

# Verbesserung statischer Programmanalysen mit Hilfe einer Annotationsprache

Judith Rohloff und Martin Grabmüller

Fakultät IV Elektrotechnik und Informatik, Technische Universität Berlin  
Berlin, Germany  
`rohloff@cs.tu-berlin.de` `magr@cs.tu-berlin.de`

**Zusammenfassung** Mit Hilfe der statischen Programmanalyse werden Informationen über ein Programm zur Übersetzungszeit ermittelt. Diese Informationen können zum einen für Compileroptimierungen genutzt werden und können zum anderen dem Programmierer Auskunft über sein Programm geben. Die meisten Analysen sind nicht vollständig berechenbar. Aus diesem Grund wird im Allgemeinen nur eine Approximation der Ergebnisse berechnet. Es wird gezeigt, wie diese Approximationen mit Hilfe von Annotationen verbessert werden können. Diese Verbesserungen werden am Beispiel der Striktheits- und der Zeitkomplexitätsanalyse vorgestellt und erläutert.

## 1 Einführung

Die statische Analyse dient der Ermittlung von Informationen zur Übersetzungszeit. Interessante Eigenschaften eines Programms, die sich mit statischen Analysen ermitteln lassen, sind u.a. Striktheit, Zeitkomplexität, mögliche auftretende Fehler oder die möglichen Ergebniswerte einer Funktion. Im Laufe dieses Beitrags beschränken wir uns auf die Striktheits- und Zeitkomplexitätsanalyse. Für beide Analysen existieren bereits Algorithmen, deren Ergebnisse sich durch die Benutzung einer Annotationsprache verbessern lassen.

Der Algorithmus für die Striktheit wurde von Mycroft entwickelt und in [1] präsentiert. Unter anderem wurde er von Burn [2] verbessert. Die zweite von uns vorgestellte Analyse ist die Zeitkomplexitätsanalyse. Einer der ersten Ansätze zur automatischen Berechnung der Zeitkomplexität stammt von Wegbreit [3]. Rosendahl hat in [4,5] diesen aufgegriffen und ihn deutlich verbessert.

Annotationen bieten im Allgemeinen dem Programmierer die Möglichkeit, sein Wissen über spezielle Eigenschaften seiner Funktionen im Quelltext einzuarbeiten. Die Angabe solcher Eigenschaften beschränkt sich bei Annotationen nicht auf die Angabe von Typen, wie in den meisten Programmiersprachen, sondern lässt exaktere Beschreibungen, wie z.B. die Striktheit von Parametern oder die Zeitkomplexität einer Funktion, zu. Diese Annotationen sind fakultativ und von einander unabhängig, damit ist die Liste der möglichen annotierten Eigenschaften beliebig fortsetzbar. Die Möglichkeit Annotationen anzugeben, wird bereits in einigen Sprachen, so z.B. in Mercury [6], gegeben. Im Fall von Mercury

werden die Annotationen vom Compiler genutzt, um optimierte Programme zu erzeugen.

Mit Hilfe der von uns entwickelten Annotationen ist es möglich die Ergebnisse der Striktheitsanalyse deutlich genauer zu bestimmen als bei Mycroft. Außerdem ist die Berechnung der Zeitkomplexitätsanalyse für mehr Funktionen als bei Rosendahl möglich, wenn Annotationen benutzt werden.

## 2 Annotationsprache

Um verschiedene Analysen und deren Eigenschaften in ein Analysetool einbauen zu können, haben wir die im Folgenden vorgestellte Annotationsprache entwickelt. Die Annotationen müssen optional und jederzeit erweiterbar sein.

Für die Annotationen muss jeweils der Name der Analyse und eine genaue Eigenschaft angegeben werden. Der Name der Analyse wird als String angegeben und die Eigenschaft selber wird als in eckige Klammern gesetzte Zeichenkette angegeben. Diese Zeichenkette wird von jeder Analyse einzeln interpretiert da unterschiedliche Anforderungen an die Annotationen gestellt werden.

## 3 Annotationen am Beispiel

### 3.1 Striktheit

Bei der Striktheitsanalyse soll ermittelt werden, ob eine Funktion im Parameter  $x_i$  strikt ist, also ob  $f(e_1, \dots, e_{i-1}, \perp, e_{i+1}, \dots, e_n) = \perp$  gilt.

Der Algorithmus von Mycroft [1] erstellt, mit Hilfe einer abstrakten Interpretation  $\mathcal{S}$  mit der Domain  $\{0, 1\}$ , zunächst für jede Funktion  $f$  eine logische Funktion  $f^\#$  erstellt. Anschließend wird für rekursive Funktionen mittels einer Fixpunktbildung das Ergebnis ermittelt.

**Annotationen** Die Annotationen für die Striktheitsanalyse müssen nicht nur enthalten, in welchen Parametern die Funktion strikt ist und in welchen nicht, sondern auch, in welchen sie strikt sein könnte. Um dies genauer erklären zu können zunächst ein Beispiel:

```
DEF f(x:bool,y:int,z:int):int == IF x THEN y ELSE z FI
DEF g(a:bool,b:int):int == f(a,b,b)
```

Die Funktion  $g$  ist sowohl in  $a$  als auch in  $b$  strikt. Da aber  $f$  nur in  $x$  strikt ist, würde die Analyse ergeben, dass  $g$  nur in  $a$  strikt wäre, da die Information über  $y$  und  $z$  der Funktion  $f$  nicht in der Annotation enthalten ist. Aus diesem Grund muss die gesamte logische Funktion der abstrakten Interpretation angegeben werden. Die Annotation für die Funktion  $f$  ist also: `STRICT[x&(y?z)]`. Wobei `&` für  $\wedge$  und `?` für  $\vee$  steht.

Somit liegt es nahe, als Annotation eine logische Funktion zu nehmen, welche zu einem Element der Domain ausgewertet werden kann.

**Algorithmus** Der Algorithmus für die Striktheitsanalyse mit Annotationen ist im wesentlichen der Algorithmus von Mycroft [1]. Zunächst wird für jede Funktionen, für die keine Annotation vorliegt eine logische Funktion mittels der Abstrakten Interpretation  $\mathcal{S}$  aufgebaut. Für die Funktionen  $f$  mit Annotation ist die logische Funktion  $f^\#$  die Annotation selbst. Anschließend wird über alle logischen Funktionen eine Fixpunktbildung durchgeführt. Das Ergebnis dieser Fixpunktbildung gibt an welche Funktion in welchen Parametern strikt ist und in welchen nicht.

**Ergebnisse** Diese Ergebnisse stellen nur eine obere Schranke für die Striktheit dar. Ist das Ergebnis zum Beispiel  $x \vee y$ , so ist es trotzdem möglich, dass die Funktion in  $x$  und  $y$  strikt ist, oder dass sie sogar nie definiert ist. Mycroft stellt in [1] auch einen Algorithmus für die untere Schranke der Striktheit vor. Durch Kombination beider Verfahren kann ein Ergebnisbereich berechnet werden.

Nun möchten wir noch zu einem anderen Beispiel übergehen, in diesem wird deutlich, wie sich die Ergebnisse einer Striktheitsanalyse mit Hilfe von Annotationen verbessern lassen.

```
DEF tru(x:int):bool == true
DEF f(x:int,y:int,z:int):int:STRICT[y] ==
    IF tru(x) THEN y ELSE z FI
DEF g(x:int,y:int,z:int):int ==
    IF tru(x) THEN f(0,y,x) ELSE f(0,y,z)
```

Die Funktion  $f$  ist nur strikt in  $y$ , da die Funktion  $tru$  immer wahr ist, somit wird immer der then-Zweig ausgeführt. Die Analyse ohne Annotationen kommt zu dem Ergebnis, dass  $f$  entweder in  $y$  oder in  $z$  strikt ist, da keine Informationen über die Semantik der Funktion  $tru$  vorliegen. Dies ist ein Beispiel für Unvollständigkeit der Striktheitsanalyse. Durch die Annotation, die angibt, dass die Funktion  $f$  nur in  $y$  strikt ist, also  $STRICT[y]$  wird diese Information der Analyse mitgeteilt. und somit das Ergebnis der Analyse für die Funktion  $g$  exakter. Durch die Annotation bei der Funktion  $f$  ergibt sich bei meinem Algorithmus, dass  $g$  nur in  $y$  strikt ist, während das Ergebnis bei Mycroft  $(y \vee x) \wedge (y \vee z)$  ist.

### 3.2 Zeitkomplexität

Unter der Zeitkomplexität einer Funktion versteht man die Zeit, die das Programm für die Berechnung benötigt. Diese Zeit wird in Abhängigkeit der Größe der Variablen angegeben. Oft ist es dabei nicht nur interessant in welcher Aufwandsklasse die Funktion liegt, sondern es sind wesentlich exaktere Ergebnisse gewünscht. Mit Hilfe des von Rosendahl [4,5] entwickelten Algorithmus für die Zeitkomplexitätsanalyse ist es möglich, für viele Funktionen die genaue Zeitkomplexität zu erhalten.

Ziel der Zeitkomplexitätsanalyse ist es eine solche Zeitfunktion für die obere Schranke einer Funktion zu bestimmen. Die Machbarkeit ist vor allem durch das Halteproblem beschränkt. Da die Analyse eine obere Schranke für die Zeit, die

das Programm bei der Ausführung benötigen wird, ausgibt, kann dies nicht für alle Funktionen möglich sein, da sonst das Halteproblem gelöst wäre.

Rosendahl hat in [4,5] einen Algorithmus zu Berechnung dieser Funktion für Listenprogramme vorgestellt. Dieser Algorithmus ist die Grundlage des hier vorgestellten Algorithmus zur Bestimmung der Zeitkomplexität mit Annotationen.

**Annotationen** Wie bereits erklärt ist das Ergebnis der Zeitkomplexitätsanalyse eine Funktion abhängig von der Größe ihrer Variablen. Bei einer Liste ist dies die Anzahl ihrer Elemente, also ihre Länge. Als mögliche mathematische Operationen stehen Multiplikation, Division, Subtraktion, Addition und Potenz zur Verfügung. Denkbar wäre außerdem der Logarithmus. Da in Rosendahls Algorithmus der Logarithmus als Ergebnis nicht möglich ist, wird dies auch hier nicht erlaubt.

Die Funktion *member* in nachfolgendem Beispiel ist mit einer Zeitannotation versehen. Diese besagt, dass die Berechnung dieser Funktion linear zur Länge der Liste *y* ist:

```
DEF member(x:int,y:List):bool:TIME[4+(10*y)] == ...
```

**Algorithmus** Der Algorithmus von Rosendahl [4,5] läuft in fünf Schritten:

1. Schrittzählversion aufbauen
2. in ein Zeitschrankenprogramm transformieren
3. mit Hilfe einer Datenflussanalyse vereinfachen
4. mit der Driving-Technik vereinfachen
5. Lösen von Differenzgleichungen

Neben einigen anderen Funktionen fügt Rosendahl die Funktion  $length^{-1}(x)$  zu jedem Zeitschrankenprogramm hinzu, diese erzeugt eine Liste mit *x* Elementen.

Eine Annotation für eine Funktion entspricht ihrer Zeitschrankenfunktion. Daher müssen die Schritte 1. und 2. nicht für annotierte Funktionen durchgeführt werden. Auch eine Vereinfachung ist für die Annotationen nicht nötig, da diese bereits fallunterscheidungsfrei sind.

**Ergebnisse** Der Algorithmus von Rosendahl ist zwar in der Lage für viele Funktionen eine Zeitkomplexitätsanalyse zu erstellen, hat aber auch deutliche Grenzen. Die Anzahl der analysierbaren Funktionen lässt sich mit Hilfe der Annotationen steigern. Außerdem müssen für annotierte Funktionen keine Zeitfunktionen aufgebaut werden und alle Transformationen für die Zeitfunktion werden überflüssig, da es sich bei einer Annotation bereits um eine fallunterscheidungsfreie Funktion handelt. Dadurch wird die Analyse beschleunigt. Also bewirken die Annotationen auch bei der Zeitkomplexitätsanalyse sowohl eine verbesserte als auch eine schnellere Analyse. Der von Rosendahl implementierte Algorithmus ist nicht in der Lage, die Zeitkomplexität aller Funktionen zu ermitteln. In seiner Masterarbeit [4] gibt er die Funktion *bsort*, diese entspricht einem Bubblesortalgorithmus, als ein Beispiel an. Auch mein Algorithmus ist natürlich nicht in der

Lage diese Funktion zu analysieren, aber es ist möglich eine Annotation für diese Funktion anzugeben. Mit Hilfe einer solchen Annotation ist es teilweise möglich Funktionen zu analysieren, die die Funktion *bsort* aufrufen. Zum Beispiel kann die Funktion  $f(x, y) = \text{bsort}(\text{union}(x, y))$  analysiert werden, wenn für *bsort* eine Annotation vorliegt. Da nur bekannt ist, dass *bsort* einen quadratischen Aufwand hat, wird hier die Annotation  $x^2$  gewählt. Außerdem ist die Zeitfunktion für *union* bekannt. Aus diesem Grund wird hier auch diese Funktion annotiert.

Hier der wesentliche Ausschnitt des Programms für die Funktion *f*:

```
DEF f(x>List,y>List):List = bsort(union(x,y))
DEF bsort(x>List):List:TIME[x^2]= ...
DEF union(x>List,y>List):List:TIME[4+16*x+10*x*y]=...
```

Unter Berücksichtigung der Annotationen ergibt sich für die Funktion *f* folgende Zeitfunktion:  $f(x, y) = 10x + x^2 + y^2 + 18xy + 8$

Wird für die Funktion *bsort* eine exaktere Annotation angegeben, so wird auch das Ergebnis für *f* exakter.

Die Funktion *f* ist eine von vielen Funktionen, die nicht mit Rosendahls Algorithmus analysiert werden können. Unter Zuhilfenahme der hier vorgestellten Annotationssprache ist es jetzt möglich solche Funktionen zu analysieren. Meine bisherigen Annotationen ermöglichen jedoch nicht die Analyse aller Funktionen. So kann z.B. die Funktion  $g(x) = \text{union}(\text{bsort}(x), y)$  auch mit Annotationen nicht analysiert werden, da die Länge der Ergebnisliste von *bsort* nicht ermittelt werden kann. Eine Lösungsidee hierfür wird im Ausblick erläutert.

## 4 Ergebnisse

Alle statischen Analysen haben ihre Grenzen, ihre Ergebnisse können mit Hilfe von Annotationen verbessert werden. Außerdem werden die Analysealgorithmen mit Hilfe der Annotationen effizienter, da für annotierte Funktionen viele Schritte der Algorithmen nicht ausgeführt werden müssen.

Durch die eingeführten Annotationen können Funktionen wesentlich exakter auf Striktheit analysiert werden. Bei der Analyse der Zeitkomplexität bewirken Annotationen, dass wesentlich mehr Funktionen, als durch Rosendahls Algorithmus [5,4], analysiert werden können. Kann eine Funktion *f* nicht mit Hilfe des von Rosendahl entwickelten Algorithmus analysiert werden, so können auch alle Funktionen, die *f* aufrufen nicht analysiert werden. Ist *f* jedoch annotiert, so können einige Funktionen die *f* aufrufen analysiert werden.

## 5 Ausblick

Trotz deutlicher Verbesserungen durch die von uns eingeführten Annotationen kann das Analysewerkzeug noch erweitert werden. Denkbar sind weitere Analysen, der Einbau von Checkingalgorithmen und eine weitere Verbesserung der Zeitkomplexitätsanalyse.

Da der Algorithmus von Rosendahl nur Listenprogramme analysiert, aber auch Zeitfunktionen für andere Datentypen wünschenswert sind, stellt sich die Frage, wie man diesen Algorithmus erweitern kann. Ich möchte hierfür zwei mögliche Lösungsansätze geben:

Die erste Möglichkeit wäre es sich für einige andere Datentypen z.B. Bäume eine Funktion  $length^{-1}$  zu überlegen und diese in den Algorithmus einzubauen. In diesem Fall müssen diese Datentypen eingebaute Datentypen sein. Dieser Ansatz löst also das Problem für benutzerdefinierte Datentypen nicht.

Eine andere Variante ist es die Annotationen auf Datentypen zu erweitern und für jeden benutzerdefinierten Datentyp eine  $length^{-1}$ -Funktion anzugeben. Bei diesem Ansatz muss vor allem auch überlegt werden in wie weit die Datenflussanalyse angepasst werden muss, da es möglich ist mehrere Elemente der gleichen Länge in einem Datentyp zu haben. Ein Beispiel dafür sind Bäume, da diese bei gleicher Größe unterschiedliche Formen haben können.

Ein weiteres Problem der Zeitkomplexitätsanalyse ist es, dass nicht für alle Funktionen ermittelbar ist, welche Größe ihr Ergebnis hat. Dies führt dazu, dass sobald eine solche Funktion als Parameter auftaucht die gesamte Analyse nicht möglich ist. Um dieses Problem zu lösen, wäre eine Analyse für die Größe der Ergebnisse sinnvoll. Diese, inklusive der dazugehörigen Annotationen, müsste zu dem Analysewerkzeug hinzugefügt werden. Anschließend kann die Zeitkomplexitätsanalyse auf diese zugreifen und die Ergebnisse bei der Analyse verwenden.

## Literatur

- [1] MYCROFT, Alan: The Theory and Practice of Transforming Call-by-need into Call-by-value. In: *Proceedings of the Fourth 'Colloque International sur la Programmation' on International Symposium on Programming*. London, UK : Springer-Verlag, 1980. – ISBN 3-540-09981-6, S. 269–281
- [2] BURN, G. L. ; HANKIN, C. L. ; ABRAMSKY, S.: The theory of strictness analysis for higher order functions. In: *on Programs as data objects*. New York, NY, USA : Springer-Verlag New York, Inc., 1985. – ISBN 0-387-16446-4, S. 42–62
- [3] WEGBREIT, Ben: Mechanical program analysis. In: *Commun. ACM* 18 (1975), Nr. 9, S. 528–539. – ISSN 0001-0782
- [4] ROSENDAHL, Mads: *Automatic Program Analysis*. University of Copenhagen, Denmark, Institute of Datalogy, Diplomarbeit, 1986
- [5] ROSENDAHL, Mads: Automatic complexity analysis. In: *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*. New York, NY, USA : ACM Press, 1989. – ISBN 0-89791-328-0, S. 144–156
- [6] SOMOGYI, Zoltan ; HENDERSON, Fergus ; CONWAY, Thomas: The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. In: *J. Log. Program.* 29 (1996), Nr. 1-3, S. 17–64
- [7] ROHLOFF, Judith: *Statische Programmanalyse und Entwicklung einer Annotationsprache für Funktionale Programme*, Fakultät für Informatik, Technische Universität Berlin, Diplomarbeit, 2007