2.6.5 beschrieben wird.

Zum Beispiel beschreibt die Regel in 2.4a die Bedingungen, die für Typen von Variablen einer Zuweisung gelten müssen, damit insgesamt eine valide Typisierung vorliegt. Dafür müssen die Werte, die an den Speicherstellen der Variablen gespeichert sind, durch denselben α-Typ $\tau \times \lambda$ beschrieben werden. Dies impliziert, dass auch die Zeigerziele beider Variablen durch denselben Typ, nämlich τ beziehungsweise λ, beschrieben werden. Dasselbe gilt rekursiv für deren Zeigerziele.

Abbildung 2.5 ist ein Beispiel für eine Typisierung, die den Regeln entspricht. Nach Regel 2.4b muss für die Zuweisung in Zeile 4 gelten, dass der Typ durch den `a` beschrieben wird gleich dem sein muss, der die Zeigerziele des Typs von `p` beschreibt.

Dies ist in Abbildung 2.5b auch der Fall. `a` wird durch den Typ τ_1 beschrieben und `p` durch den Typ $\tau_2 = ref(\tau_1 \times \bot)$. Dieselbe Regel findet für die Zeilen 5 bis 7 Anwendung.

In Zeile acht wird durch die Zuweisung `x = y` gefordert, dass die Zeigerziele beider Variablen durch denselben Typ beschrieben werden. Da die Ziele von `x` mit demselben Typ wie `p`, sowie die Ziele von `y` mit demselben Typ wie `q` beschrieben werden sollen, müssen `p` und `q` nun durch denselben Typ beschrieben werden. Dies wiederum impliziert, dass auch deren Ziele `a` beziehungsweise `b` durch denselben Typ beschrieben werden müssen.

Der daraus resultierende Points-to-Graph ist in Abbildung 2.5c dargestellt.

2.6.4. Implementierung mit Union-Find

Eine Möglichkeit zu einer validen Typisierung zu gelangen besteht darin, jeder Variable initial einen eigenen Typ zuzuweisen. Diese Typen müssen anschließend entsprechend der Bedingungen, die durch die Regeln gegeben sind, zusammengefasst werden. Das funktioniert, da die Regeln nur Bedingungen bezüglich der Gleichheit von Typen aufstellen.

Um das Zusammenfassen von Typen effizient zu implementieren, können Typen als Mengen in einem Union-Find Algorithmus modelliert werden. Zu Beginn existiert für jede Variable, Speicherstelle oder Funktion ein eigener Typ, der eine von den anderen disjunkte Menge darstellt.

Anschließend wird für jedes Statement die passende Regel betrachtet und die entsprechenden *Typ-Mengen*, eventuell rekursiv, vereinigt.

2.6.5. Pending Lists

Steensgaard beschreibt die Typregeln nicht wie hier dargelegt, sondern in einer optimierten Form. Illustriert wird diese anhand des Beispiels in 2.6.

Die Typisierung in Abbildung 2.6c, welche die Typregeln aus 2.6a erfüllt, beschreibt die Zeigerziele von a, x und y durch denselben Typ. Dies ist allerdings nicht notwendig, da a immer ein Integer und nie ein Zeiger ist. Durch die Zuweisung von a an x und y können somit keine Zeigerziele zugewiesen werden.

Bei einer Zuweisung `x = a` müssen die Zeigerziele also nur durch denselben Typ beschrieben werden, falls a Zeigeradressen enthalten kann. Eine erweiterte Typregel, die dies berücksichtigt, zeigt Abbildung 2.7.

Für die Implementierung ergibt sich die Herausforderung, dass aufgrund der Flussinsensitivität, die Besuchsreihenfolge der Anweisungen nicht definiert ist. Zum Zeitpunkt der Zuweisung `x = a` kann also nicht mit Sicherheit festgestellt werden, ob a Adressen enthält oder nicht. Aus diesem Grund wird für jeden Typ eine *Pending*-Liste geführt. Ist der Typ von a zum Betrachtungszeitpunkt der Zuweisung $ref(\bot, \bot)$, so werden die Typen nicht vereinigt, sondern dem Zieltyp von a wird der Zieltyp von x als *pending* zugewiesen. Sofern sich der Typ von a im weiteren Verlauf der Analyse verändert, wird die Vereinigung mit allen Typen in der Pending-Liste nachgeholt.

2.6.6. Structs und Unions

Steensgaard schlägt in [Ste96a] eine Erweiterung des Typsystems und der Typregeln vor, die es erlaubt, Structs und Unions feldsensitiv zu behandeln. Dadurch können deutlich genauere

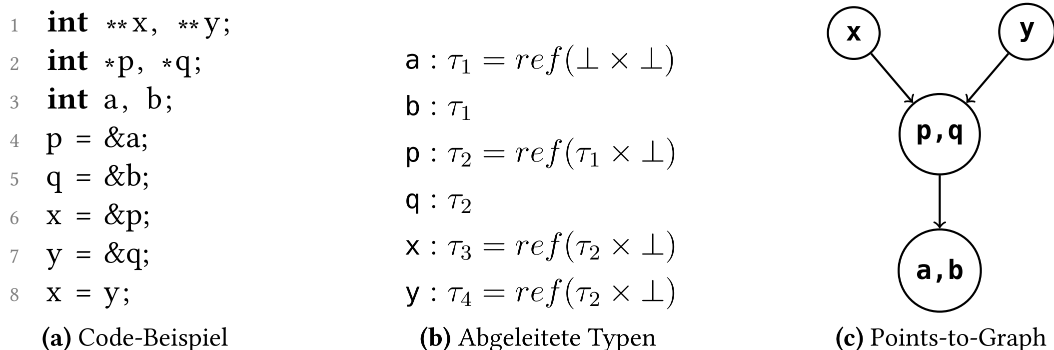


Abbildung 2.5.: Beispiel für Typinferenz nach Steensgaard

$$\frac{A \vdash x : ref(\alpha) \quad A \vdash y : ref(\alpha)}{A \vdash \mathit{welltyped}(x = y)}$$

(a) Verwendete Typregel

```
a = 4
x = a
y = a
```

(b) Code-Beispiel

a : $ref(\tau_1 \times \lambda_1)$
x : $ref(\tau_1 \times \lambda_1)$
y : $ref(\tau_1 \times \lambda_1)$

(c) Typisierung des Code-Beispiels

Abbildung 2.6.: Ungenauigkeit durch Zuweisung von Variablen, die keine Zeiger enthalten.

$$\frac{A \vdash x : ref(\alpha_1) \quad A \vdash y : ref(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(x = y)}$$

(a) Optimierte Typregel

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$
$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4)$$

(b) Definition der Operation $\trianglelefteq$

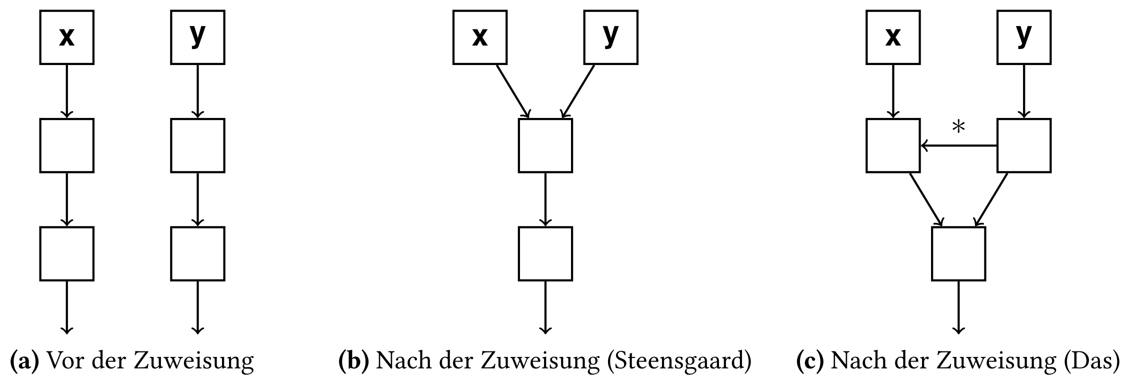
Abbildung 2.7.: Eine optimierte Typregel, die Zuweisungen von Nicht-Zeigervariablen ignoriert.

Analyseergebnisse erzielt werden. Auf diese Erweiterung wird hier allerdings nicht weiter eingegangen, da die Anpassungen am Typsystem und den Typregeln sehr groß sind.

2.7. Zeigeranalyse nach Das

Die Analyse von Manuvir Das wurde als Erweiterung zu Steensgaards Analyse veröffentlicht [Das00]. Sie ist ebenfalls fluss- und kontextinsensitiv, beachtet aber teilweise die Zuweisungsrichtung. Dies erhöht die Genauigkeit und ebenfalls die Laufzeit der Analyse. Die obere Schranke für die asymptotische Laufzeit ist $\mathcal{O}(n^2)$. Laut [Das00] werden für praktische Programme annähernd so genaue Ergebnisse erreicht, wie mit Andersens Analyse, wobei die Laufzeit linear mit der Länge der Programme zunimmt.

Die Analyse baut auf dem Typsystem aus Steensgaards Analyse auf und führt sogenannte *Flow-Kanten* ein, welche Teilmengen-Beziehungen ausdrücken. Bei einer Zuweisung `x = y` mit den Typen x : $ref(\tau_1 \times \lambda_1)$ und y : $ref(\tau_2 \times \lambda_2)$ werden diese gerichteten Kanten von τ_2 nach τ_1 und von λ_2 nach λ_1 eingeführt. Diese Typen müssen so nicht vereinigt werden, womit eine höhere Genauigkeit erreicht wird. Um zu verhindern, dass diese Teilmengen-Beziehungen iterativ aufgelöst werden müssen, werden die Typen, mit denen die Ziele von τ_1 etc. beschrieben werden, vereinigt. Die Zuweisungsrichtung wird also nur auf oberster Ebene berücksichtigt, während alle weiteren Ebenen ungerichtet verarbeitet werden. Abbildung 2.8 zeigt die Unterschiede in der Verarbeitung einer Zuweisung zwischen Steensgaards und Das' Analyse.



(a) Vor der Zuweisung **(b)** Nach der Zuweisung (Steensgaard) **(c)** Nach der Zuweisung (Das)

Abbildung 2.8.: Vergleich der Verarbeitung von Zuweisungen durch Steensgaard und Das

2.7.1. Funktionsweise

Die Analyse läuft in zwei Phasen ab. Die erste Phase entspricht im Allgemeinen Steensgaards Algorithmus. Der einzige Unterschied liegt in der Verarbeitung von Zuweisungen. Anstatt der Vereinigung der Ziel-Typen, werden die Flow-Kanten eingeführt. Dies gilt nicht nur für Zuweisungen wie `x = y`, sondern entsprechend auch für Dereferenzierungen oder Parameterübergaben bei Funktionsaufrufen. Die rekursive Behandlung der Ziel-Typen erfolgt nach Steensgaards Algorithmus.

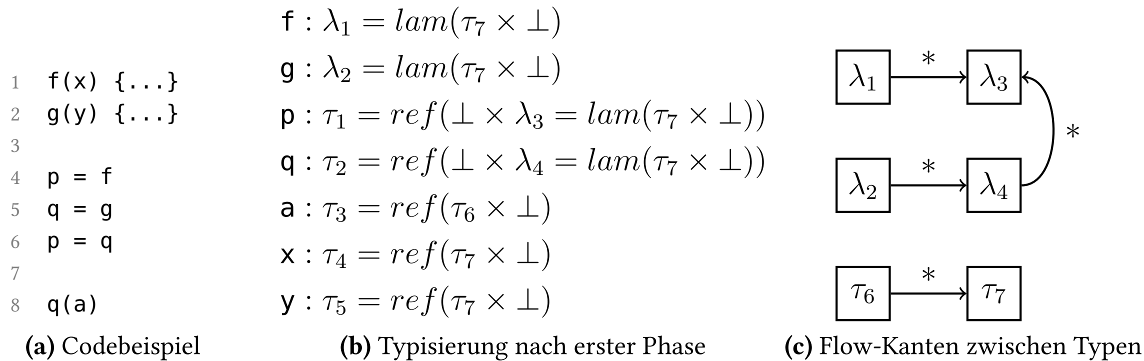
In der zweiten Phase werden die Zeigerziele, wie Variablen, Funktionen oder dynamisch allozierte Speicherstellen, entlang der Flow-Kanten propagiert. Dieser Schritt erhöht die Komplexität im Vergleich mit Steensgaards Analyse auf $\mathcal{O}(n^2)$. Im schlechtesten Fall wird jede Speicherstelle über die Flow-Kanten zu jedem Typ-Knoten propagiert.

2.7.2. Indirekte Funktionsaufrufe

Das gibt in seiner Veröffentlichung nur Typregeln für eine intraprozedurale Analyse an. Regeln für Funktionsaufrufe und Funktionszeiger fehlen und er beachtet Funktionszeigertypen in seinen Typregeln nicht.

Der konsequenteste Weg, Funktionszeiger im Sinne der Analyse zu behandeln, ist, Flow-Kanten zwischen den λ-Typen einzuführen und die α-Typen, welche die formalen Parameter beschreiben, zu vereinigen. Damit ist die höhere Genauigkeit der Analyse auf die Ausgabe der Funktionszeigerziele, also den Call-Graph limitiert. Für die Parametertypen wird die Zuweisungsrichtung nicht berücksichtigt.

Abbildung 2.9 zeigt ein Codebeispiel mit den Funktionszeiger `p` und `q`. Dazu werden eine valide Typisierung und die Flow-Kanten zwischen diesen Typen dargestellt. Funktionen und Funktionszeiger werden alle durch unterschiedliche λ-Typen repräsentiert. Die Ziele der formalen Parameter werden allerdings, wie bei Steensgaards Analyse, durch einen gemeinsamen Typ



(a) Codebeispiel **(b)** Typisierung nach erster Phase **(c)** Flow-Kanten zwischen Typen

Abbildung 2.9.: Beispiel für eine Möglichkeit der Behandlung von Funktionszeigern und indirekten Funktionsaufrufen in der Analyse nach Das.

τ_7 beschrieben. Die eingeführten Flow-Kanten zwischen den Typen sind in Abbildung 2.9c dargestellt. Typen ohne Flow-Kanten sind zur besseren Übersicht nicht dargestellt.

Jede Flow-Kante entsteht durch (mindestens) eine Zuweisung. Die Kanten zwischen den λ-Typen werden aufgrund der Zuweisungen in Zeile 4 bis 6 (Abbildung 2.9a) erzeugt. Die Kante zwischen τ_6 und τ_7 symbolisiert die implizite Zuweisung der Parameter beim Funktionsaufruf in Zeile 8.

Im zweiten Schritt der Analyse werden die von den Typen beschriebenen Speicherstellen entlang der Flow-Kanten weitergegeben. Der Typ λ_1 beschreibt beispielsweise die Speicherstelle der Funktion `f`, welche entlang der Flow-Kante zu λ_3 propagiert wird. Das bedeutet, dass sich `f` in den Zeigerzielen von p befindet (Abbildung 2.9a, Zeile 4). Ebenso wird die Speicherstelle von `q` von λ_2 zu λ_4 (Zuweisung in Zeile 5) und anschließend zu λ_3 (Zuweisung in Zeile 6) weitergegeben. Für die Funktionszeiger wird in diesem Beispiel somit die Zuweisungsrichtung beachtet. In den Zielen von `p` befinden sich `f` und `g`, wohingegen `q` nur auf `g` zeigen kann.

2.8. Zeigeranalyse nach Andersen

Die 1994 von Lars Ole Andersen veröffentlichte Zeigeranalyse ist die älteste hier vorgestellte [And94]. Sie ist die genauste der drei Analysen und in der Grundform wie die anderen fluss- und kontextinsensitiv. Andersen gibt selbst keinen optimalen Algorithmus an und beschreibt die Komplexität seines Algorithmus nur als *polynomial*. Andere Arbeiten haben gezeigt, dass sich die Analyse mit kubischer Worst-Case Komplexität implementieren lässt [HL07; HT01].

Andersens Analyse kann, analog zu der von Steensgaard oder Das, als Algorithmus zur Typinferenz betrachtet werden. Im Vergleich zu beiden anderen Zeigeranalysen, lässt Andersens Analyse Subtypisierung ohne Einschränkung zu.

Der Algorithmus, den Andersen vorschlägt, modelliert das Problem auf andere Weise. Anstelle der Verwendung von Typen, beschreibt er Variablen, Funktionen oder andere Objekte im Speicher durch abstrakte Speicherstellen (*abstract locations*). Diese besitzen eine Zeigerzielmenge, welche die abstrakten Speicherstellen beinhaltet, auf die zur Laufzeit gezeigt werden kann. Um Mengen herzuleiten, die eine konservative Abschätzung der tatsächlich möglichen Zeigerziele darstellen, werden anhand der vorhandenen Anweisungen im Programm Lösungsbedingungen aufgestellt. Durch das iterative Lösen dieser Bedingungen wird die Abschätzung der Zeigerziele berechnet.

2.8.1. Ablauf der Analyse

Zu Beginn der Analyse werden durch einen Pass über das Programm, initiale Lösungsbedingungen, die zwischen den abstrakten Speicherstellen gelten müssen, aufgestellt. Hierbei unterscheidet Andersen vier verschiedene Arten von Zuweisungen, auf die alle Sprachkonstrukte von C abgebildet werden können.

Typ 1) x = & Die Zuweisung der Adresse einer Variablen an eine andere Variable wird direkt als points-to-Beziehung gesehen.

Typ 2) x = y Die Zuweisung einer Variablen an eine andere erzeugt eine Subset-Bedingung. Die Zeigerzielmenge von y muss eine Teilmenge der von x sein.

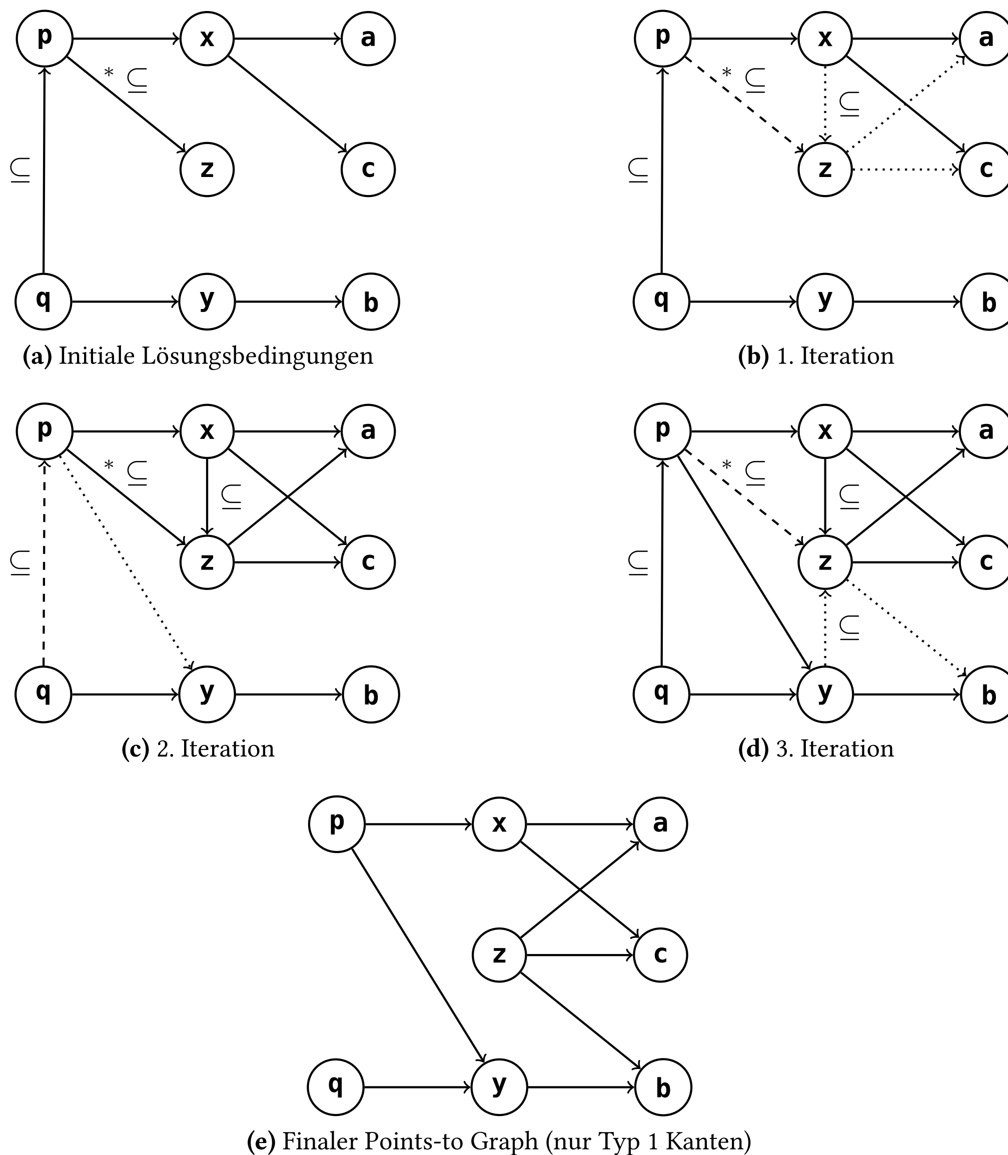
Typ 3) x = *y Die Zuweisung einer dereferenzierten Variablen wird in einer Bedingung abgebildet, die aussagt, dass alle abstrakten Speicherstellen in der Zielmenge von y in einer Typ 2 Beziehung mit x stehen müssen.

Typ 4) *x = y Analog zur vorigen Zuweisung wird hierfür eine Bedingung eingeführt, welche bedeutet, dass zwischen y und allen Speicherstellen auf die x zeigt, eine Typ 2 Beziehung bestehen muss.

Die Lösungsbedingungen werden als gerichtete Kanten (der entsprechende Typ ist annotiert), mit den abstrakten Speicherstellen als Knoten, in einem Graph festgehalten.

In der zweiten Phase des Algorithmus werden neue Kanten in den Graph eingefügt. Ziel der Analyse ist es sämtliche Kanten von Typ 2-4 durch Typ 1 *Points-to*-Kanten auszudrücken. Für eine Typ 2 Kante von y nach x (x = y) werden Typ 1 Kanten von x zu allen Zielen (Typ 1 Kanten) von y eingeführt. Für Typ 3 Kanten von y nach x werden von den Zielen von y zu x Typ 2 Kanten erzeugt. Analog werden für Typ 4 Kanten von y nach x Typ 2 Kanten erzeugt, die von y auf die Ziele von x zeigen.

Da es möglich ist, dass beispielsweise nach der Verarbeitung einer Typ 2 Kante (von y nach x) an anderer Stelle eine zusätzliches Zeigerziel zur Zielmenge von y hinzukommt, müssen die Kanten möglicherweise mehrfach betrachtet werden. Es wird solange über die Kanten iteriert, bis keine neuen Kanten mehr erzeugt werden.



(a) Initiale Lösungsbedingungen

(b) 1. Iteration

(c) 2. Iteration

(d) 3. Iteration

(e) Finaler Points-to Graph (nur Typ 1 Kanten)

Abbildung 2.10.: Möglicher Ablauf von Andersens Algorithmus zum Umschreiben der Lösungsbedingungen für das Beispiel in Listing 2.4. Die gerade betrachtete Kante ist gestrichelt dargestellt. Die Kanten die durch die Bearbeitung (rekursiv) neu eingefügt werden sind gepunktet. Typ 1 Kanten sind ohne Beschriftung dargestellt. Die vorkommenden Typ 2 Kanten sind durch das Label $\subseteq$ und die Typ 3 Kanten durch das Label $^* \subseteq$ gekennzeichnet.

Listing 2.4 Code-Beispiel

```
int a, b, c;
int *x, *y, *z;
int **p, **q;

x = &a;
y = &b;
x = &c;
p = &x;
q = &y;
p = q;
z = *p;
```

2.8.2. Effizientes Auflösen der Lösungsbedingungen

Anstatt Kanten willkürlich aufzulösen, kann eine Optimierung vorgenommen werden. Die Einführung einer Typ 2 Kante (durch die Verarbeitung einer Typ 3 oder Typ 4 Kante) kann keine anderen Kanten im Graph beeinflussen. Es ist also sinnvoll nach dem Erstellen einer Typ 2 Kante, diese direkt zu verarbeiten und die entsprechenden Typ 1 Kanten zu erzeugen. Da Typ 1 Kanten allerdings jede andere Kante beeinflussen können, müssen anschließend wieder alle Kanten betrachtet werden, bis sich keine Änderung mehr ergibt.

Abbildung 2.10 zeigt ein Beispiel für den Algorithmus. Im Teil b) wird beispielsweise die Kante (Typ 3) von `p` nach `z` betrachtet. Für sie wird eine Typ 2 Kante von Zielen von `p` (zu diesem Zeitpunkt `x`) nach `z` eingefügt. Die Typ 1 Kanten die ebenfalls gepunktet dargestellt sind entstehen durch die Verarbeitung dieser Typ 2 Kante. Durch Teil c) und d) wird klar, weshalb eine iterative Auflösung notwendig ist. Die in Teil c) erstellte Kante von `p` nach `y` erfordert die erneute Betrachtung der Kante von `p` nach `z`.

Sobald sich keine neuen Kanten ergeben symbolisieren die Typ 1 Kanten eine konservative Abschätzung der Zeigerziele.

3. Prototypenentwicklung

In diesem Kapitel wird auf die entwickelten Prototypen für die drei Zeigeranalysen nach Steensgaard, Das und Andersen eingegangen. Zu Beginn wird die grundlegende Architektur erläutert. Anschließend werden die verwendeten SKilL-Funktionen und die konkrete Verwendung von SKilL für die Implementierung der einzelnen Analysen beschrieben.

3.1. Architektur der entwickelten Prototypen

Als Grundlage für die Analysen dienen vollständig gelinkte C Programme, die in der Bauhaus-Zwischendarstellung vorliegen und mit SKilL serialisiert sind.

C Programme, die als Quellcode vorliegen, müssen zunächst mithilfe des Bauhaus-Compilers in diese Zwischendarstellung übersetzt werden. Da der Compiler nicht direkt Dateien im SKilL-Format erzeugen kann, müssen die entstehenden IML-Dateien nach der Übersetzung mithilfe des `iml2sf` Tools konvertiert werden.

Die auf diesen Daten operierenden Analysen werden in zwei Phasen ausgeführt. Während der ersten Phase wird die Zwischendarstellung (IML) des Bauhaus-Compilers auf eine kompaktere, für die Zeiger-Analysen optimierte Darstellung reduziert. Diese wird zusätzlich zur Bauhaus-Darstellung serialisiert und enthält Referenzen auf die ursprüngliche Struktur. In der zweiten Phase wird der Algorithmus der jeweiligen Analyse auf dieser reduzierten Darstellung ausgeführt.

3.1.1. Reduktion des Zwischencodes

Die implementierten Zeiger-Analysen benötigen alle deutlich weniger Informationen über das zu analysierende Programm, als in der Zwischendarstellung von Bauhaus enthalten sind. Die Analysen nach Steensgaard und Das werden auch von den Autoren nicht für C, sondern für eine reduzierte Sprache beschrieben, welche nur die relevanten Anweisungen enthält. Zudem sind alle Analysen fluss- und kontextinsensitiv und benötigen deshalb kein Wissen über den Kontrollfluss oder die Aufrufkontexte.

Durch das Herausfiltern der relevanten Informationen entsteht für alle Analysen eine einheitliche Basis. Dies erleichtert die Implementierung der Analysen, da die Datenstruktur

UserLocation	eine Variable / einen Parameter
Constant	ein Literal
AnonymousLocation	ein anonymes Objekt
Address	eine Adressnahme (&-Operator)
Equals	eine Zuweisung
Allocation	Speicherplatz der z.B. durch `malloc` alloziert wird
IndirectRoutineCall	das Ergebnis eines indirekten Funktionsaufruf
ReturnLocation	eine virtuelle Speicherstelle, die alle Rückgaben eine Funktion sammelt
DereferencingRead	ein dereferenzierendes Lesen
DereferencingWrite	ein dereferenzierendes Schreiben
RoutineAddress	eine Adressnahme einer Funktion
RoutineLocation	eine Funktion
UnaryOperation	eine unäre Operation
Operation	eine Operation mit beliebig vielen Operanden

Tabelle 3.1.: Die `Location` Subklassen und ihre jeweilige Bedeutung.

übersichtlicher ist als die IML. Es können für die Analyse irrelevante Knoten, wie Typkonversionen, ausgelassen, oder beispielsweise direkte Funktionsaufrufe in die entsprechenden Zuweisungen für die Parameterübergabe und die Rückgabe übersetzt werden[1]. Zusätzlich vereinfacht dies nachträgliche Änderungen, wie zum Beispiel das Unterstützen einer anderen Programmiersprache oder das Behandeln eines noch nicht berücksichtigten Sprachelements.

Die reduzierte Programmstruktur ist eine Art abstrakter Syntaxbaum (AST), welcher nur die Informationen enthält, die von den Zeigeranalysen benötigt werden. Da keine Informationen über den der Kontrollfluss enthalten sind, hängen die Teilbäume, welche Anweisungen repräsentieren, nicht zusammen, sondern bilden einen Wald. Die Darstellung basiert auf der Modellierung von *virtuellen Speicherstellen*, die durch eine abstrakte Basisklasse `Location` repräsentiert werden. Diese virtuellen Speicherstellen beschreiben Variablen, Parameter, Funktionen, auf dem Heap allozierte Objekte, sowie Zwischenergebnisse, die eventuell nicht real im Speicher vorliegen.

Durch die Anordnung im Baum beschreiben die Subklassen (siehe Tabelle 3.1) die Datenflussbeziehungen zwischen den Speicherstellen. Beispielsweise referenziert eine `Location` vom Typ `Address` eine andere `Location`, deren Adresse sie darstellt.

[1]Dies ist nur möglich, da die Analysen alle kontextinsensitiv sind und Aufrufe daher wie diese Zuweisungen behandeln.

Listing 3.1 Zerlegen eines komplexen Ausdrucks.

```
1     Komplexer Ausdruck:
2     *x = **y
3
4     In einfache Ausdrücke zerlegt:
5     t1 = *y
6     t2 = *t1
7     t3 = t2
8     *x = t3
```

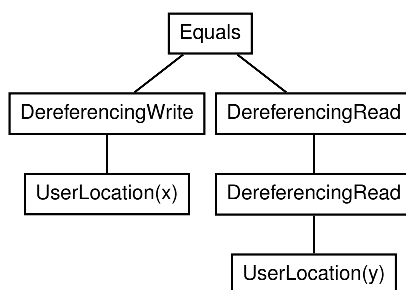


Abbildung 3.1.: Generierte Struktur für das Beispiel `*x = **y`.

Komplexe Ausdrücke, die von den Zeigeranalysen nicht direkt verarbeitet werden können, sind im Baum automatisch in atomare Ausdrücke zerlegt. Listing 3.1 zeigt diese Zerlegung für einen Ausdruck mit mehrfacher Dereferenzierung in textueller Form. Der äquivalente, im Programm erzeugte Baum, ist in Abbildung 3.1 veranschaulicht.

Die entstehenden Ausdrücke erzeugen für die benannten Objekte (im Beispiel `x` und `y`) Zeigerzielmengen, welche dieselben benannten Ziele enthalten. Es können allerdings auch temporäre Speicherstellen, wie zum Beispiel ein `DereferencingRead`, in den Zielmengen auftreten. Diese müssen anschließend aus den Zeigerzielmengen entfernt werden, um nutzbare Ergebnisse zu liefern.

Zur Abbildung der Zeigerziele referenziert jede Instanz von `Location` ein `PointsToSet`, welches wiederum eine Menge an `Location`-Objekten enthält, die potentielle Zeigerziele repräsentieren (siehe Abbildung 3.2). Dabei enthält das `Location`-Objekt die potentiellen Zeigerziele nicht direkt, um die Verwendung eines `PointsToSets` in mehreren `Locations` zu ermöglichen. Das ist insbesondere notwendig um das Ergebnis der Analyse nach Steensgaard in linearer Größe darzustellen.

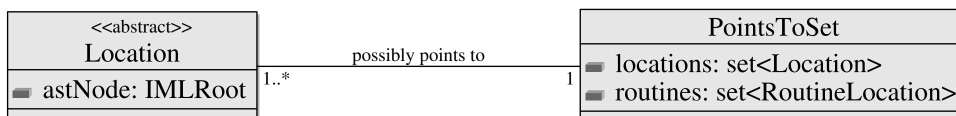


Abbildung 3.2.: Vereinfachtes Klassendiagramm der Klasse `Location`.

Die entstehende Darstellung ist feldbasiert, was bedeutet, dass Struct-Felder behandelt werden, als gäbe es nur eine Instanz des Structs (siehe Kapitel 2.4.2). Die Ergebnisse werden damit deutlich ungenauer, als mit einem feldsensitiven Ansatz. Bei Struct-Typen mit wenigen Feldern, welche häufig instanziiert werden, ist auch ein feldinsensitiver Ansatz genauer, da statt einem Objekt pro Struct-Feld ein Objekt pro Instanz verwendet wird. Allerdings wird die Implementierung durch den feldbasierten Ansatz erleichtert und es ergeben sich keine Auswirkungen auf die Untersuchung der Prototypenentwicklungseigenschaften von SKilL.

Da die Analysen auf der gemeinsamen Zwischenstruktur arbeiten, könnte eine feldsensitive Variante durch alleinige Anpassung des Programms zur Erstellung der Zwischenstruktur realisiert werden.

3.1.2. Ablauf der Analysen

Jede Analyse ist als separates Programm implementiert. Bei der Ausführung wird zunächst die zuvor erstellte Struktur aus einer Datei eingelesen. Anschließend wird der Algorithmus der Analyse ausgeführt. Dieser verwendet je nach Bedarf zusätzliche Datenstrukturen. Ein Beispiel ist die Union-Find Struktur, die von Steensgaard und Das verwendet wird. Nach der eigentlichen Analyse werden die Ergebnisse in die, im vorigen Abschnitt erläuterte Struktur überführt. Damit können die Ergebnisse anschließend einfacher verglichen werden.

3.2. Implementierung der Prototypen

In diesem Kapitel wird die konkrete Implementierung der verschiedenen Programme erläutert. Zunächst wird allerdings auf die speziellen Funktionen von SKilL, die für die Implementierung wichtig sind, eingegangen.

3.2.1. SKilL-Funktionen

Diese Erläuterung der Funktionen von SKilL erhebt keinen Anspruch auf Vollständigkeit, sondern soll helfen die Implementierung der Analysen zu verstehen und den Nutzen von SKilL für die Implementierung zu erkennen. Einige Features unterscheiden sich je nach Zielsprache. Da sämtliche Prototypen in Scala geschrieben sind, stellen die hier besprochenen Funktionalitäten die in der SKilL-Implementierung für Scala beobachteten Funktionen dar.

Listing 3.2 Code-Beispiel zur Verwendung der Iteratoren, die über die Pools von SKilL bereitgestellt werden. Der Pool ist ein Feld des `skillfile`-Objektes mit dem Namen des Typs der verwalteten Objekte. In diesem Beispiel ist der Typ `SkillType`.

```
1 // Öffnen einer SKilL-Datei
2 var skillfile = SkillFile.open(filepath)
3 // Nutzen des Iterators
4 for(object <- skillfile.SkillType.all) {
5       // Verarbeiten des Objekts
6       doSomething(object)
7 }
```

Storage Pools

Der SKilL-Generator erzeugt für jeden SKilL-Typ einen *Pool*, über den ein Iterator für alle Objekte des entsprechenden Typs abgerufen werden kann. Listing 3.2 zeigt ein Beispiel für die Nutzung der Pools.

Im Rahmen dieser Arbeit ist ein sehr wichtiges Feature, dass diese Pools für alle Typen innerhalb der Typhierarchie zur Verfügung gestellt werden. Dabei beinhaltet der Pool für einen Typ auch Subpools für Subtypen. Die Objekte dieser Subtypen werden über den Iterator ebenfalls verfügbar gemacht. Dies erlaubt es, sowohl sehr spezielle Typen zu betrachten, als auch, an Stellen an denen die Details nicht relevant sind, auf abstrakten Obertypen zu arbeiten.

Wie in Abschnitt 2.2 beschrieben, implementiert die IML das Konzept möglichst viele Gemeinsamkeiten zwischen Klassen in Oberklassen zusammenzufassen. Diese Abstraktionen können mithilfe der SKilL-Pools direkt verwendet werden.

Interfaces

Interfaces in SKilL können im Verleich zu Interfaces in Sprachen wie Java [GJSB05] eher als *Mixin* [BC90] gesehen werden. Sie bieten eine Möglichkeit, Felder anzugeben, die in Klassen eingefügt werden, welche das Interface implementieren. Zusätzlich kann das Interface dabei Felder von anderen Interfaces und Klassen erben. Wenn ein Feld in eine Klasse eingefügt wird, welches diese schon besitzt, wird ein Fehler erzeugt. Das Interface sagt also nicht nur aus, dass ein spezielles Feld vorhanden sein muss, sondern sorgt dafür dass dieses erzeugt wird.

Auto Fields

Viele Algorithmen benötigen neben den Eingabe- und Ausgabedaten zusätzliche Felder, die lediglich zur Berechnung verwendet werden und für das Ergebnis irrelevant sind. Mit dem `auto`-modifier erlaubt es SKilL solche Felder in eine Spezifikation einzufügen. Die generierten Klassen enthalten die entsprechenden Felder, diese werden jedoch nicht serialisiert. Aufgrund der Auf- und Abwärtskompatibilität des Formats, können die Felder in den Spezifikationen von

Listing 3.3 SKilL-Spezifikation, die Custom Fields für Scala verwendet. Das Beispiel ist angelehnt an die Spezifikation aus Listing 2.1.

```
1 Object {
2       string description;
3
4       custom Scala
5       !modifier "/* no modifier */"
6       "myScalaPackage.myPoint" location;
7 }
```

Programmen, welche die Daten lediglich lesen wollen, ausgelassen werden. Dies ermöglicht Spezifikationen, die keine irrelevanten Felder enthalten.

Custom Fields

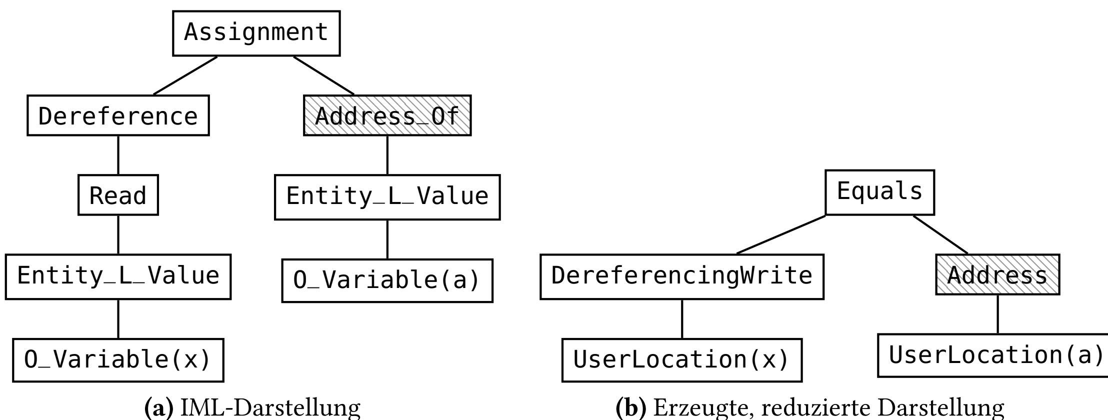
Normalerweise kann ein SKilL-Typ nur Felder besitzen, deren Typ entweder ein primitiver Typ in SKilL ist, oder ebenfalls in der Spezifikation vom Benutzer angegeben wird. Um es dennoch zu ermöglichen, Referenzen auf Klassen zu erhalten, die SKilL nicht bekannt sind, gibt es *Custom Fields*. Diese ermöglichen es, in der Spezifikation für eine konkrete Zielsprache zusätzliche Felder anzugeben, deren Typen zur Zeit der Generierung unbekannt sind. Dazu wird in der Spezifikation, wie in Listing 3.3 verdeutlicht, der vollständige Name einer Klasse inklusive Paketnamen angegeben. SKilL erzeugt dann Code der Felder mit diesem Typ enthält. Der generierte Code kompiliert nur, wenn dieser Typ tatsächlich vorhanden ist.

Die Objekte, welche über diesen Mechanismus referenziert werden, sind SKilL unbekannt und können deshalb nicht serialisiert werden. Sie stellen einen Spezialfall der auto-Felder dar und eignen sich vor allem für Daten, die für Berechnungen benötigt, aber nicht serialisiert werden sollen.

3.2.2. Erstellung der reduzierten Darstellung

Das Programm zum Erstellen der reduzierten Zwischendarstellung übersetzt die IML-Darstellung in die in Kapitel 3.1.1 beschriebene Darstellung. Jede Anweisung aus der IML-Darstellung wird dafür in Postorder Reihenfolge traversiert. Beim Aufsteigen im Baum wird ein `Location`-Objekt zurückgegeben, welches den gerade besuchten Teilbaum der Anweisung abbildet.

Abbildung 3.3 zeigt ein Beispiel für eine Anweisung in IML-Darstellung und die dafür generierte, reduzierte Darstellung. Zur Erzeugung des Teilbaums, der den markierten Knoten `Address_Of` repräsentiert, wird entsprechend der Postorder Traversierung zunächst in den darunterliegenden Teilbaum abgestiegen. Die reduzierte Darstellung wird dann beim Aufsteigen im Baum zusammengesetzt. Das Auftreten einer Variablen wird in der reduzierten



(a) IML-Darstellung **(b)** Erzeugte, reduzierte Darstellung

Abbildung 3.3.: Beispiel für die IML-Darstellung der Anweisung `*x = &a`, sowie der erzeugten, reduzierten Darstellung auf Basis der `Location` Klasse.

Darstellung durch eine `UserLocation` dargestellt. Der Knoten `Entity_L_Value` ist für die Analyse unwichtig, weshalb das `UserLocation`-Objekt einfach weitergereicht wird. Für den `Address_Of`-Knoten wird abschließend ein `Address`-Objekt erzeugt, welches das unterhalb im Baum erzeugte `UserLocation`-Objekt referenziert.

SKilL Pools werden verwendet, um Iteratoren für die zu besuchenden Anweisungen zu erhalten. Da das Konstrukt *Anweisung* in der IML nicht existiert, sondern wie jeder Teilausdruck ein `Value` ist, kann dafür nicht direkt ein Iterator für Anweisungen verwendet werden. Beim Iterieren über alle Objekte des Typs `Value` würden viele dieser Objekte mehrfach besucht, da sie selbst `Values` und zusätzlich Kinder anderer `Value`-Knoten sind. Alle Zuweisungen zu besuchen ist ebenfalls nicht zielführend, da auch diese ein Teil anderer Zuweisungen sein können. Außerdem müssen unter anderem ebenfalls Funktionsaufrufe betrachtet werden, welche nicht Teil einer Zuweisung sind.

Um mehrfache Besuche zu verhindern und tatsächlich über alle Anweisungen iterieren zu können, werden die Anweisungsfolgen betrachtet, welche beispielsweise von Funktionen oder Schleifen referenziert werden[2]. Über den Pool des Typs `StatementSequence` kann direkt auf alle Anweisungsfolgen – und über diese auf alle Anweisungen – zugegriffen werden.

3.2.3. Analyse nach Steensgaard

Wie alle anderen Analysen arbeitet die Implementierung der Analyse nach Steensgaard auf der im vorherigen Schritt erzeugten Darstellung. Zusätzlich enthält die verwendete SKilL-Spezifikation für jedes `Location`-Element aus der Zwischendarstellung ein Custom Field

[2]Mit Anweisungsfolgen sind die Blöcke von Anweisungen gemeint, die beispielsweise in C in geschweiften Klammern stehen. Einzeln stehende Anweisungen (zum Beispiel als Teil eines `if`-Statements) werden in der IML auch als `StatementSequence` abgebildet.

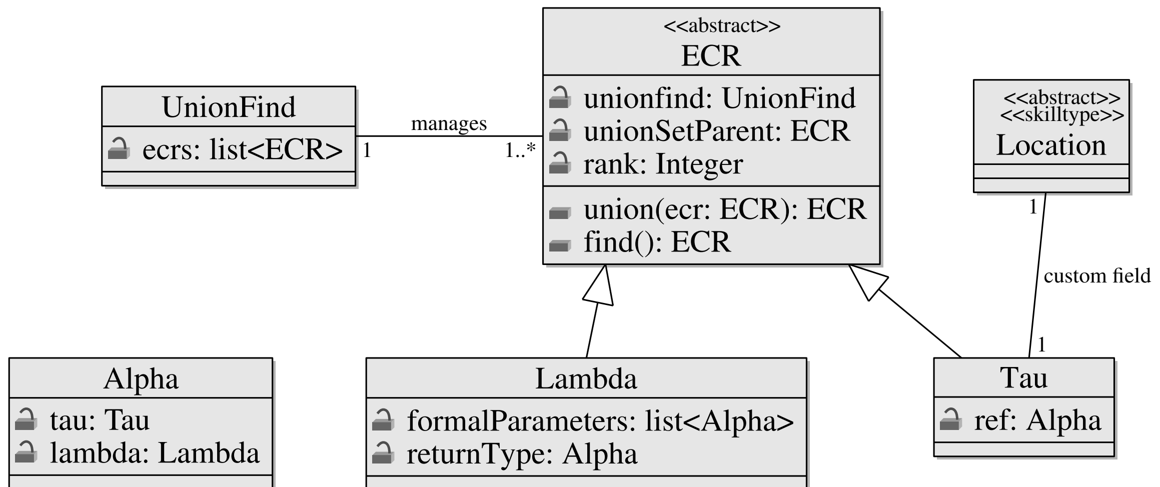


Abbildung 3.4.: Die wichtigsten Klassen in der Implementierung der Analyse nach Steensgaard. Es sind nur die relevanten Felder und Funktionen der Klassen aufgeführt.

über das der τ-Typ, der die Speicherstelle beschreibt, referenziert wird (siehe Abbildung 3.4). Aufgrund der Abwärts- und Aufwärtskompatibilität von SKilL muss das Custom Field in der von den anderen Programmen verwendeten Spezifikation nicht enthalten sein.

Die unterschiedlichen, für Steensgaards Algorithmus notwendigen Typen, sind als jeweils eigene Klassen implementiert. Die τ- und λ-Typen müssen für die Zeigeranalyse mit Union-Find vereinigt werden. Deshalb erben sie von der Equivalence Class Representatives (ECR)-Klasse. Diese stellt ein Element dar, welches mithilfe des Union-Find-Algorithmus (siehe Abschnitt 2.5) verwaltet wird. α-Typen sind Tupel, welche einen τ-Typ und einen λ-Typ beinhalten.

Die Operationen des Union-Find Algorithmus, `union` und `find`, sind als Methoden der ECR-Klasse implementiert. Ebenso sind die von Steensgaard angegebenen Operationen zum Vereinigen von Typen als Methoden der Typ-Klassen umgesetzt. So können die Operationen in ähnlicher Weise zu Steensgaards Pseudocode, ohne zusätzliche Referenz auf ein anderes Objekt, welches die Implementierung der Operationen enthält, aufgerufen werden. Ohne Custom Fields könnten die ECR-Klassen keine Methoden besitzen, da sie dann als SKilL-Typen umgesetzt werden müssten. Nur so könnte der günstige Zugriff auf den Typ einer `Location` über die Referenz ermöglicht werden. Da SKilL-Typen selbst allerdings keine Methoden beinhalten können und von ihnen nicht geerbt werden kann[3], müssten die Operationen an anderer Stelle implementiert werden.

In einem früheren Prototyp, der eine SKilL-Version ohne Custom Fields verwendet, wurde ein Feature von Scala verwendet, um Operationen wie Methoden aufrufen zu können. Dazu können

[3]Skill generiert in Scala *sealed classes*, von denen nicht geerbt werden kann.

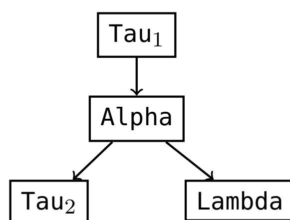


Abbildung 3.5.: Repräsentation von $ref(\perp \times \perp)$ in der Implementierung von Steensgaards Analyse.

in Scala sogenannte *implizite* Klassen definiert werden[4]. Diese sind Wrapper, die Methoden beinhalten, die Objekte des beinhalteten Typs manipulieren. Der Compiler erzeugt diese Wrapper automatisch, wenn eine der Methoden des Wrappers auf einem Objekt aufgerufen wird, das von dem Wrapper aufgenommen werden kann. Dadurch sehen die Funktionsaufrufe aus wie normale Methodenaufrufe. Allerdings ist dadurch nicht direkt ersichtlich wo die Methodendefinition zu finden ist. Andere Features, wie polymorphe Methoden, sind ebenfalls nicht nutzbar. Die Verwendung von Custom Fields ist für Klassen, die nicht serialisiert werden müssen daher vorzuziehen.

Zur Durchführung der Analyse wird zu Beginn für jede `Location` ein Typ-Objekt angelegt. Steensgaard gibt an, dass jede Variable mit dem τ-Typ $ref(\perp \times \perp)$ initialisiert werden soll. Um bei einer Zuweisung zweier Variablen mit den Typen $ref(\perp_1 \times \perp_2)$ und $ref(\perp_3 \times \perp_4)$ die entsprechende Vereinigung von Typen durchzuführen, müssen jeweils $\perp_1$ und $\perp_3$, sowie $\perp_2$ und $\perp_4$ zusammengefasst werden. Da in der Implementierung zum Zeitpunkt des Zusammenfassens beider Typen, wie auch in Steensgaards Pseudocode, keine Referenz auf den Elterntyp gegeben ist, muss jedes $\perp$ als ein ECR-Objekt modelliert werden. Das bedeutet, dass ein $ref(\perp \times \perp)$ durch ein `Tau`-Objekt modelliert wird, das auf ein `Alpha`-Objekt verweist, welches wiederum ein `Tau` und ein `Lambda` enthält (siehe Abbildung 3.5).

Bei der Allokation eines ECR-Objektes (also auch von `Tau` und `Lambda` Objekten) wird dieses in eine Liste eingefügt, in der alle durch Union-Find verwalteten ECR-Objekte enthalten sind. Dieser Schritt entspricht der Union-Find Operation `makeset`.

Darauffolgend werden die relevanten Objekte aus der Zwischenstruktur über die jeweiligen Pools abgerufen und deren Typen entsprechend der von Steensgaard angegebenen Regeln verarbeitet. Dabei wird für Tau-Typen auch die Pending-Lists Optimierung umgesetzt.

Am Ende werden für alle, durch die ECR-Objekte repräsentierten Mengen, die beinhalteten `Locations` gesammelt. Die enthaltenen Points-to-Beziehungen werden in die einheitliche `PointsToSet` Darstellung der `Locations` umgesetzt.

[4]siehe die Scala Dokumentation unter: http://docs.scala-lang.org/overviews/core/implicit-classes.html

3.2.4. Analyse nach Das

Der Prototyp der Analyse nach Das basiert auf der Implementierung von Steensgaards Analyse. Dabei ist das Klassendiagramm aus Abbildung 3.4 weiterhin gültig. Es werden außerdem keine zusätzlichen Funktionen von SKilL genutzt.

Die Erweiterung besteht darin, dass Zuweisungen keine Vereinigung der Typen auf oberster Ebene veranlassen, sondern lediglich eine Flowkante eingefügt wird. Diese Kanten sind als Liste von Referenzen auf die Ziele der Flowkanten in jedem `Tau`- und `Lambda`-Objekt implementiert. Zusätzlich zum Einfügen der Flowkante muss sichergestellt werden, dass für die Typen auf tieferer Ebene die Vereinigung wie in Steensgaards Algorithmus durchgeführt wird. Dabei kann es vorkommen, dass ein Typ, der mit einem anderen vereinigt werden soll, noch gar nicht existiert.

Angenommen zwei Variablen `x` und `y` besitzen den initialen Typ, wie in Abbildung 3.5 dargestellt, dann wird für eine Zuweisung `x = y` eine Flowkante zwischen den beiden $\texttt{Tau}_2$-Objekten eingefügt. Diese besitzen beide keine initialisierte Typkomponente, weshalb ein neues Typobjekt erzeugt und in den beiden $\texttt{Tau}_2$-Objekten als Typkomponente referenziert wird.

Da Pending-Lists von Das nicht beschrieben werden, sind sie im Prototyp nicht enthalten. Die Typen auf oberster Ebene werden allerdings nicht vereinigt, wodurch einige der Ungenauigkeiten, die Pending-Lists auflösen, erst gar nicht auftreten. Insbesondere in dem von Steensgaard angegebenen Beispiel in Abbildung 2.6 würden bei Das' Analyse `x` und `y` nicht durch denselben Typ beschrieben werden, sondern es würde lediglich eine Flowkante von `a` nach `x` beziehungsweise `y` eingeführt. Einzig die über mehr als eine Indirektion referenzierten Ziele werden gleich behandelt.

Flowkanten werden in der Implementierung nur für die Tau-Typen erzeugt. Lambda-Typen werden nicht wie in Kapitel 2.7.2 beschrieben behandelt, sondern wie bei Steensgaards Analyse direkt vereinigt. Auf diese Weise können auch Lambda-Typen mit unterschiedlich vielen Parametertypen einander zugewiesen werden, wobei dynamisch in dem Typ, der weniger Parameter besitzt, die Parametertypen des anderen Typs eingefügt werden.

Dies ist notwendig, da durch Typumwandlungen Zuweisungen möglich sind, wie sie in Abbildung 3.4 dargestellt sind. Eine korrekte Behandlung dieses Beispiels erfordert die Beschreibung beider Funktionszeiger mit zwei Parametertypen ($lam(\alpha_1, \alpha_2) \rightarrow \alpha_3$).

Zu Beginn der Analyse ist unbekannt, wieviele Parameter durch einen Lambda-Typ beschrieben werden müssen. Diese müssen also bei Bedarf erzeugt werden. Wenn bei einer Zuweisung eine Flowkante zwischen zwei Lambda-Typen eingeführt wird, bedeutet dies, dass die Parametertypen vereinigt werden müssen. Angenommen beide Lambda-Typen beinhalten zu diesem Zeitpunkt keine Parametertypen und wird anschließend durch eine weitere Zuweisung dem einen der beiden Lambdas ein Parametertyp hinzugefügt, muss dieser auch dem anderen hinzugefügt werden. Beide sind jedoch unabhängig und es müssten zu diesem Zweck zusätzliche Referenzen zwischen den Lambdas eingeführt werden. Werden die Lambdas jedoch vereinigt,

Listing 3.4 Beispiel für Zuweisung zwischen Funktionszeigern mit verschiedenen Signaturen.

```
1  int* funktion1(int* param) { ... }
2  int* funktion2(int* test, int* zwei) { ... }
3
4  int main(int argc, char *argv[]) {
5        int integer = 5;
6
7        int* (*fp1)(int*) = &funktion1;
8        int* (*fp2)(int*, int*) = &funktion2;
9
10       fp2 = (int* (*) (int*, int*)) fp1;
11       int* pointer = ((int* (*) (int*))fp2)(&integer);
12 }
```

wirkt sich das Einfügen eines neuen Parametertyps automatisch auf beide Typen (die jetzt einen Typ darstellen) aus.

Der Präzisionsverlust betrifft nur die Funktionsziele der Zeiger, jedoch nicht die Parameterzuweisungen, welche durch indirekte Aufrufe entstehen, da die Parametertypen in jedem Fall vereinigt werden.

Anschließend an die Erstellung der Flowkanten und der Vereinigung der darunterliegenden Typen, müssen die Flowkanten aufgelöst werden. Dazu werden, wie bei der Implementierung von Steensgaards Analyse, die Repräsentanten der Typmengen zusammengetragen. Dabei werden auch alle, durch diese Typen beschrieben `Location`-Objekte gesammelt. Die von einem Typen ausgehenden Flowkanten werden schon bei der Vereinigung von Typen immer dem aktuellen Repräsentaten der Typmenge zugewiesen, sodass die Flowkanten in der Liste des Repräsentaten vollständig sind. Die von einer Typmenge beschriebenen Speicherstellen werden jeder Typmenge hinzugefügt, die über die Flowkanten erreichbar ist.

Die Übersetzung der Typmengen in die analyseübergreifende Darstellung erfolgt anschließend analog zur Umsetzung für Steensgaards Analyse.

3.2.5. Analyse nach Andersen

Der Prototyp von Andersens Analyse arbeitet in zwei Phasen. In der ersten Phase werden aus den Anweisungen des Programms Lösungsbedingungen aufgestellt, welche in der zweiten Phase gelöst werden. Zu deren Aufstellung werden wie für die Implementierungen der anderen Analysen die SKilL-Pools verwendet, um die jeweils relevanten `Location`-Objekte abzurufen.

Die Darstellung der Lösungsbedingungen basiert auf den *auto*-Feldern von SKilL, die in Listing 3.5 dargestellt sind. Dazu stellen die `Location`-Objekte Knoten in einem gerichteten Graphen dar. Lösungsbedingungen werden durch Kanten zwischen diesen Knoten abgebildet. Typ 1 Kanten werden direkt durch das vorhandene `PointsToSet` abgebildet. Die Ergebenisse der Analyse liegen so ohne zusätzlichen Konvertierungsschritt in der einheitlichen Datenstruktur

Listing 3.5 Teil der SKilL-Spezifikation, welche für die Implementierung von Andersens Analyse genutzt wird.

```
Location {
IMLRoot astNode;
PointsToSet pointsToSet;

auto set<Location> subset;
auto set<Location> derefSubset;
auto set<Location> subsetDeref;

auto set<IndirectRoutineCall> indirectCalls;
}
```

vor. Eine Referenz in der `subset`-Menge (Listing 3.5, Zeile 5) stellt eine Typ 2 Kante dar. Die Mengen `derefSubset`, sowie `subsetDeref` repräsentieren die Typ 3 und Typ 4 Kanten. Außer den Typ 1 Kanten sind alle Kanten lediglich für die Berechnung und nicht für das Ergebnis relevant. Da sie als auto-Felder implementiert sind, werden sie nicht serialisiert und belegen keinen zusätzlichen Speicherplatz.

Die indirekten Aufrufe über einen Funktionszeiger werden in einem initialen Pass zusätzlich im `Location`-Objekt des Funktionszeiger gesammelt. Dies ermöglicht die Erzeugung neuer Lösungsbedingungen für Parameterzuweisungen, sobald neue Funktionsziele in die Zeigerzielmenge aufgenommen werden.

Zum Lösen der Lösungsbedingungen wird der Basisalgorithmus aus der Veröffentlichung von Hardekopf und Lin verwendet [HL07]. Die vorgeschlagene Zyklenerkennung ist nicht implementiert.

Der Algorithmus verwaltet eine Liste mit Knoten, die noch betrachtet werden müssen. Zu Beginn des Algorithmus wird diese mit allen Knoten initialisiert. Solange die Liste nicht leer ist, wird ein Knoten von der Liste genommen und dessen Lösungsbedingungen bearbeitet. Für Typ 3 und Typ 4 Bedingungen werden entsprechende Typ 2 Bedingungen eingeführt, und für Typ 2 Bedingungen die Zielzeigermengen aktualisiert. Dabei werden die Knoten, deren Zielmengen verändert wurden, erneut der Liste hinzugefügt, da sich dadurch erneute Änderungen ergeben können.

Sobald die Liste leer ist, können die Typ 2 bis 4 Kanten verworfen und das Ergebnis serialisiert werden.

3.3. Schwierigkeiten bei der Implementierung

Die Prototypen sind nicht als einsatzfähige Zeigeranalysen gedacht, da beispielsweise die Effekte von *eingebauten* Funktionen, deren Code nicht verfügbar ist, nicht behandelt werden. Außerdem existieren noch einige Optimierungen, durch die insbesondere Andersens Analyse

Listing 3.6 Code-Beispiel für die Verwendung von variablen Argumentenlisten.

```
1  #include <stdlib.h>
2  #include <stdarg.h>
3  #include <stdio.h>
4
5  int *pointer;
6
7  void test(int count, ...) {
8     va_list ap;
9     va_start(ap, count);
10    for(;count>0;count--) {
11       pointer = va_arg(ap, int*);
12       printf("%d\n", *pointer);
13    }
14    va_end(ap);
15 }
16
17 int main(int argc, char *argv[])
18 {
19    int a = 2
20       ,b = 4
21       ,c = 6;
22    test(3, &a, &b, &c);
23 }
```

besser auf größere Programme skalieren würde. Dazu können beispielsweise Zyklen im Graph der Lösungsbedingungen erkannt und zusammengefasst werden [HL07].

Neben diesen Limitierungen sind einige andere Komplikationen während der Implementierung aufgetreten. Diese sind nicht auf die Verwendung von SKilL zurückzuführen.

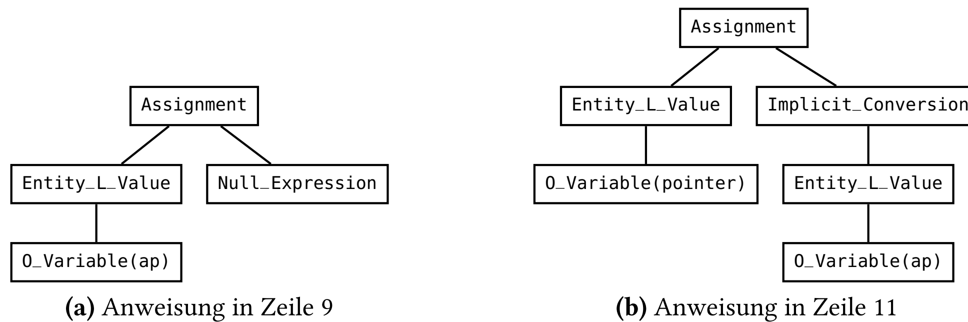
3.3.1. Variable Argumentenlisten

Variable Argumentenlisten werden von den Prototypen nicht behandelt, da hierfür keine sinnvolle Möglichkeit gefunden wurde.

Variable Argumentenlisten erlauben es einer Funktion, eine variable Anzahl an Parametern entgegenzunehmen. Ein Beispiel für eine Funktion, die dieses Feature nutzt, ist `printf`. Sie ermöglicht es einen Formatstring anzugeben, welcher Platzhalten für Variablenwerte enthält. Durch die Übergabe einer beliebigen Anzahl an Parametern können diese Platzhalter ausgefüllt werden. Ein Aufruf kann folgendermaßen aussehen:

`printf(“Zahl1: %d, Zahl2: %d“, int1, int2);`

Dabei werden von der Funktion soviele Parameter erwartet, wie Platzhalter (zum Beispiel `%d`) vorhanden sind.



(a) Anweisung in Zeile 9

(b) Anweisung in Zeile 11

Abbildung 3.6.: Erzeugte IML-Darstellung für die Makroinvokationen in Listing 3.6.

Listing 3.7 Initialisierung eines Arrays von Struct-Objekten.

```
1  typedef struct test {
2  int value;
3  int* pointer;
4  void (*fp)(int*);
5  } test;
6
7  int main(int argc, char *argv[])
8  {
9  int integer;
10 test obj[] = {{4, &integer,},{5, &integer}};
11 }
```

Listing 3.6 zeigt ein anderes Beispiel für eine solche Funktion. Hier wird die Anzahl der erwarteten Parameter explizit angegeben. Der Zugriff auf die Parameterliste erfolgt über die Makros `va_start`, `va_arg` und `va_end`. Der von Bauhaus für diese Makros generierte Code, stellt allerdings keinerlei Beziehung zwischen den Parametern und der Variablen `ap` her, welche den Zugriff auf die Argumente ermöglicht. Für den Aufruf der Makros `va_start` in Zeile 9 und `va_arg` in Zeile 11 wird die IML-Darstellung in Abbildung 3.6 aufgezeigt.

Die Initialisierung der `ap`-Variablen mit dem ersten Parameter in Zeile 9 wird in eine `Null_Expression` übersetzt. Die einzige Möglichkeit die wegfallenden Informationen zurückzugewinnen wäre es, die Variablen mit dem Typ `va_list` zu finden, und dieser alle Parameter der Aufrufseiten gleichzusetzen. Dies ist allerdings nicht in allen Fällen korrekt, da die Variable eventuell gar nicht, oder für einen anderen Zweck verwendet wird.

3.3.2. Struct Aggregates

In der IML werden Initialisierungswerte von Structs und Arrays durch `Aggregates` dargestellt. Ein Beispiel ist die Liste `{1,2,3}` in der Initialisierungsanweisung `int a[] = {1,2,3}`. Im Fall von Structs wird eine solche einfache Initialisierung durch Zuweisungen der Werte an die Felder des Structobjekts abgebildet.

Listing 3.8 Felder des Structs `table_ty` in der Datei *sub.c* des Programms *cook*[5].

```
1  option:              int
2  set:                 char*
3  resubstitute:        int
4  must_be_used:        int
5  append_if_unused:    int
6  override:            int
7  name:                char*
8  opposite:            enum(flag_value_ty)
9  value:               int
10 func:                fp (typedef für einen Funktionszeiger)
11 unset:               char*
```

Wird allerdings ein Array aus Structs initialisiert, wie in Listing 3.7 gezeigt, sind die Struct-Objekte nicht bekannt und für die Initialisierung der Structs werden ebenfalls `Aggregates` erzeugt. Dadurch geht die Information verloren, welchen Feldern des Structs welche Werte zugewiesen werden. Zwar existiert die Typinformation des Structs, allerdings entspricht die Reihenfolge der dort angegebenen Felder nicht der Reihenfolge der Werte in den `Aggregates`. Zusätzlich ist es möglich, dass nicht alle Felder des Structs initialisiert werden, wie in Listing 3.7 dargestellt. In einem Beispiel sind die Felder des Structs wie in Listing 3.8 beschrieben. Die zugewiesenen `Aggregates` beinhalten nur zwei Werte mit den Typen `const char*` und `fp`. Offensichtlich entsprechen diese Typen nicht denen der beiden ersten Felder des Structs.

Da die Felder in einigen Beispielen hinsichtlich der Reihenfolge mit den Initialisierungswerten übereinstimmen, werden trotzdem Zuweisung zwischen den Feldern und Initialisierungswerten mit gleichem Index in die Zwischenstruktur eingefügt. In den fehlerhaften Fällen werden so allerdings fehlerhafte Zeigerzielmengen erzeugt. Da die Darstellung von allen Analysen verwendet wird, bleiben die Ergebnisse der Analysen untereinander vergleichbar.

3.3.3. Beschreibung der Algorithmen

Die Implementierung der Prototypen wurde insbesondere dadurch erschwert, dass zu keiner Analyse eine veröffentlichte Referenzimplementierung existiert. Zusätzlich wird ein Teil der Analysen von den Autoren nicht vollständig beschrieben. Beispielsweise beschreibt Das nicht, wie mit Funktionszeigern umgegangen werden soll und Andersen gibt nur einen sehr primitiven Algorithmus zur Lösung seines Problems an. Die vielen anderen Veröffentlichungen zu Andersens Analyse verwenden zusätzlich Optimierungen, welche die asymptotische Worst-Case Laufzeit nicht verbessern, aber in der Praxis zu deutlich kürzeren Laufzeiten führen. Diese sind jedoch oft komplex zu implementieren und für einen Prototyp nicht notwendig.

[5]https://answers.launchpad.net/ubuntu/+source/cook/2.26-1

4. Evaluation

In diesem Kapitel wird betrachtet, ob die Prototypenentwicklung mit SKilL effizient möglich ist. Dazu wird zunächst die während der Entwicklung verwendete Testmethodik erläutert und anschließend die Plausibilität der Ergebnisse sowie das Laufzeitverhalten der Prototypen für reale Programme untersucht. Abschließend werden die Funktionen und Eigenschaften von SKilL betrachtet, welche für die Prototypenentwicklung hilfreich oder hinderlich sind.

4.1. Testplattform

Sämtliche Zeitmessungen im folgenden Kapitel werden unter Ubuntu 16.04 auf einem Desktop Rechner mit einem Intel Core i7-2700K Prozessor ausgeführt. Dieser verfügt über 4 Kerne (8 Threads) welche dynamisch mit 3,5 bis 3,9 GHz getaktet sind. Als Hauptspeicher stehen 16 GB DDR3 Speicher zur Verfügung, die mit 1333 MHz betrieben werden. Serialisierte Daten werden von einer, über SATA-3 angebundenen, 128 GB großen Crucial M4 SSD gelesen.

4.2. Automatisiertes Testen

In diesem Abschnitt wird erläutert, wie die implementierten Prototypen automatisiert getestet werden. Zunächst wird jedoch eine kurze Motivation für das Testen von Software im Allgemeinen sowie speziell für das automatisierte Testen gegeben.

4.2.1. Motivation für automatisiertes Testen

Software kann auf unterschiedlichen Ebenen getestet werden [MSB11]. Zum einen können einzelne Module mit Modultests in Isolation vom Rest des Programmes auf korrekte Funktion geprüft werden. Zum anderen kann das Programm als Ganzes getestet werden. Dabei kann die Funktionalität der Anwendung an sich überprüft werden. Zusätzlich können aber auch nicht-funktionale Anforderungen, wie Performance oder Usability, untersucht werden.

Da bei der Entwicklung von Software oftmals Fehler entstehen, ist es empfehlenswert, ein Programm zumindest funktional zu testen. Ein gelegentliches, manuelles „Ausprobieren“ des Programms während der Entwicklung kann bereits als funktionaler Test angesehen werden.

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
	int i, j, k;
	int *a = &i;
	int *b = &k;
	a = &j;
	int **p = &a;
	int **q = &b;
	q = p;
	int *c = *q;
}
```

(a)

```
digraph pointsTo {
"a" -> "i"
"a" -> "j"
"b" -> "k"
"p" -> "a"
"q" -> "b"
"q" -> "a"
"c" -> "i"
"c" -> "j"
"c" -> "k"
}
```

(b)

Abbildung 4.1.: Beispiel für einen Testfall als kleines C-Programm in (a), sowie dem erwarteten Ergebnis (für Andersens Analyse) in (b).

Das Problem dabei ist, dass meist nur eine spezielle Funktionalität betrachtet wird und zuvor durchgeführte Testabläufe nicht erneut durchlaufen werden. Durch weitere Entwicklungsarbeiten können allerdings Fehler in bereits funktionierenden Teilen des Programms entstehen. Es empfiehlt sich deshalb, sämtliche bereits durchgeführten Testabläufe zu sammeln, damit diese bei Bedarf wieder benutzt werden können.

Die Größe der so entstehenden Testsammlung macht es aufwendig alle Tests von Hand auszuführen. Durch die Automatisierung der Tests kann die Ausführungszeit deutlich reduziert werden. Dies ermöglicht die Ausführung aller Tests nach jeder größeren Änderung des Programms und erhöht dadurch die Wahrscheinlichkeit neu entstandene Fehler zu finden.

Für jede verbreitete Programmiersprache existieren Bibliotheken, die das Automatisieren von Tests vereinfachen. Diese bieten in der Regel eine Möglichkeit Testcode anzugeben, der beispielsweise Funktionen aus der zu testenden Software aufruft. Die Ergebnisse eines solchen Funktionsaufrufs können dann über die von der Testbibliothek zur Verfügung gestellten Operationen mit dem erwarteten Ergebnis verglichen werden. Für das Testen der in Scala implementierten Prototypen wird das Test-Tool ScalaTest[1] verwendet.

4.2.2. Implementierung der Tests für die Prototypen

Zum Testen der Prototypen werden einige Modultests implementiert, die beispielsweise die Union-Find Datenstruktur aus Steensgaards Analyse in Isolation vom Rest des Programms testen. Hauptsächlich werden jedoch funktionale Tests verwendet, die das Ergebnis der Analyse

[1] http://www.scalatest.org/

überprüfen. Dazu können C-Programme angegeben werden, welche durch den Bauhaus-Compiler in IML übersetzt und anschließend mithilfe des iml2sf Tools in die SKilL-Darstellung überführt werden. Das Testprogramm führt die Analyse auf diesen Programmen aus und vergleicht das Ergebnis mit dem erwarteten Ergebnis.

Dieses erwartete Ergebnis wird in einem textbasierten Format angegeben (siehe Abbildung 4.1b). Zum Vergleich der Erwartung mit dem tatsächlichen Resultat der Analyse, wird dieses ebenfalls in das Format gebracht. Durch die Annahme, dass in jeder Zeile (erste und letzte Zeile ausgenommen) jeweils eine Points-to-Beziehung dargestellt ist, können die Ergebnisse auf String-Ebene verglichen werden. Dazu wird die Datei mit den erwarteten Ergebnissen eingelesen und in n Zeilen zerlegt. Die Zeilen 2 bis $n-1$ werden alphabetisch sortiert und paarweise mit den sortierten Ergebnissen der Analyse verglichen.

Es wird für all C-Programme, die in einem bestimmten Ordner liegen, automatisch ein Testfall generiert. Die Datei mit den erwarteten Ergebnissen wird in einem anderen Ordner unter der Konvention erwartet, dass der Dateiname dem des C-Programms entspricht, die Dateiendung jedoch *.res* ist. Liegt zu einem Programm kein erwartetes Ergebnis vor, wird der Test als *pending* markiert. So kann während der Implementierung zwischen fehlerhaften und noch nicht implementierten Funktionen unterschieden werden.

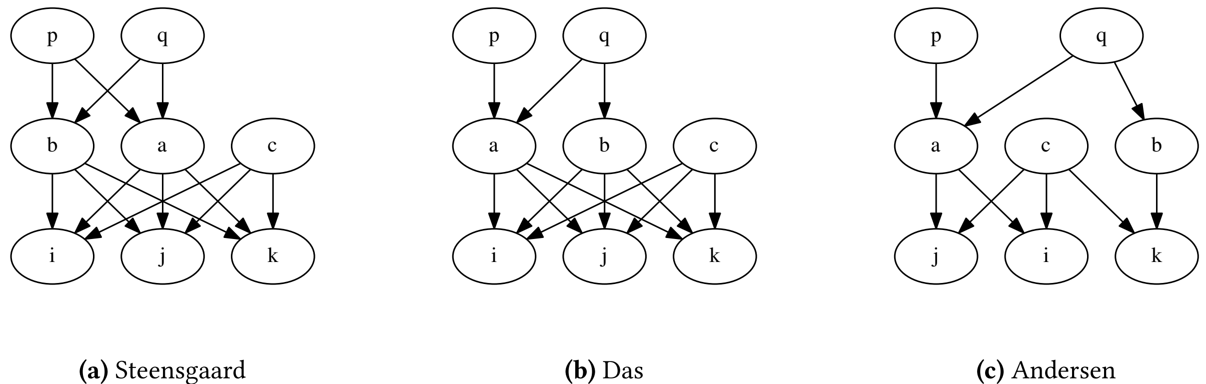
Das Format in dem die erwarteten Ergebnisse angegeben werden, ist ein Subset der DOT-Sprache von Graphviz[2]. Damit lassen sich die angegebenen Points-to-Beziehungen auch mit Graphviz visualisieren. Abbildung 4.2 zeigt diese Graphen zu den Ergebnissen aller drei Analysen für das Beispiel in Abbildung 4.1a. Die Darstellung als Graph erleichtert die Fehlerfindung, für den Fall, dass tatsächliche und erwartete Ergebnisse voneinander abweichen.

Die vorhandene Testsuite besteht aus ca. 50 kleinen C-Programmen, die jeweils verschiedene Sprachfeatures von C abdecken. Dazu gehören unter anderem die Verwendung von Structs, Arrays und Funktionszeigern. Ein kompletter Testlauf benötigt weniger als 2 Sekunden, womit die Tests auch nach kleineren Änderungen ausgeführt werden können, ohne den Entwickler maßgeblich zu beeinträchtigen.

4.3. Analyse von realen Programmen

Neben den Tests mit kleinen Programmfragmenten sollen auch reale Programme analysiert werden. Eine Auswahl von 49 bereits übersetzten und in SKilL-Darstellung vorliegenden Programmen wurde dafür zur Verfügung gestellt (siehe Anhang A.1). Zwei dieser Programme (*bash* und *gnugo*) wurden nicht analysiert, da die Laufzeiten für Andersens Analyse mit über einer Stunde zu lang sind. Für alle anderen Beispiele konnten die Analysen ausgeführt werden.

[2]http://www.graphviz.org/content/dot-language



(a) Steensgaard **(b)** Das **(c)** Andersen

Abbildung 4.2.: Visuelle Darstellung der erwarteten Ergebnisse für das Beispiel in Abbildung 4.1a. Erzeugt mit dem in Graphviz enthaltenen Tool *dot* (`$ dot -Tpdf erwartetesErgebnis.txt > graph.pdf`).

Im nächsten Abschnitt wird auf die Plausibilität der Analyseresultate eingegangen. Anschließend wird die Präzision der Ergebnisse und das Laufzeitverhalten der Prototypen anhand der Beispiele betrachtet.

4.3.1. Plausibilität der Analyseergebnisse

Für die kleinen Programmfragmente, die für das automatische Testen verwendet werden, kann von Hand überprüft werden, ob diese mit den Erwartungen übereinstimmen. In realen, größeren Programmen sind die Datenflüsse jedoch deutlich komplexer und die Anzahl an Zeigervariablen sehr groß, sodass die Analyseergebnisse für diese Programme nicht mehr sinnvoll manuell überprüft werden können.

Anhand einiger Bedingungen die zwischen den Ergebnissen der einzelnen Analysen gelten müssen, können diese allerdings auf Plausibilität überprüft werden. Insbesondere müssen die Zeigerzielmengen Teilmengenbeziehungen erfüllen. Die Ergebnisse der Analyse nach Andersen müssen in den Ergebnissen der beiden anderen Analysen enthalten sein. Generell gilt ebenso, dass die Ergebnisse von Das' Analyse Teilmenge der Ergebnisse von Steensgaards Analyse sein müssen. Da für Steensgaards Analyse jedoch, im Gegensatz zu Das' Analyse, die Pending-List Optimierung implementiert wurde, sind hier Abweichungen zu erwarten. Die Teilmengenbeziehungen können programmatisch leicht überprüft werden, da alle Analysen auf derselben Darstellung des Programms arbeiten.

Abbildung 4.3 zeigt, dass die durchschnittlichen Größen der Zeigerzielmengen tatsächlich für alle Beispiele in der erwarteten Relation stehen. Zusätzlich wurde ein Werkzeug entwickelt, dass konkret für jede Zielmenge überprüft, ob sie die Teilmengenbeziehungen erfüllt.

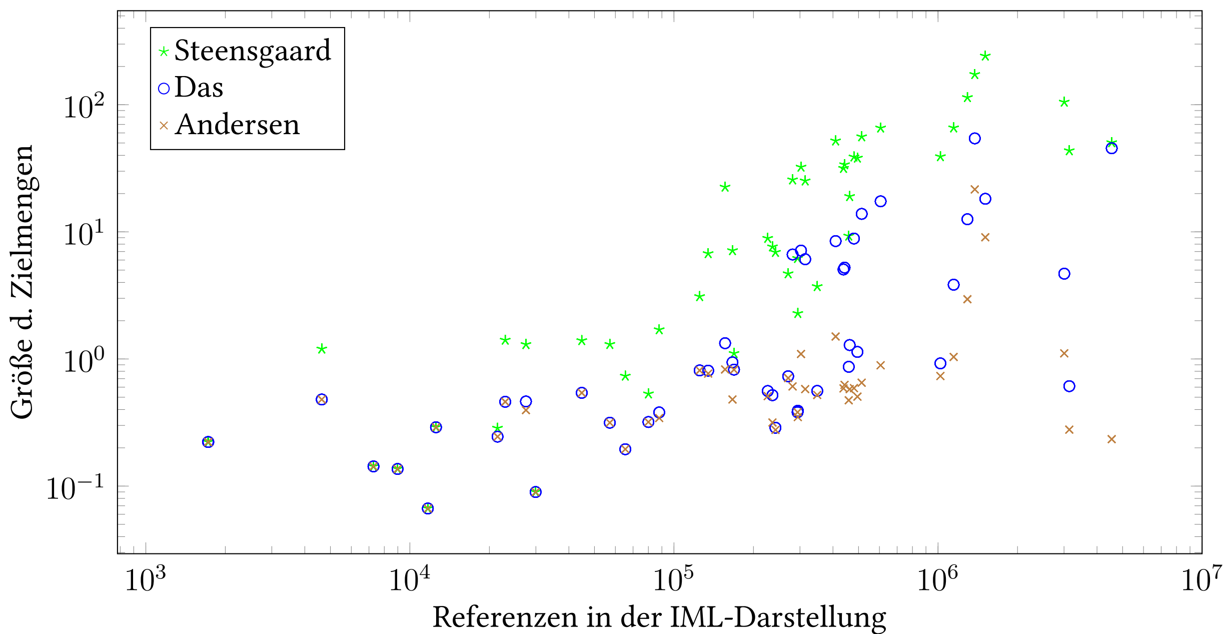


Abbildung 4.3.: Durchschnittliche Größe der Zeigerzielmengen für alle `UserLocation`- und `Allocation`-Objekte (lokale und globale Variablen, Parameter, sowie dynamisch allozierte Objekte auf dem Heap). Eine vollständige, den Programmen zugeordnete, Auflistung der Größen ist in Anhang A.2 zu finden.

Es vergleicht die in den Zeigerzielmengen enthaltenen Objekte anhand der intern von SKilL verwendeten `skillID`. Dieser Vergleich basiert auf der Annahme, dass die Objekte der reduzierten Programmdarstellung für alle Analysen dieselbe `skillID` besitzen. Die Analysen müssen für den Vergleich deshalb auf der gleichen Datenbasis ausgeführt werden.

Der Vergleich der Zeigerzielmengen ergibt, dass die Teilmengenbeziehungen zwischen Andersen und Das, sowie Andersen und Steensgaard, für alle getesteten Programme bis auf *trueprint* erfüllt sind. Für dieses Programm weichen die Ergebnisse für zwei Variablen ab. Der Grund dafür konnte nicht identifiziert werden.

Die Ergebnisse der Analysen erscheinen abgesehen davon untereinander konsistent und plausibel. Trotzdem werden wegen der in Kapitel 3.3 dargelegten Probleme keine echten, konservativen Abschätzungen erzeugt.

4.3.2. Präzision der Analysen

Abbildung 4.3 zeigt, dass mit Steensgaards Analyse deutlich schlechtere Ergebnisse erzielt werden, als mit den anderen beiden. Die Ergebnisse von Das' Analyse reichen oft nah an die von Andersens Analyse heran oder gleichen diesen sogar. Diese Beobachtung deckt sich mit den von Das selbst beschriebenen [Das00]. Dies ist der Fall, obwohl im hier entwickelten

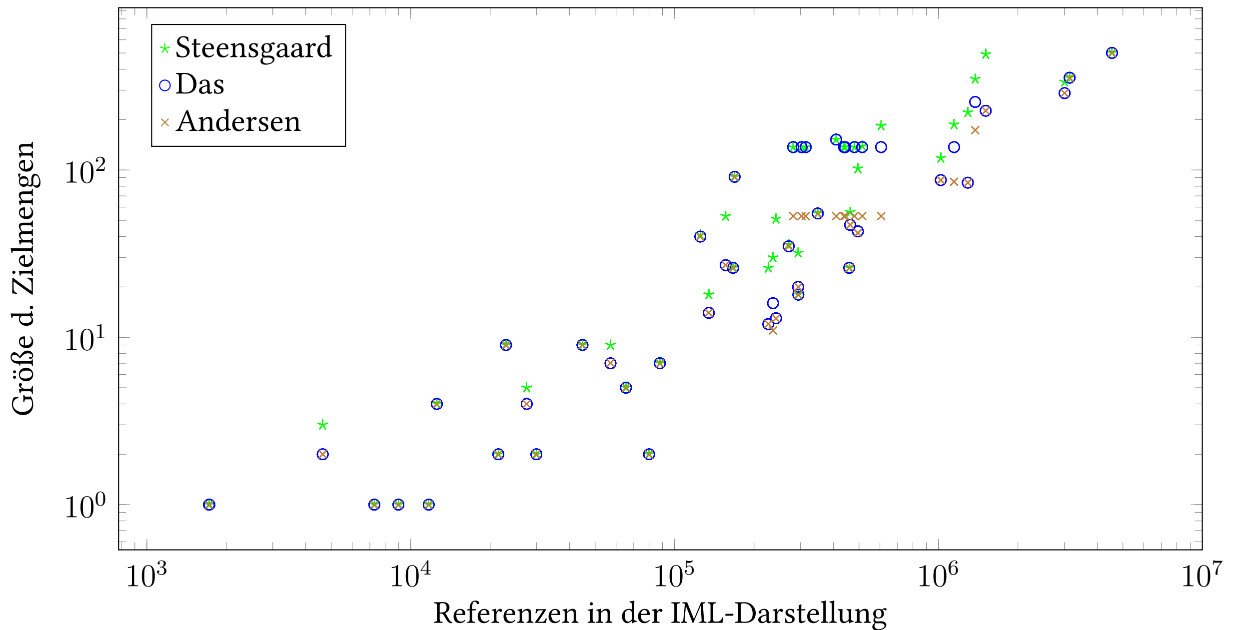


Abbildung 4.4.: Maximale Größe der Zeigerzielmengen für alle `UserLocation`- und `Allocation`-Objekte (lokale und globale Variablen, Parameter, sowie dynamisch allozierte Objekte auf dem Heap).

Prototyp von Das' Analyse keine Pending-Lists implementiert sind und die Funktionszeiger nicht über den One-Level-Flow Algorithmus behandelt werden (siehe Kapitel 3.2.4).

Während die durchschnittliche Größe der Zeigerzielmengen durch die genaueren Analysen deutlich verkleinert werden kann, fällt auf, dass die Anzahl der Elemente in der größten Zielmenge weniger stark sinkt (siehe Abbildung 4.4). Eine mögliche Erklärung hierfür ist die Kontextinsensitivität der Analysen. In den meisten Programmen existieren Funktionen, die an vielen unterschiedlichen Stellen aufgerufen werden. Dies können beispielsweise Factory-Funktionen sein, die verwendet werden um Objekte auf dem Heap zu allozieren. Für alle Analysen entsteht ein gleiches Maß an Ungenauigkeit durch den Zusammenfluss der verschiedenen Aufrufe.

4.3.3. Laufzeitverhalten der Prototypen

Um das Laufzeitverhalten der Prototypen zu analysieren wird jede Analyse für alle Testprogramme zehnmal ausgeführt. Vor jedem Durchlauf werden die zu analysierenden Dateien gegen Dateien ausgetauscht, die noch keine Zeigerinformationen enthalten. Das ist insbesondere für Andersens Analyse wichtig, die direkt auf den `PointsToSets` operiert. Sind diese schon mit den korrekten Ergebnissen initialisiert, werden deutlich weniger Iterationen benötigt und die Laufzeit verkürzt sich stark.

Die Ausführungszeiten werden mit Bashs `time`-Befehl gemessen. Da alle Zeiten im Sekundenbereich liegen, können Messungenauigkeiten vernachlässigt werden. Als Messwert wird die *real* Zeit verwendet. Abweichungen zwischen den verschiedenen Messläufen können nur für die längeren Ausführungszeiten beobachtet werden und liegen innerhalb der Erwartungen. Sie sind auf Unterbrechungen durch das System, sowie Schwankungen in der Leistung zurückzuführen, welche beispielsweise durch die dynamische Anpassung der Prozessorfrequenz bedingt werden.

Abbildung 4.5 zeigt die Ausführungszeit in Abhängigkeit zur Anzahl der `Location`-Objekte. Um zu verdeutlichen, wie die unterschiedlichen Analysen skalieren, wurde eine lineare Darstellung gewählt. Dadurch sind die Ergebnisse für die kleineren Programme schlecht zu unterscheiden. In Anhang A.4 findet sich zusätzlich eine logarithmische Visualisierung, sowie Darstellungen in Abhängigkeit anderer Werte.

In Abbildung 4.5 wird deutlich, dass die Analyse nach Steensgaard für die Beispiele linear mit der Anzahl der Knoten skaliert. Zudem zeigen die Messungen, dass Das' Analyse in der Praxis ebenfalls linear skaliert und nur unwesentlich teurer ist als Steensgaards Analyse, obwohl deutlich genauere Ergebnisse erreicht werden. Bei den Ausführungszeiten des Prototyps für Andersens Analyse kann beobachtet werden, dass diese Analyse ohne zusätzliche Optimierungen, wie beispielsweise Zyklen-Eliminierung, deutlich schlechter skaliert. Es lässt sich mit den vorhandenen Daten nicht ermitteln, ob die Laufzeit in der Praxis quadratisch oder kubisch ansteigt, da einige starke Ausreißer vorhanden sind. Diese sind darauf zurückzuführen, dass sich nicht allein die Größe des Programms auf die Laufzeit auswirkt, sondern ebenso die Art wie Zeiger verwendet werden.

4.4. Umsetzung der Analysen mit SKilL

Es ist schwierig die Prototypenentwicklungseigenschaften von SKilL im Bauhaus-Kontext zu bewerten, da kein direkter Vergleich zur Prototypenentwicklung ohne SKilL gezogen werden kann. Dies wurde in dieser Arbeit nicht berücksichtigt und es existieren auch keine anderen Arbeiten, die sich mit der Prototypenentwicklung in Bauhaus beschäftigen. Es kann jedoch festgestellt werden, dass es mithilfe von SKilL möglich war, innerhalb kurzer Zeit Prototypen zu drei verschiedenen Zeigeranalysen zu entwickeln. Die dabei aufgetretenen Hindernisse betreffen hauptsächlich die Implementierung der Analysen an sich und in seltenen Fällen die IML-Darstellung von Bauhaus (siehe Kapitel 3.3). Im Folgenden wird genauer auf die Nützlichkeit der SKilL-Funktionen für die Prototypenentwicklung eingegangen.

Sprachunabhängigkeit

SKilL erlaubt es den zur Serialisierung benötigten Code für verschiedene Sprachen zu generieren. Dies ermöglicht es, Prototypen in einer Sprache zu entwickeln, die dem Entwickler bereits

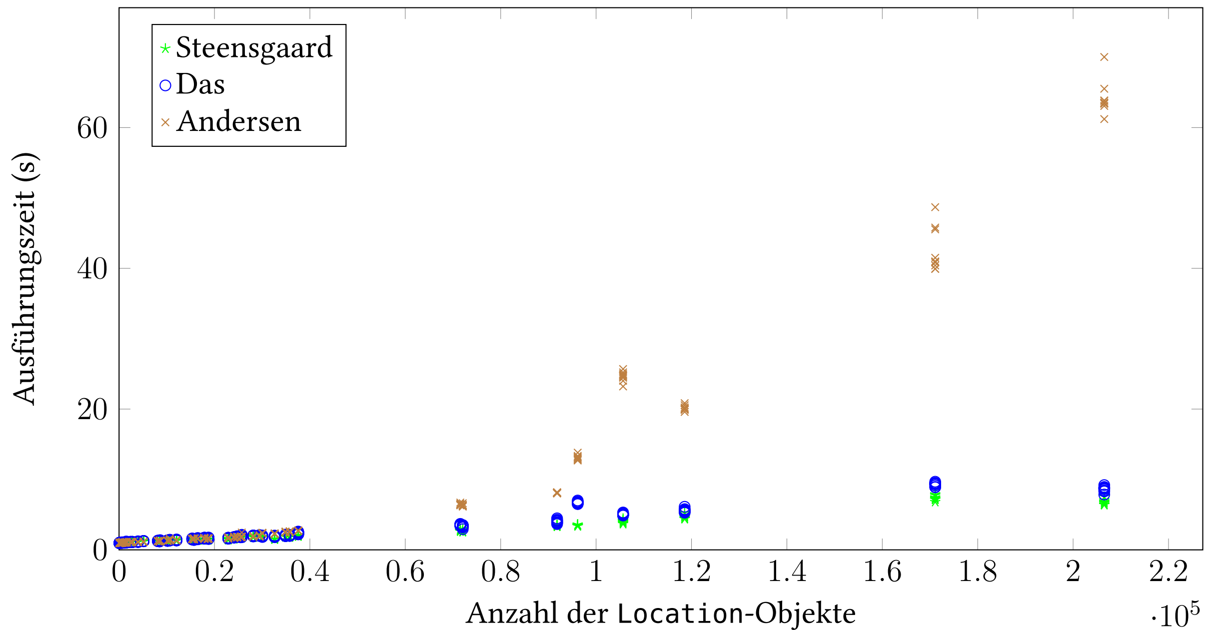


Abbildung 4.5.: Ausführungszeiten der Analysen in Relation zur Anzahl der `Location`-Objekte in der reduzierten Zwischendarstellung des Programms. Darstellungen in Abhängigkeit zu anderen Werten und mit logarithmischen Skalen finden sich im Anhang A.4.

bekannt ist. Dadurch kann effizienter an der tatsächlichen Problemstellung gearbeitet werden. Außerdem kann bestehender Code genutzt werden, auch wenn dieser in einer anderen Sprache vorliegt. Beispielsweise wurde für die Auswertung der Analysen die Knoten- und Kantenzahl in der Zwischendarstellung mithilfe eines bestehenden Java Werkzeugs ermittelt, während die Prototypen selbst mit Scala entwickelt wurden.

Da SKilL nativen Code für die Zielsprachen erzeugt, kann die Programmierschnittstelle an Konventionen der jeweiligen Programmiersprache angepasst werden. Die Verwendung entspricht deshalb den Erwartungen die ein Entwickler der entsprechenden Programmiersprache mitbringt. Ein Nachteil der Codegenerierung besteht darin, dass beispielsweise für die IML-Darstellung sehr viel Code entsteht. Dieser sorgt selbst bei kleinen Prototypen für vergleichsweise lange Kompilierungszeiten, welche bei einem Ansatz ohne Codegenerierung nicht auftreten würden. Dafür ist die Serialisierung und Deserialisierung sehr schnell, was speziell für Prototypen im Bereich der Programmanalysen hilft, die oft sehr langen Laufzeiten zu verkürzen.

Custom Fields und Auto Fields

Viele Algorithmen benötigen neben den Datenstrukturen für die Darstellung des Ergebnisses zusätzliche Strukturen, um einen effizienten Ablauf des Algorithmus zu ermöglichen. Ein

Beispiel hierfür sind beispielsweise Markierungen, die beschreiben ob ein Knoten in einem Graph bereits bearbeitet wurde oder nicht. Diese Daten sind für das Ergebnis nicht relevant und sollten nicht in den serialisierten Daten enthalten sein.

SKilL ermöglicht solche Datenfelder durch Auto Fields, sowie den Spezialfall Custom Fields. Mithilfe dieser Mechanismen können die Felder markiert werden, die nicht serialisiert werden sollen und SKilL ignoriert diese automatisch bei der Serialisierung.

Auf- und Abwärtskompatibilität

Zusätzlich dazu, dass irrelevante Datenfelder nicht in den serialisierten Daten enthalten sein sollten, wird durch solche Felder auch die Spezifikation des Datenmodells unübersichtlicher. Da SKilL eine weitreichende Abwärtskompatibilität implementiert können beispielsweise Auto Fields in den Spezifikationen für andere Programme ausgelassen werden.

Während der Entwicklung eines Prototypen kommt es häufig vor, dass sich das verwendete Datenmodell ändert. Bestehende Dateien können aufgrund der Aufwärtskompatibilität in vielen Fällen ohne erneute Generierung deserialisiert werden. Beispielsweise wurde während der Implementierung auf eine neuere Version von SKilL gewechselt, wobei auch die Spezifikation angepasst wurde. Die Testprogramme können trotzdem weiterhin eingelesen werden.

Interfaces

Zur Entwicklung der Prototypen war ursprünglich vorgesehen, die reduzierte Zwischendarstellung mithilfe von SKilL-Interfaces (siehe Kapitel 3.2.1) zu erzeugen. Dieser Ansatz wurde aus mehreren Gründen verworfen und durch die aktive Erzeugung der reduzierten Darstellung durch ein zusätzliches Programm ersetzt.

Zum einen muss die Spezifikation stark angepasst werden, wenn bestehende Klassen durch Interfaces abstrahiert werden sollen. Felder, die alle Typen, welche das Interface implementieren, gemeinsam haben sollen, müssen aus den Typbeschreibungen selbst entfernt werden und ins Interface verschoben werden. Es ist also ein großer Eingriff in die bestehende Spezifikation notwendig.

Zum anderen wird in jedem Fall ein aktiver Übersetzungsschritt benötigt. Dies liegt daran, dass in der reduzierten Darstellung weniger Knoten existieren sollen, als in der IML. Dabei müssen auch die Referenzen auf Kindknoten angepasst werden, was nicht über eine bloße Anpassung der Datenstruktur erreicht werden kann. Aus diesem Grund bietet es sich an, die Zwischenstruktur mithilfe eines zusätzlichen Übersetzungsschrittes über unabhängige Datentypen darzustellen.

Dies ermöglicht es außerdem direkte Funktionsaufrufe schon in der Zwischenstruktur als Zuweisungen der Parameter und Rückgabewerte darzustellen, oder Aufrufe von speziellen

Funktionen wie `malloc` nicht als Funktionsaufruf, sondern als `Allocation`-Objekt darzustellen.

5. Zusammenfassung und Ausblick

Das Serialisierungsframework SKilL stellt eine Alternative zur aktuell in Bauhaus verwendeten, XML-basierten Serialisierung dar und könnte diese in Zukunft ablösen. Dabei bietet es gegenüber XML den Vorteil, dass die Datendefinition von SKilL typgeprüft wird und die Serialisierung durch das Typsystem der entsprechenden Programmiersprache abgesichert ist. Ein weiterer Vorteil ist das binäre Datenformat, dass deutlich kleinere Dateien ermöglicht. Dies ist insbesondere im Bereich der Programmanalysen relevant, da Programmdarstellungen schnell sehr groß werden können. Zusätzlich kommen alle weiteren Vorteile zum Tragen, die in Kapitel 4.4 aufgeführt werden. Beispielsweise bietet SKilL die Möglichkeit das Datenmodell an einer zentralen Stelle zu spezifizieren und zu ändern. Durch die Auf- und Abwärtskompatibilität kann eine maximale Interoperabilität zwischen verschiedenen Werkzeugen gewährleistetet werden, auch wenn diese nicht jeder neuen Version des Datenmodells angepasst werden.

Im Rahmen dieser Arbeit sollten die Prototypenentwicklungseigenschaften von SKilL untersucht werden. Dafür wurden Prototypen für die drei Zeigeranalysen nach Steensgaard, Das und Andersen entwickelt. Es wurde eine von allen Zeigeranalysen gemeinsam genutzte Darstellung der Programme entwickelt. Außerdem wurden zusätzliche Werkzeuge geschaffen, um Ergebnisse von Zeigeranalysen in einem Graphviz Format zur visuellen Darstellung auszugeben und gegenseitig auf Inklusion zu überprüfen.

Die Ergebnisse der Analysen erscheinen nach der Evaluation plausibel und die Laufzeiten liegen für die verwendeten Beispielprogramme innerhalb der Worst-Case Grenzen der jeweiligen Analyse. Die Implementierungen stellen somit funktionsfähige Prototypen der Zeigeranalysen dar.

Probleme während der Umsetzung ließen sich nie auf SKilL zurückführen, sondern hingen immer mit der Bearbeitung der jeweiligen Problemstellung zusammen.

Ausblick

Aktuell befindet sich eine weitere Masterarbeit in Bearbeitung, welche das Ziel hat, SKilL stärker in Bauhaus zu integrieren. Das Bauhaus Frontend könnte die Zwischendarstellung der Programme dann beispielsweise direkt mit SKilL serialisieren. In Zukunft werden also vermutlich viele Prototypen auf dieser Basis implementiert.

Dies ermöglicht es, dass Entwickler Prototypen in ihrer bevorzugten Sprache entwickeln, ohne die Serialisierung und Deserialisierung der Daten zeitaufwändig selbst implementieren zu müssen.

A. Anhang

A.1. Liste der Programme, die zur Analyse zur Verfügung gestellt wurden

Programm	IML Objekte	IML Referenzen	Objekte in reduzierter Darstellung
aget	19156	57156	2394
Astro	52856	168812	32618
bash	843653	2746253	209372
bashversion	3221	7276	309
bc	89054	270676	22888
bf_test	9751	27507	1584
bison	301696	1020870	72052
bluefish	915273	3141440	118571
concepts	49637	156176	10624
cook	405716	1292034	91825
cu	186258	606265	37562
darkhttpd	28945	87925	5140
dc	40252	125126	8135
doc2gih	28446	80034	8477
gethost	8306	21472	679
gnugo	2712798	9097736	1526963
gnuplot	76629	242291	206530
gnuplot_x11	923099	3006837	15336
gqview	1331004	4547014	171091
grep	140073	462742	35772
joseki	86911	293879	18762
make.new	143355	459160	30048
mkbuiltins	14905	44752	2775
mkeyes	8260	22959	1164
mkpat	75567	236021	15693

mksignames	5193	12553	474
mksyntax	4651	11690	718
nano	110685	348836	24108
psize.aux	3983	8978	98
screen	432696	1377895	96146
sed	152306	495132	34891
sgfgen	2322	4633	197
simple	1347	1718	75
smtprc	71627	226534	12044
tcsh	467090	1510538	105659
time	10255	29934	1507
trueprint	94235	294494	18588
tstuu	22130	65419	3976
units	44238	134764	10169
uuchk	91244	302803	17834
uucico	353390	1145969	71535
uuconv	126725	409485	25726
uucp	133664	438629	24763
uulog	85343	281383	16490
uuname	51159	166573	8593
uupick	95344	314075	17923
uustat	146887	480352	28134
uux	134466	442668	25511
uuxqt	157157	514020	29457

Tabelle A.1.: Die im März 2016 zur Verfügung gestellten Programme, die mit den Prototypen analysiert werden sollen, sowie eine Abschätzung deren Größe in Form der IML Knoten- und Kantenzahl.

A.2. Durchschnittliche Anzahl der Elemente in den erzeugten Zeigerzielmengen

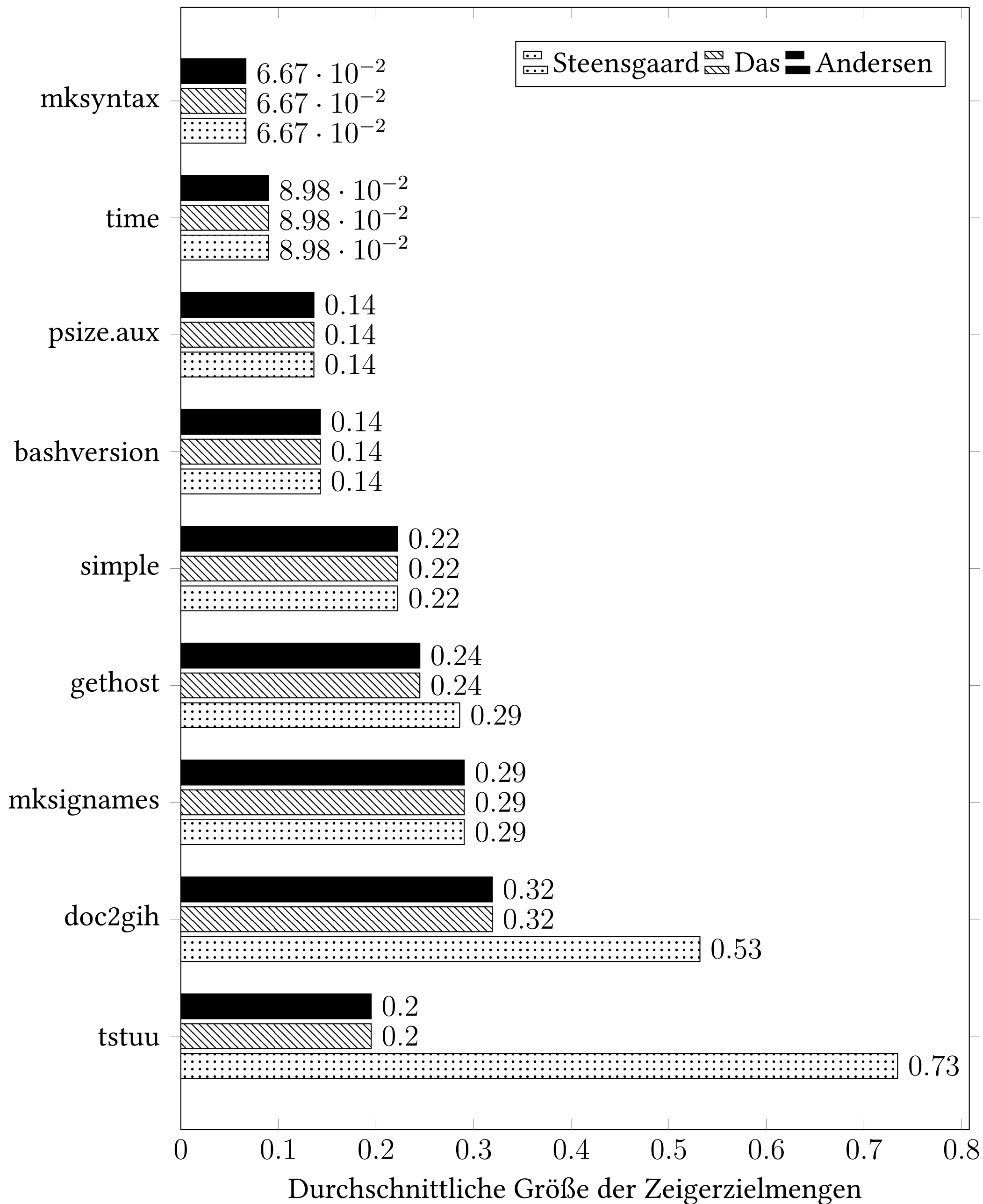


Abbildung A.1.: Durchschnittliche Größen der Zeigerzielmengen für alle getesteten Programme (1).

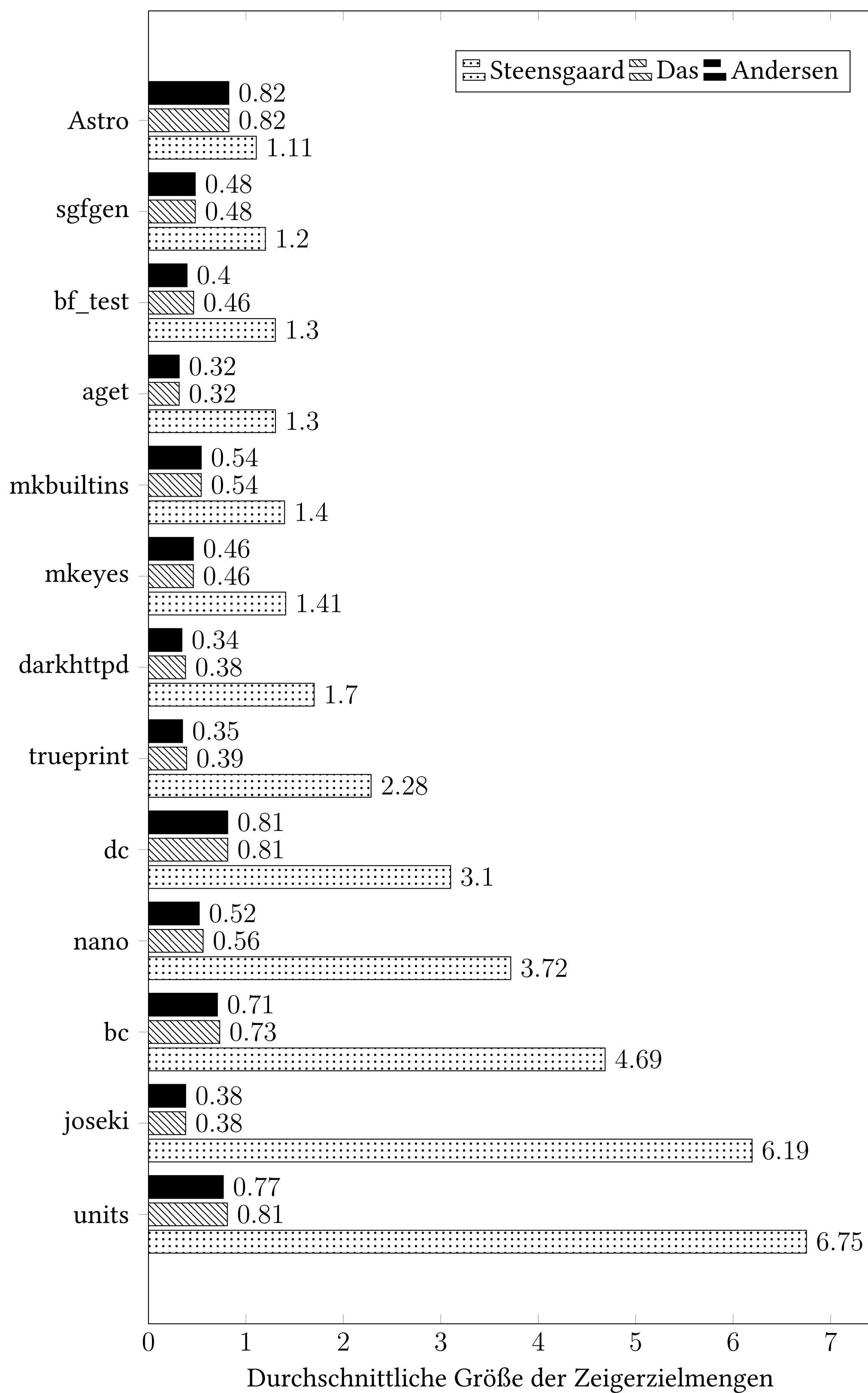


Abbildung A.2.: Durchschnittliche Größen der Zeigerzielmengen für alle getesteten Programme (2).

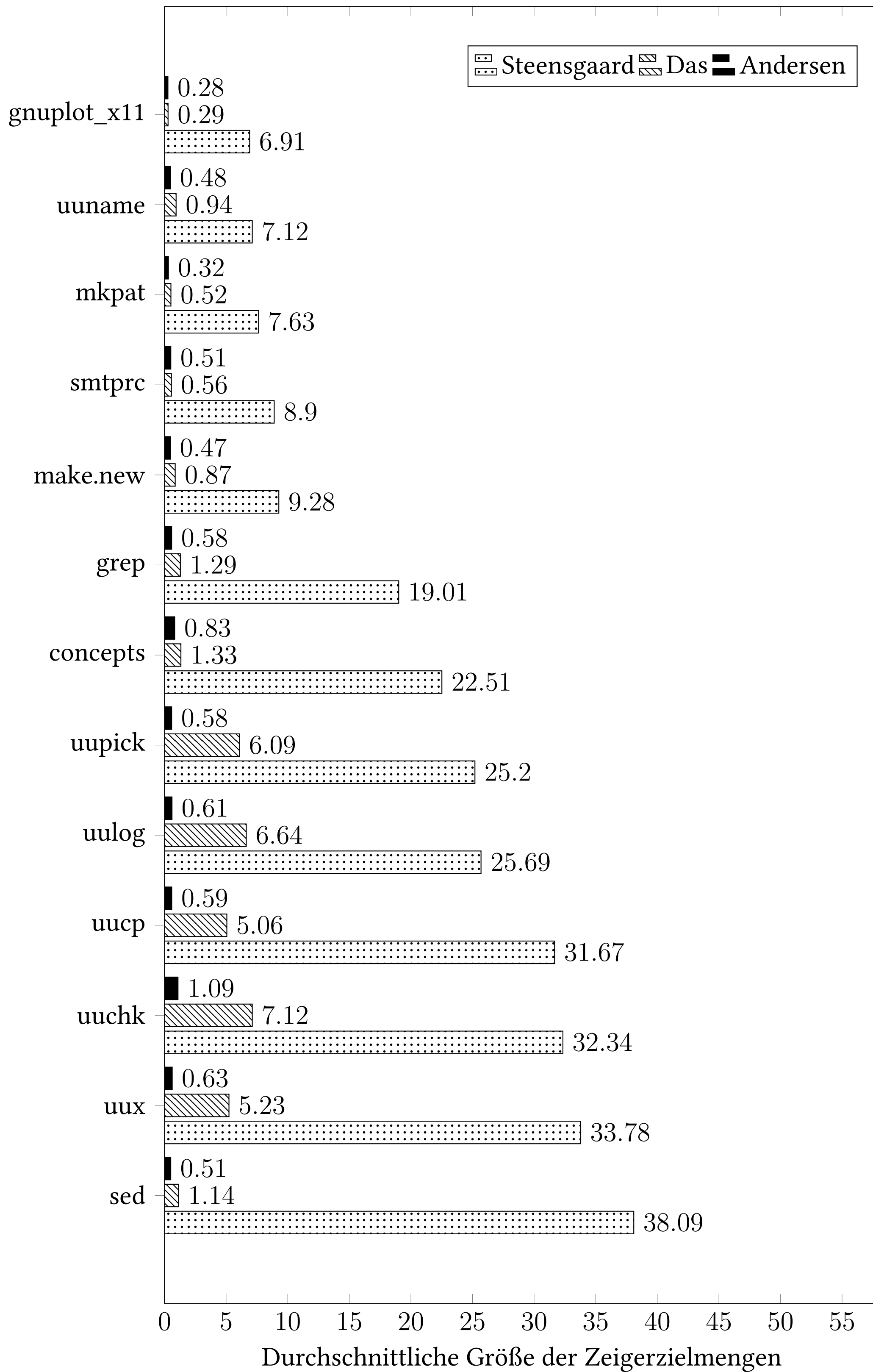


Abbildung A.3.: Durchschnittliche Größen der Zeigerzielmengen für alle getesteten Programme (3).

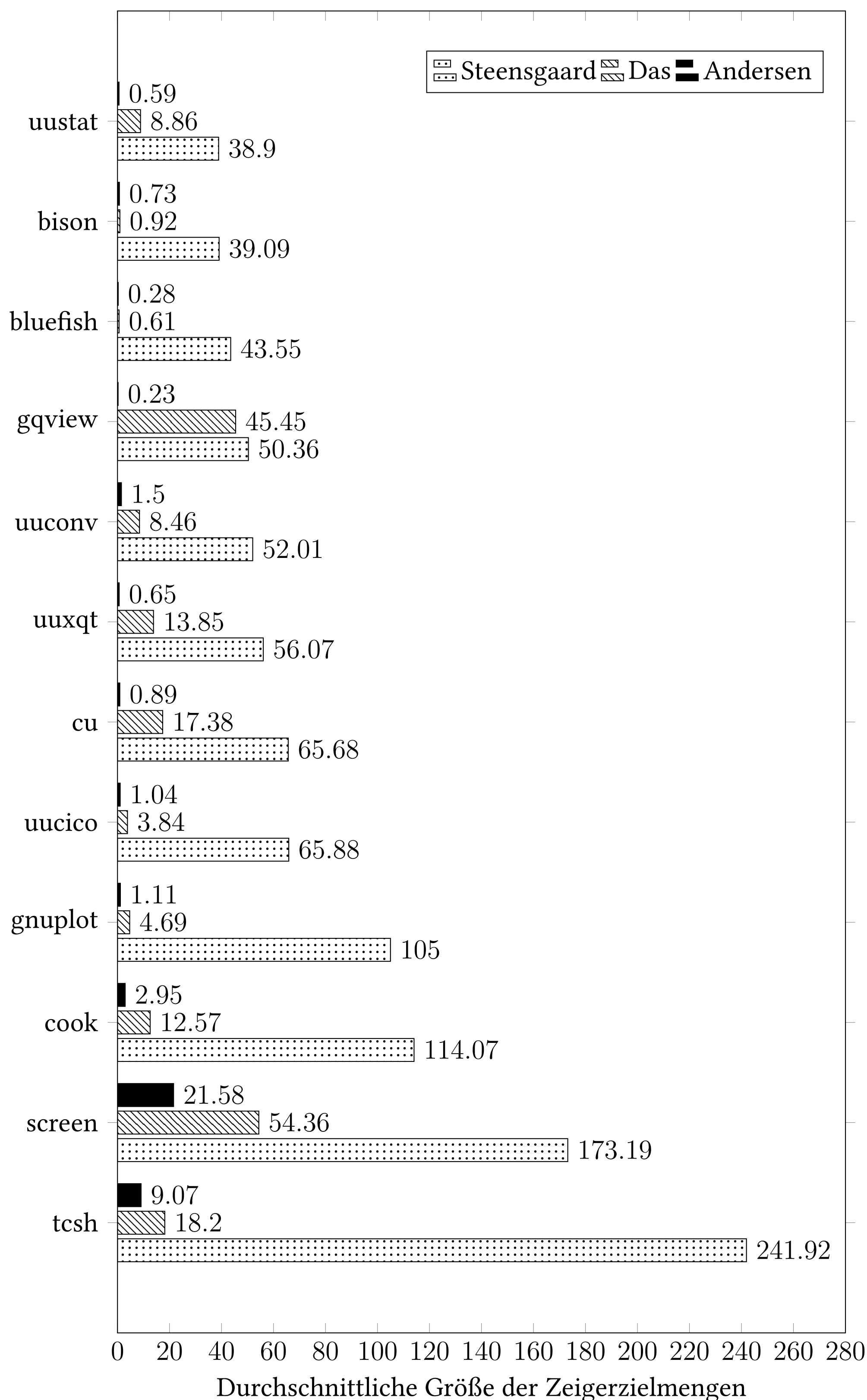


Abbildung A.4.: Durchschnittliche Größen der Zeigerzielmengen für alle getesteten Programme (4).

A.3. Maximale Anzahl der Elemente in den erzeugten Zeigerzielmengen

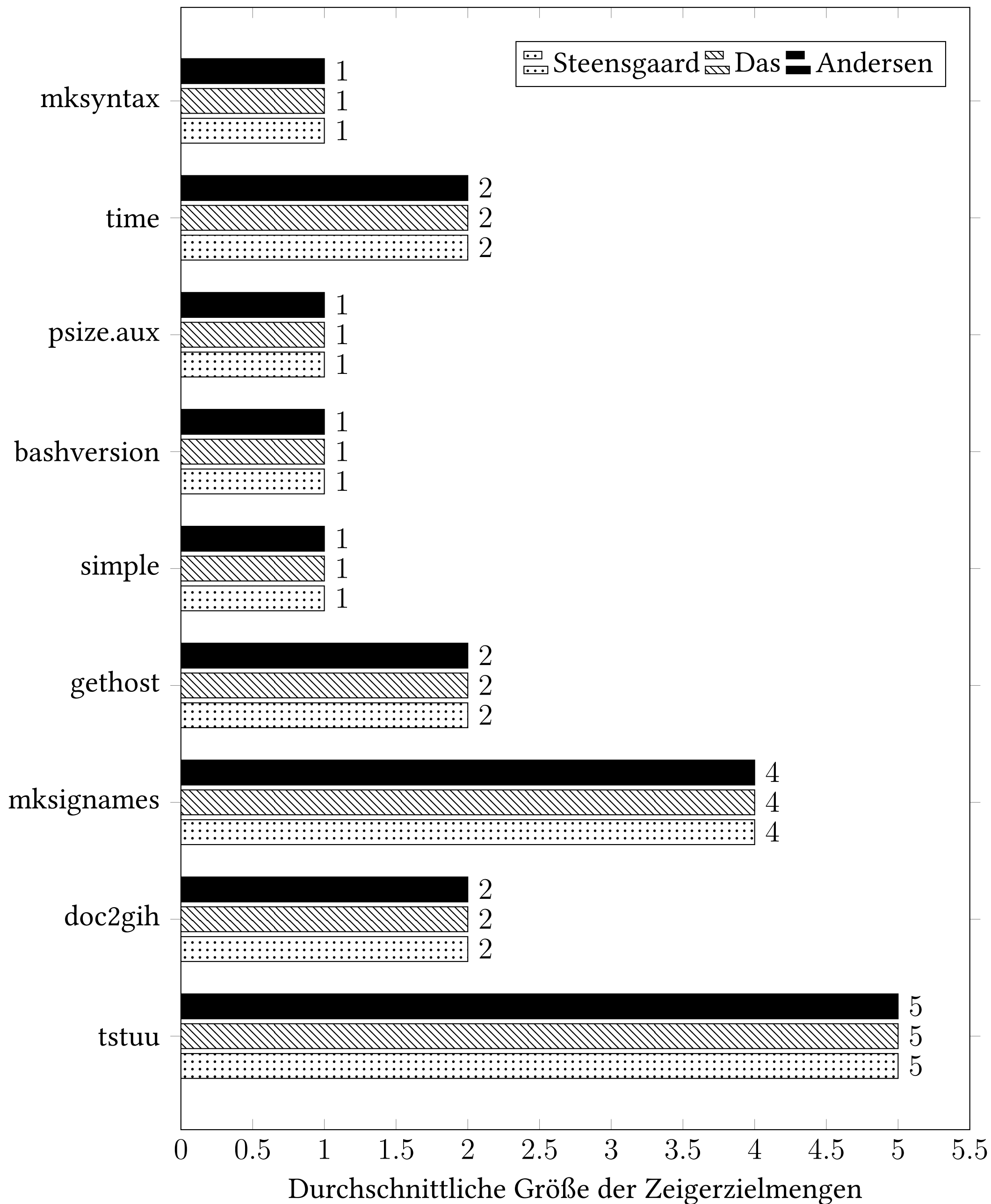


Abbildung A.5.: Maximale Größe der Zeigerzielmengen für alle getesteten Programme (1).

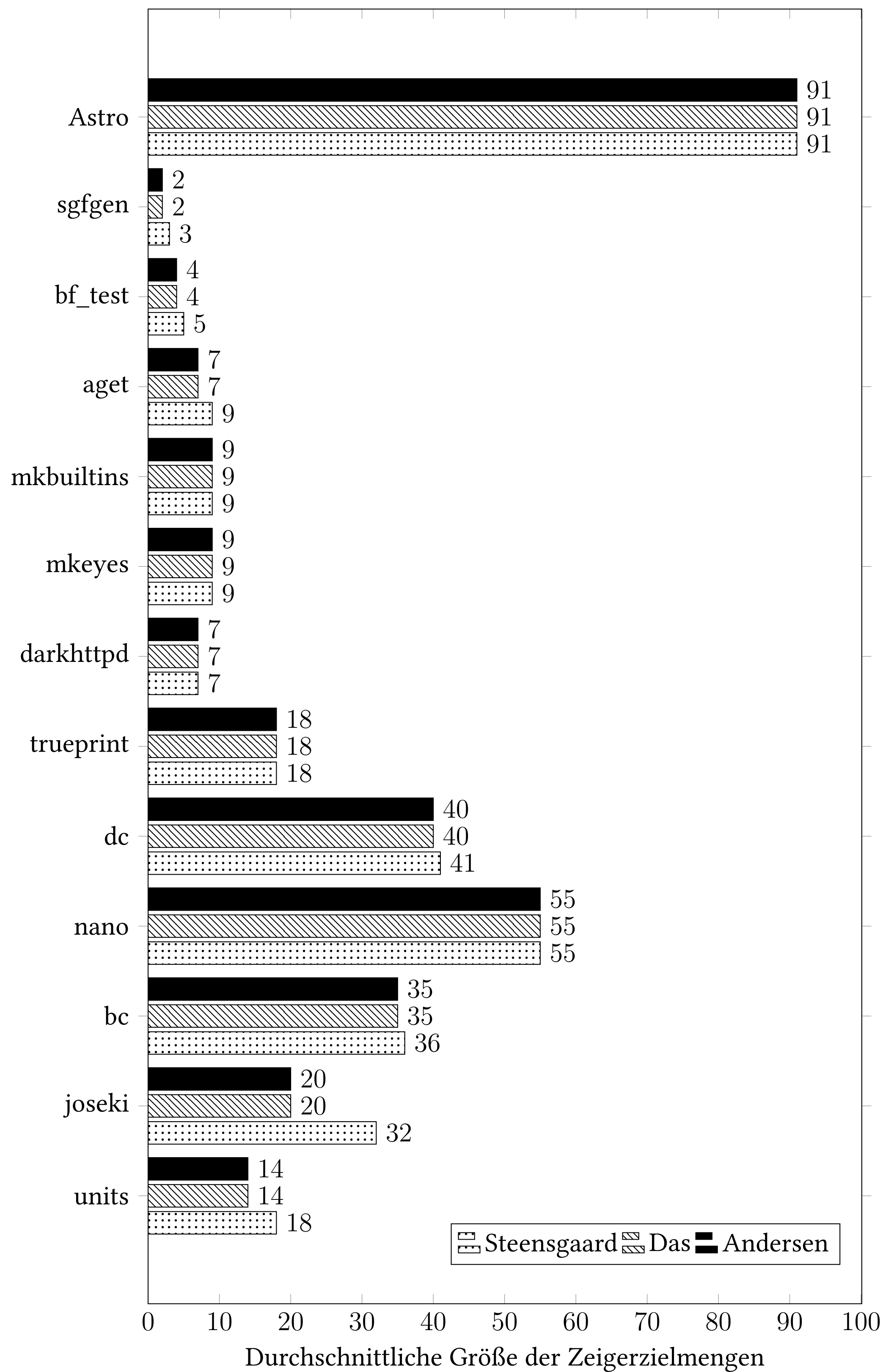


Abbildung A.6.: Maximale Größe der Zeigerzielmengen für alle getesteten Programme (2).

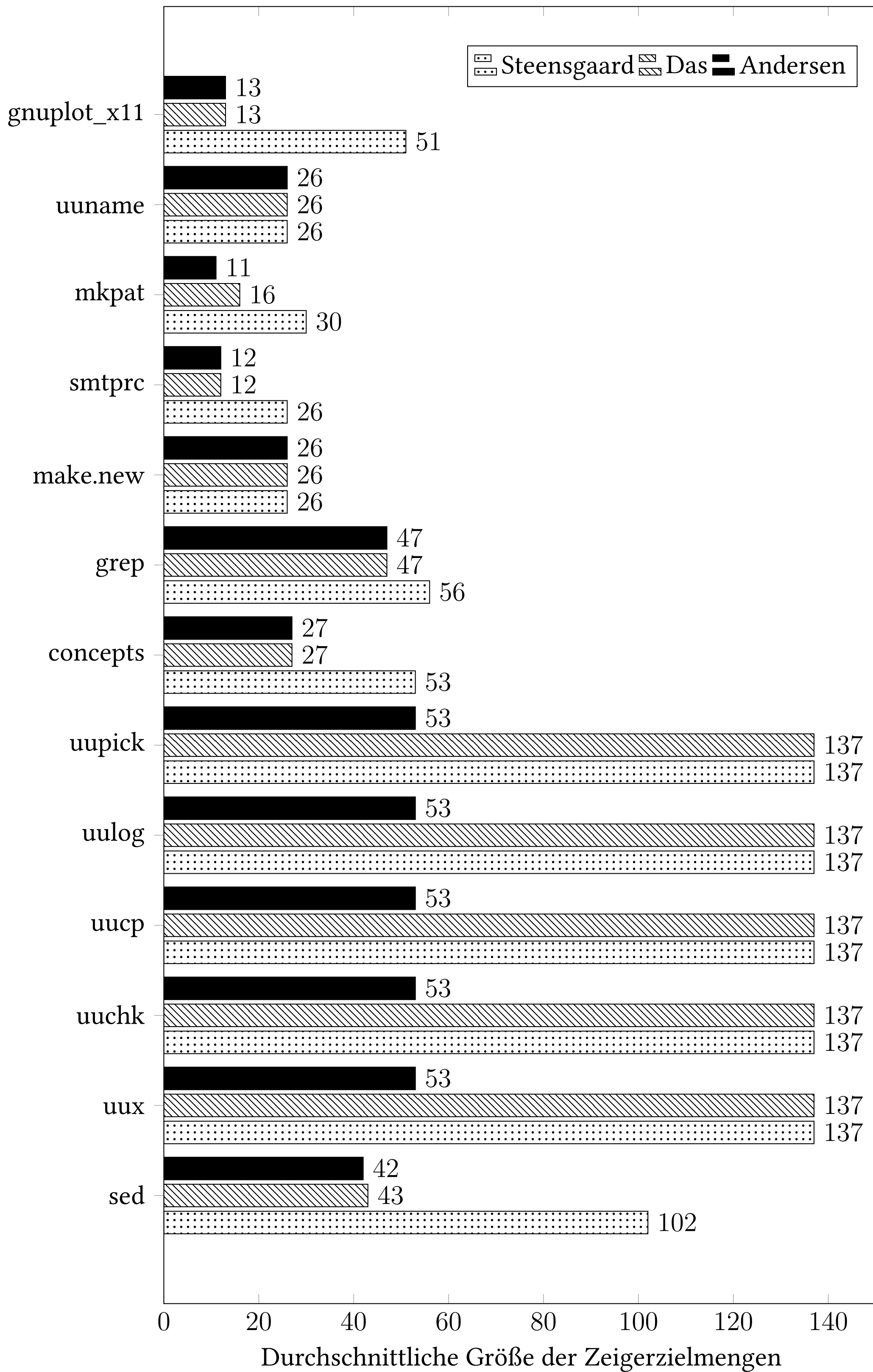


Abbildung A.7.: Maximale Größe der Zeigerzielmengen für alle getesteten Programme (3).

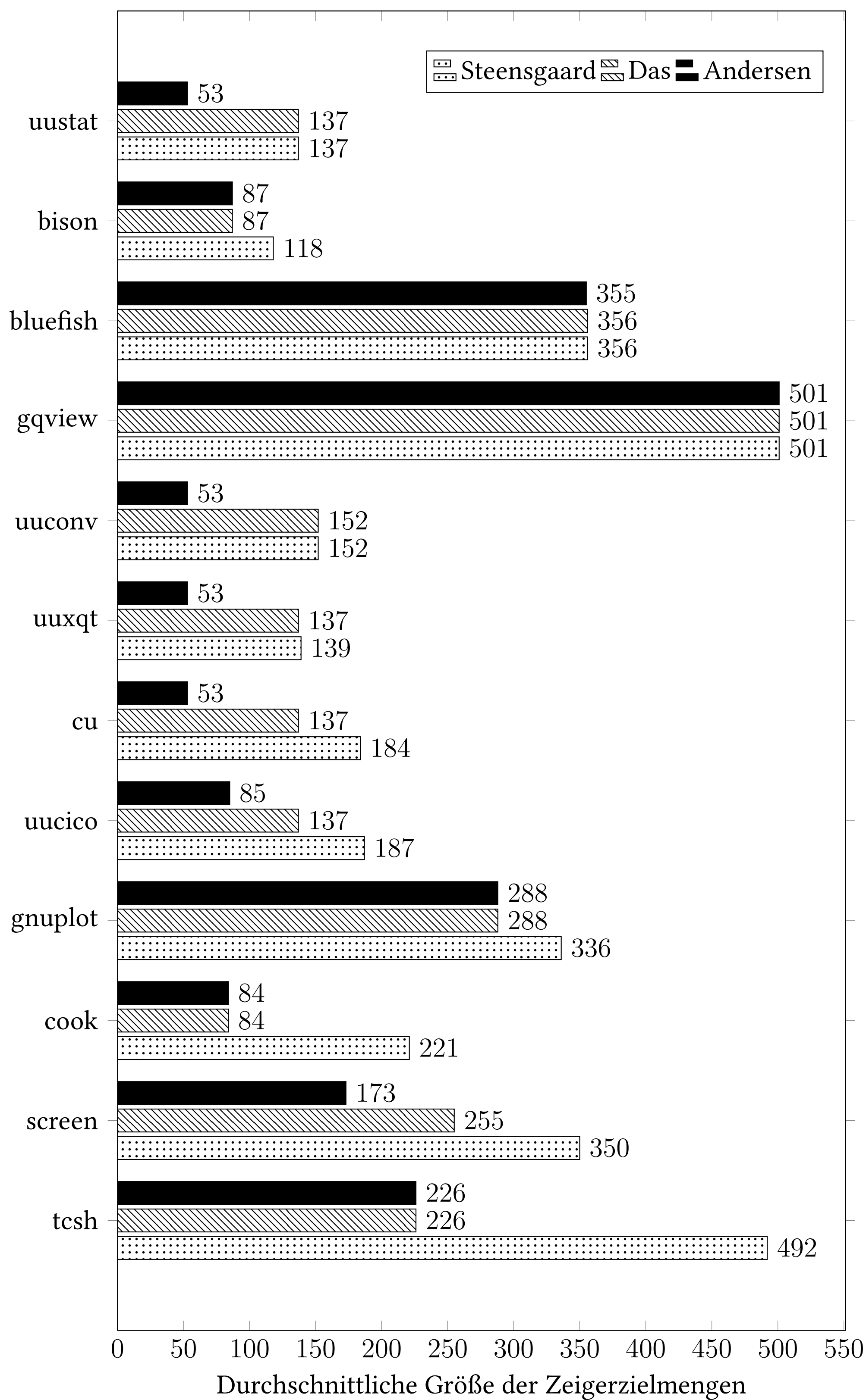


Abbildung A.8.: Maximale Größe der Zeigerzielmengen für alle getesteten Programme (4).

A.4. Ausführungszeiten der Analysen

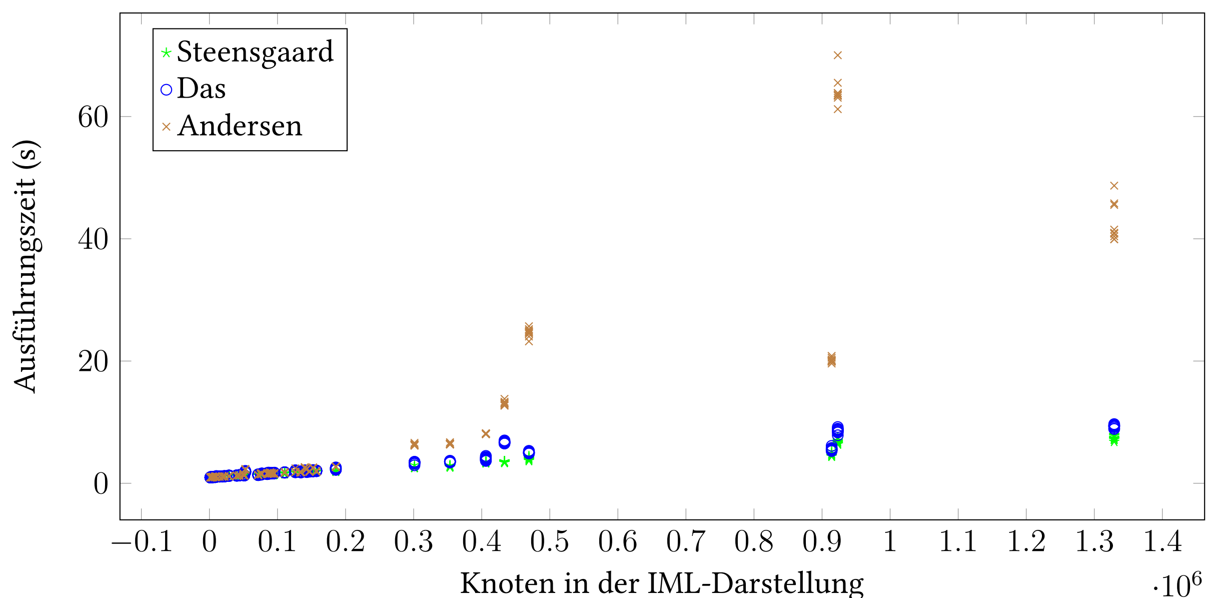


Abbildung A.9.: Lineare Visualisierung der Ausführungszeit in Abhängigkeit der Anzahl an Objekten in der IML-Darstellung des Programms.

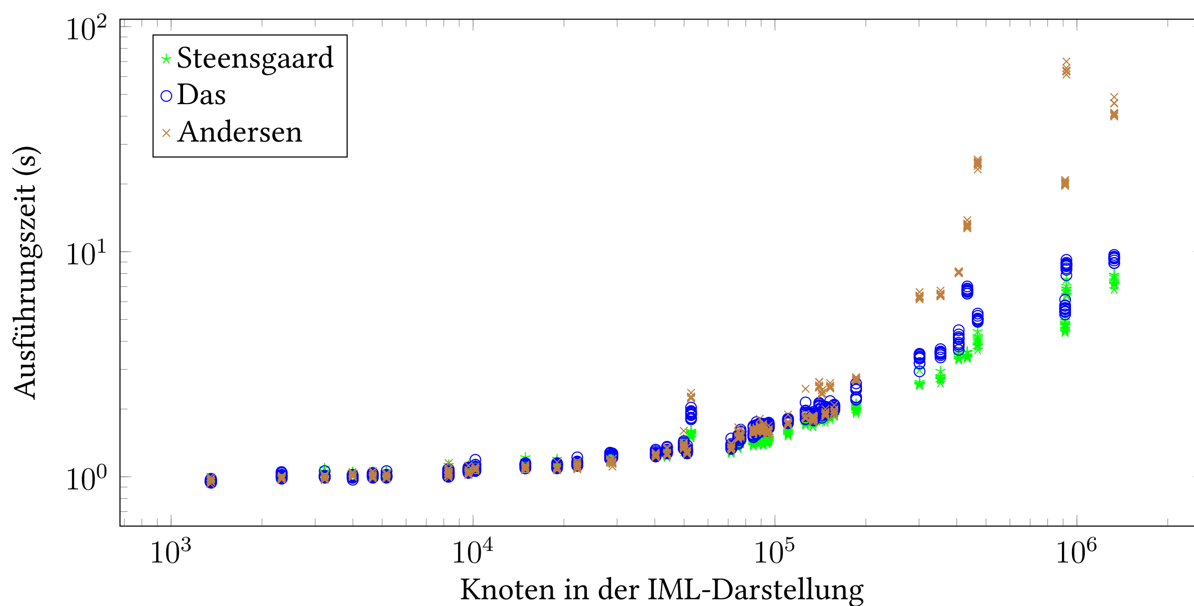


Abbildung A.10.: Logarithmische Visualisierung der Ausführungszeit in Abhängigkeit der Anzahl an Objekten in der IML-Darstellung des Programms.

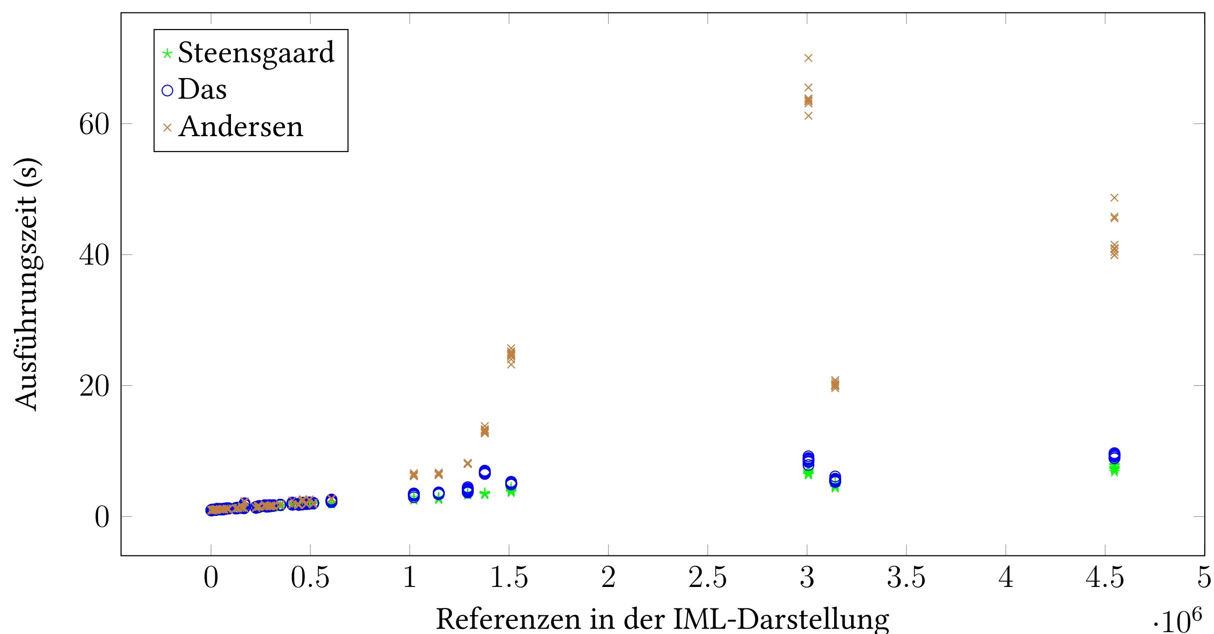


Abbildung A.11.: Lineare Visualisierung der Ausführungszeit in Abhängigkeit der Anzahl an Objektreferenzen in der IML-Darstellung des Programms.

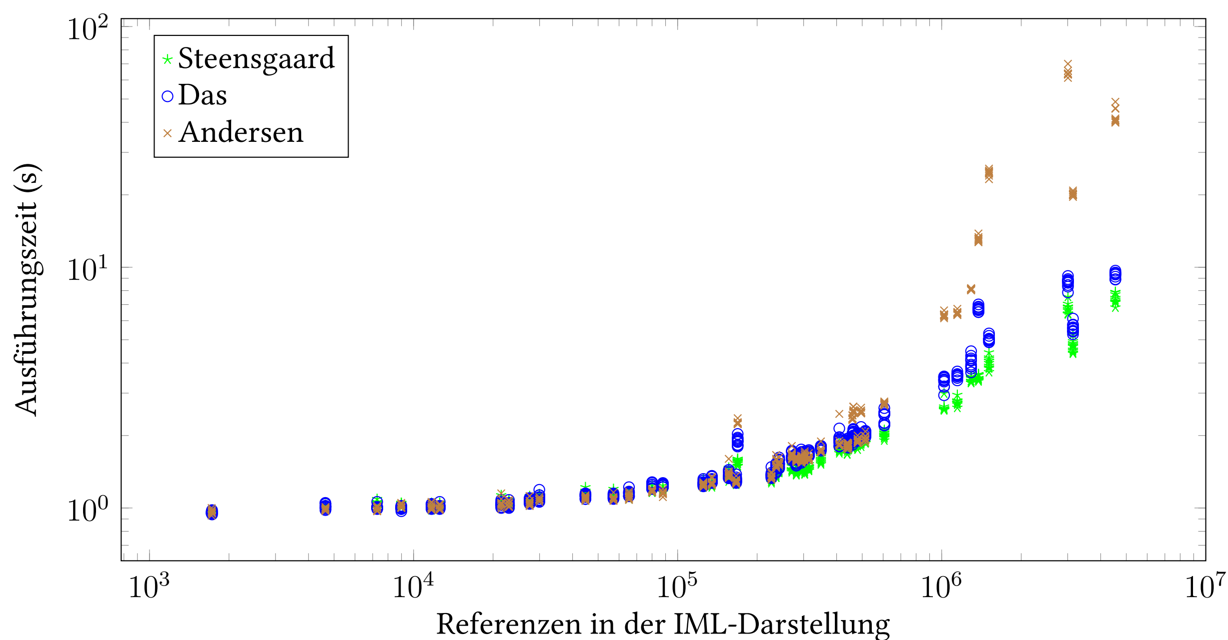


Abbildung A.12.: Logarithmische Visualisierung der Ausführungszeit in Abhängigkeit der Anzahl an Objektreferenzen in der IML-Darstellung des Programms.

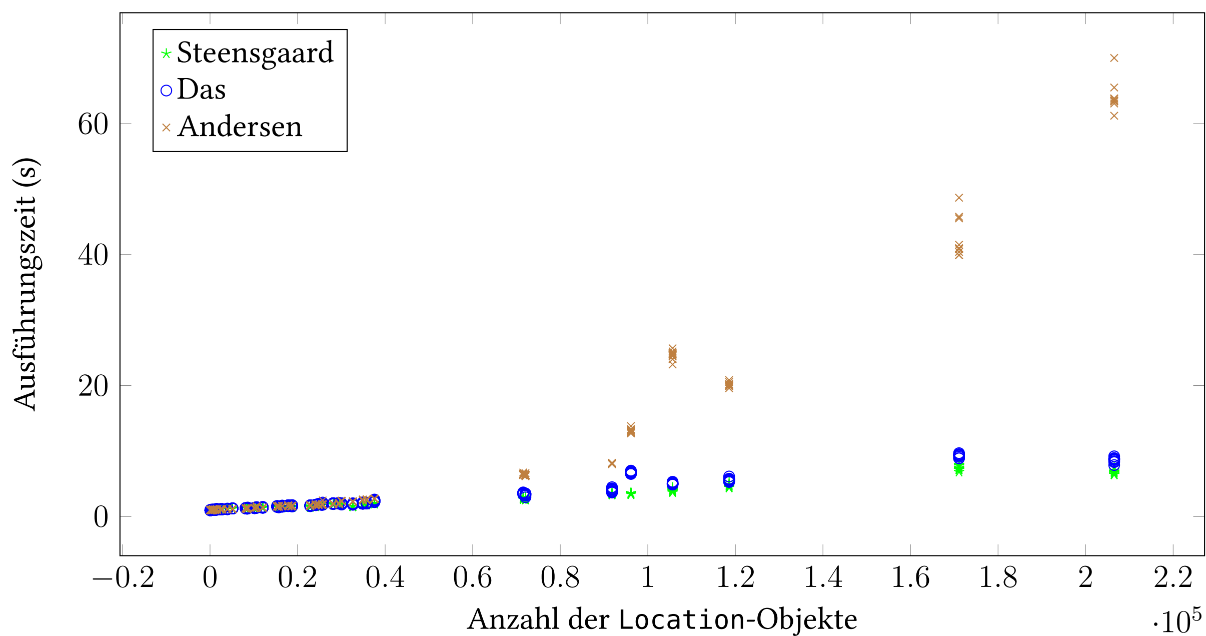


Abbildung A.13.: Lineare Visualisierung der Ausführungszeit in Abhängigkeit der Anzahl an `Location`-Objekten.

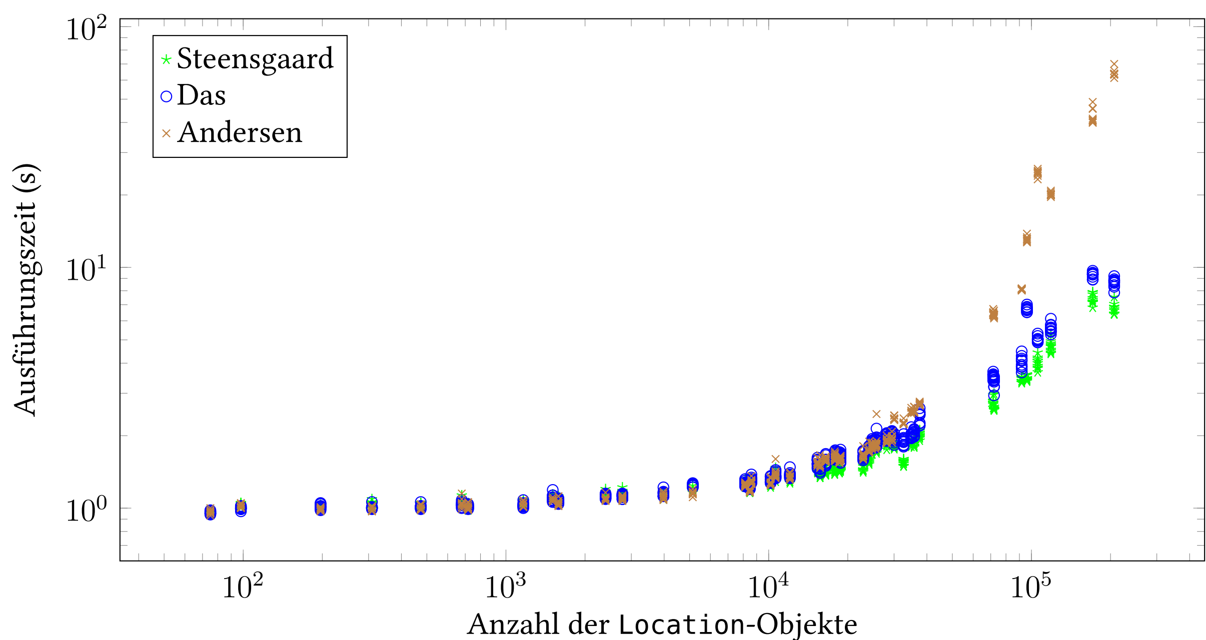


Abbildung A.14.: Logarithmische Visualisierung der Ausführungszeit in Abhängigkeit der Anzahl an `Location`-Objekten.

Literaturverzeichnis

[And94] L. O. Andersen. „Program analysis and specialization for the C programming language“. Diss. University of Cophenhagen, 1994 (zitiert auf S. 25).

[BC90] G. Bracha, W. Cook. „Mixin-based inheritance“. In: *ACM Sigplan Notices* 25.10 (1990), S. 303–311 (zitiert auf S. 33).

[Bro87] F. P. Brooks. „No Silver Bullet: Essence and Accidents of Software Engineering“. In: *IEEE Computer* 20 (1987), S. 10–19 (zitiert auf S. 11).

[Das00] M. Das. „Unification-based pointer analysis with directional assignments“. In: *Acm Sigplan Notices* 35.5 (2000), S. 35–46 (zitiert auf S. 19, 23, 49).

[EKP+99] T. Eisenbarth, R. Koschke, E. Plödereder, G. Girard, M. Würthner. „Projekt Bauhaus–Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen“. In: *Proceedings Workshop Reengineering, Bad Honnef.* Bd. 27. 1999, S. 28 (zitiert auf S. 9, 12).

[Fah00] M. Fahndrich. „Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C“. In: *Proceedings of the 7th International Static Analysis Symposium.* Juni 2000 (zitiert auf S. 19).

[Fel14] T. Felden. „Efficient and Change-Tolerant Serialization for Program Analysis Tool-Chains.“ In: *Softwaretechnik-Trends* 34.2 (2014) (zitiert auf S. 9, 12).

[GH98] R. Ghiya, L. J. Hendren. „Putting pointer analysis to work“. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. 1998, S. 121–133 (zitiert auf S. 14).

[GJSB05] J. Gosling, B. Joy, G. Steele, G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005. ISBN: 0321246780 (zitiert auf S. 33).

[Har09] B. C. Hardekopf. „Pointer analysis: building a foundation for effective program analysis“. In: (2009) (zitiert auf S. 15).

[Hin01] M. Hind. „Pointer analysis: haven't we solved this problem yet?“ In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. 2001, S. 54–61 (zitiert auf S. 19).

[HL07] B. Hardekopf, C. Lin. „The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code“. In: *ACM SIGPLAN Notices.* Bd. 42. 6. ACM. 2007, S. 290–299 (zitiert auf S. 25, 40, 41).

[HT01] N. Heintze, O. Tardieu. „Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second“. In: (2001) (zitiert auf S. 25).

[ISO11] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dez. 2011, 683 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853 (zitiert auf S. 13).

[Lan92] W. Landi. „Undecidability of static analysis“. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), S. 323–337 (zitiert auf S. 15).

[MSB11] G. J. Myers, C. Sandler, T. Badgett. *The art of software testing*. John Wiley & Sons, 2011 (zitiert auf S. 45).

[PKH07] D. J. Pearce, P. H. Kelly, C. Hankin. „Efficient field-sensitive pointer analysis of C“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.1 (2007), S. 4 (zitiert auf S. 16).

[RVP06] A. Raza, G. Vogel, E. Plödereder. „Bauhaus–a tool suite for program analysis and reverse engineering“. In: *International Conference on Reliable Software Technologies*. Springer. 2006, S. 71–82 (zitiert auf S. 9, 12).

[SAB+07] S. R. Schach, T. O. Adeshiyan, D. Balasubramanian, G. Madl, E. P. Osses, S. Singh, K. Suwanmongkol, M. Xie, D. G. Feitelson. „Common coupling and pointer variables, with application to a Linux case study“. In: *Software Quality Journal* 15.1 (2007), S. 99–113 (zitiert auf S. 14).

[Ste96a] B. Steensgaard. „Points-to analysis by type inference of programs with structures and unions“. In: *International Conference on Compiler Construction*. Springer. 1996, S. 136–150 (zitiert auf S. 22).

[Ste96b] B. Steensgaard. „Points-to analysis in almost linear time“. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1996, S. 32–41 (zitiert auf S. 16, 19).

[Tar75] R. E. Tarjan. „Efficiency of a good but not linear set union algorithm“. In: *Journal of the ACM (JACM)* 22.2 (1975), S. 215–225 (zitiert auf S. 19).

[Tar83] R. E. Tarjan. *Data structures and network algorithms*. Bd. 44. Siam, 1983 (zitiert auf S. 16).

Alle URLs wurden zuletzt am 06. 10. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift