Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2330

Entwicklung und Implementierung eines nebenläufigen Contraint-Solver für die Points-To-Analyse

Stefan Bühler

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. / Harvard Univ. Erhard Plödereder
Betreuer:	Dipl.-Inf. Steffen Keul
begonnen am:	1. April 2011
beendet am:	1. Oktober 2011
CR-Klassifikation:	F.3.2, F.3.3

Inhaltsverzeichnis

1 Einleitung

1.1 Points-To-Analyse

Die Points-To-Analyse ist eine statische Programmanalyse, um zu bestimmen, auf welche Speicherbereiche ein Zeiger zeigen kann. Dies kann dazu verwendet werden, um die Maybe-Alias Frage zu lösen, d.h. ob zwei Ausdrücke den selben Speicherbereich referenzieren können.

Beispiel 1.1. Nach folgendem C-Code sind `i` und `*p` Aliase für denselben Speicherbereich, der für `i` reserviert wurde.

```
int i, *p = i;
```

Das Ergebnis einer Points-To-Analyse würde ergeben, dass *p* ein Zeiger auf *i* ist.

Manchmal kann die Points-To-Analyse auch die Must-Alias Frage beantworten, d.h. ob zwei Ausdrücke immer den selben Speicherbereich referenzieren. Dies ist der Fall, wenn die Analyse für einen Zeiger nur ein Speicherobjekt als mögliches Ziel ergibt. In der hier vorgestellten Variante der Points-To-Analyse tritt dies nur für globale Speicherobjekte auf, da andere Objekte zusammengefasst werden (siehe unten).

Jede Points-To-Analyse kann dabei im Allgemeinen nur eine Überabschätzung liefern, d.h. die Analyse findet mehr mögliche Points-To-Ziele für einen Zeiger als tatsächlich auftreten:

Beispiel 1.2.

```
int i, j, *p;
p = foo() ? &i : &j;
```

Da das Ergebnis von `foo()` in der statischen Analyse im Allgemeinen nicht berechnet werden kann, wird eine Points-To-Analyse für die möglichen Ziele von `p` immer `i` und `j` angeben müssen (es sei denn, die statische Analyse kann das Ergebnis von `foo()` berechnen), auch wenn zum Beispiel die Menge der Points-To-Ziele von `p`, die tatsächlich auftreten können, nur {`i`} ist.

Insbesondere aber nehmen Points-To-Analysen Vereinfachungen vor, die zu größeren Points-To-Mengen führen. Einige dieser Vereinfachungen werden weiter unten beschrieben.

Die Ergebnisse der Points-To-Analyse können für Optimierungen und weitere statische Code-Analysen verwendet werden, hier einige Beispiele:

Konstantenpropagierung

Wenn ein Zeiger nur ein Speicherobjekt als Ziel haben kann, so kann jede Dereferenzierung des Zeiger statisch berechnet werden; dies ist nicht nur für einfache Variablenzugriffe interessant, sondern auch beim Aufruf von Funktionen über Funktionszeiger.

In objektorientierten Sprachen ist dies besonders interessant - Dispatching wird häufig mit mindestens zwei Dereferenzierungen implementiert, die entfallen können, wenn der dynamische Typ eines Objekts in der statischen Analyse bestimmt werden kann.

Beispiel 1.3. In folgendem Beispiel kann der Aufruf `a->foo()` direkt als Aufruf von `B::foo(a)` übersetzt werden, da die Points-To-Mengen für `a->_vtable` und `B::_vtable->foo` jeweils nur ein Element enthalten.

Die angegeben Entsprechungen mit Hilfe von `_vtable` basieren nur auf einem vereinfachtes Modell.

```
class A {
public:
   virtual void foo() { /* ... */ }
};
/* entspricht: */
   void A::foo(A* this) { /* ... */ }
   A::_vtable = { foo: &A::foo }
class B : public A {
public:
   virtual void foo() { /* ... */ }
};
/* entspricht: */
   void B::foo(B* this) { /* ... */ }
   B::_vtable = { foo: &B::foo }
int main() {
   A *a = new B();
   /* entspricht: */
      a = malloc(sizeof(B));
      a->_vtable = &B::_vtable;
   a->foo();
   /* entspricht: */
      (a->_vtable->foo)(a);
   return 0;
}
```

Entfernung von nicht ereichbarem Programmcode

Durch die zusätzliche Konstantenpropagierung können auch weitere Unterprogramme gefunden werden, die nicht mehr vom Hauptprogramm erreichbar sind und darum entfernt werden können. Dies kann wiederum die Points-To-Analyse genauer machen.

Im vorigen Beispiel kann nach der Konstantenpropagierung die Funktion `A::foo` aus dem Programm eliminiert werden, da sie nicht aufgerufen wird.

Caching in Registern

Wenn durch die Points-To-Analyse sichergestellt werden kann, dass ein Speicherbereich während der Ausführung eines Programmblocks nicht verändert wird, so kann dessen Inhalt in Registern zwischengespeichert werden, ohne dass er nach dem Block neu geladen werden muss.

Beispiel 1.4. Ohne Points-To-Analyse muss im folgenden Code der Wert von `j` bei jeder Verwendung neu gelesen werden, da `j` durch das Zuweisen an `*q` bzw. `*p` verändert worden sein könnte. Mit Points-To-Analyse ist klar, an welchen Stellen `j` verändert bzw. nicht verändert wurde, so dass `j` nach `*p = j + 1` nicht neu aus dem Speicher gelesen werden muss.

```
int foo() {
    int i = 1, j = 2, *p = &i, *q = &j;
    *q = 4;
    *p = j+1;
    return j;
}
```

In den meisten Sprachdefinitionen darf der Compiler annehmen, dass die anderen Threads nichts verändern, solange nicht spezielle Sprachkonstrukte wie „volatile“ Speicherbereiche und Memory-Barriers etwas anderes vorschreiben.

Signal-Handling (das Unterbrechen des Programms, um auf bestimme Ereigisse zu reagieren) wird hierfür wie die Nebenläufigkeit behandelt, d.h. im Normalfall ignoriert.

1.2 Eigenschaften verschiedener Points-To-Analysen

Es gibt verschiedene Points-To-Analysen [Plo08] [And94]; ein höherer Aufwand in der Analyse liefert in der Regel präzisere Ergebnisse, d.h. kleinere Points-To-Mengen.

Die hier in dieser Arbeit betrachtete Points-To-Analyse hat folgende Eigenschaften:

Flussinsensitiv

Die Reihenfolge der Zuweisungen wird komplett ignoriert, und die Points-To Mengen werden unabhängig von Positionen im Programmcode berechnet:

Beispiel 1.5. Im Folgenden Beispiel wird `{i, j}` als Points-To Menge für `p` berechnet, auch wenn klar ist, dass abhängig von der Position im Programmcode genauere Angaben möglich wären.

```
int i, j, *p;
p = &i;
*p = 1;
p = &j;
```

Kontextinsensitiv

Der Aufrufkontext einer Funktion wird nicht beachtet, d.h. alle Aufrufe werden zusammengefasst.

Beispiel 1.6.

```
void assign(int **x, int *y) {
   *x = y;
}

int main() {
   int i, j, *p, *q;
   assign(&p, &i);
   assign(&q, &j);
   return 0;
}
```

Der folgende Speicherausschnitt mit den Ergebnissen der Points-To-Analyse zeigt, dass die Points-To-Mengen der Parameter für `assign` die Ziele für beide Aufrufe enthalten, und sie deshalb in der Analyse nicht mehr auseinandergehalten werden können:

Index	Kontext	Beschreibung	Points-To-Menge
10	assign	Parameter x	{26, 27}
11	assign	Parameter y	{24, 25}
24	main	Variable i	∅
25	main	Variable j	∅
26	main	Variable p	{24, 25}
27	main	Variable q	{24, 25}

Durch die Kontextinsensitivät kann, wie im Beispiel ersichtlich, die Präzision der Analyse schlechter werden. Es kann daher sinnvoll sein, häufig verwendete oder kleine Funktion

durch Inlining kontextsensitiv zu analysieren. Dies funktioniert nur endlich oft bei rekursiven Aufrufen.

Kontextinsensitiv bedeutet hier auch, dass nur je eine „Instanz“ von lokalen Variablen betrachtet wird; lokale Variablen werden also als statische (globale) Variablen modelliert.

Gerichtet

Die Datenfluss-Richtung bei Zuweisungen wird beachtet.

Beispiel 1.7.

```
int i, j, *p, *q;
p = &i;
q = &j;
p = q;
```

Die Points-To-Menge für `q` ist nur {`j`}. In einer ungerichteten Analyse würde die Anweisung `p = q` zusätzlich auch als `q = p` aufgefasst werden, und die Points-To-Menge für `q` würde auch `i` enthalten. Eine ungerichtete Analyse liefert unpräzisere Ergebnisse, kann aber auch mit deutlich schnelleren Algorithmen implementiert werden.

Struktursensitiv

Die Komponenten von Objekten werden als unterschiedliche Speicherfelder betrachtet (in einer strukturinsensitiven Analyse würden alle Felder eines Objektes zu einem zusammengefasst).

Feldinsensitiv

Im Gegensatz zu Strukturen werden die Felder eines Arrays zu einem zusammengefasst. Der Grund dafür ist, dass bei Arrayzugriffen der Index zum Zeitpunkt der statischen Analyse meist nicht genauer bestimmt werden kann. Dann müssten alle möglichen Indizes eingesetzt werden, was in der Analyse äquivalent zur Zusammenfassung der Felder ist.

Ein weiteres Problem ist die Modellierung von Arrays mit variabler Größe in einem endlichen Speichermodell.

Beispiel 1.8.

```
int* arr[3], i;
arr[foo()] = &i;
```

Im Gegensatz zu Komponenten von Strukturen, die einen festen bekannten Offset haben, ist hier nicht klar, auf welches Feld des Arrays zugegriffen wird - also müssen für eine feldsensitive Points-To-Analyse alle Möglichkeiten eingesetzt werden, d.h. `arr[0] = arr[1] = arr[2] = &i;`.

In einer feldinsensitiven Analyse wird der Index einfach weggelassen, das obige Programm also folgendermaßen interpretiert:

```
int* arr, i;
foo();
arr = &i;
```

Der Aufruf von `foo()` sollte im Allgemeinen nicht komplett entfernt werden, da er je nach Analyse wichtige Seiteneffekte haben kann. Insbesondere ist die Funktion noch erreichbar, und darf nicht als unereichbarer Code entfernt werden.

Weitere Eigenschaften

Programme können in der Theorie beliebig viel Speicher auf dem Heap anlegen; auch wenn der Speicher auf der Maschine begrenzt ist, ist dies in der Analyse natürlich nicht zu verarbeiten. Darum werden Objekte auf dem Heap nach der Position im Programmcode identifiziert, an der sie angelegt werden (in C also die Stelle des `malloc()` Aufrufs).

Beispiel 1.9.

```
struct A {
    int *p;
};
struct A *newA(void) {
    return malloc(sizeof(struct A));
}
int main() {
    int i, j;
    struct A *x = newA(), *y = newA();
    x->p = &i;
    y->p = &j;
    return 0;
}
```

Da die Objekte `*x` und `*y` an der selben Stelle im Quellcode mit einem Aufruf von `malloc` angelegt wurden, wird in der Points-To-Analyse dasselbe Objekt verwendet. Der Quellcode wird also zu folgendem transformiert:

```
struct A {
    int *p;
};
struct A *newA(void) {
```

```
    static struct A heap_object_A;
    return &heap_object_A;
}
int main() {
    int i, j;
    struct A *x = newA(), *y = newA();
    x->p = &i;
    y->p = &j;
    return 0;
}
```

`x` und `y` zeigen also auf dasselbe Objekt `heap_object_A`, `x->p` und `y->p` verwenden dasselbe Speicherfeld und die Points-To-Menge für dieses Speicherfeld ist {`i`, `j`}.

Es sind auch andere Strategien denkbar, um Objekte auf dem Heap durch endlichen Speicher zu modellieren - als einfachste Variante mit einem einzigen Heap-Objekt, das natürlich groß genug sein muss, um jedes beliebige Objekt zu fassen. Oder mit einem Heap-Objekt für jeden Typ, für den Objekte auf dem Heap angelegt werden.

1.3 Constraint-Solver

Die Aufgabe der Studienarbeit ist eine Neuimplementierung eines Constraint-Solver. Der Constraint-Solver bekommt als Eingabe eine Menge von abstrakten Objekten und Zuweisungen, und soll daraus möglichst kleine Points-To-Mengen berechnen, die die Constrains in Form der Zuweisungen erfüllen.

Die existierende Implementierung des Constraint-Solvers wird von einem Programm verwendet, das als Eingabe die Zwischendarstellung eines C-Programms bekommt und daraus die Eingabe für den Constraint-Solver generiert. Dieses Programm wird ohne große Änderungen weiterverwendet, nur der Constraint-Solver wird ausgetauscht.

Dieses Programm wird auch für den Benchmark verwendet, und einige Beispiele enthalten eine aufbereite Form der Ausgabe dieses Programms.

Die Neuimplementierung soll die parallelisierte Ausführung auf einer variablen Anzahl an Prozessoren unterstützen. Außerdem soll nach Zyklen in den „Copy_To“-Constraints gesucht werden. In einem solchen Zyklus sind die Points-To-Mengen äquivalent, und die Speicherfelder in einem solchen Zyklus können (für die meisten Berechnungen, siehe Kapitel 2) vereinigt werden. Die Hoffnung ist, dass eine solche Zyklenkontraktion Rechenzeit einspart, und die Neuimplementierung soll diese umsetzen.

2 Positive Set-Constraint-Probleme

Das Ziel ist, die Points-To-Analysen auf Set-Constraint-Probleme zurückzuführen. In diesem Kapitel sollen die formalen Grundlagen dafür erarbeitet werden.

In diesen Set-Constraint-Problemen soll für jede Variable in einer Menge von Variablen V eine Teilmenge der Variablen gefunden werden soll. Eine Lösung $L : V \to 2^V$ muss dazu spezielle Bedingungen (Constraints) erfüllen; die Übersetzung in die Points-To-Analyse liefert dann für jede Variable x ein Menge von Variablen, auf die x ein Zeiger sein könnte.

Für die Points-To-Analyse sind dabei insbesondere „kleine“ Lösungen interessant, also Lösungen $L : V \to 2^V$ mit kleinen Mengen $L(v)$.

2^V steht dabei für die Potenzmenge von V, also die Menge aller Teilmengen von V:

$$2^V := \{U \mid U \subseteq V\}$$

Definition 2.1. Ein Tupel $P = (V, C)$ heißt positives Set-Constraint-Problem, wenn V eine endliche Menge von Variablen ist und $C \subseteq E$ eine endliche Teilmenge von einfachen und erweiterten Constraints B bzw. E':

$$B := \{“x \in y“, “x \subseteq y“ \mid x, y \in V\}$$

$$E' := \{(v, f) \mid k > 0, v \in V^k, f : V^k \to B\}$$

$$E = B \cup E'$$

Die erweitertern Constraints dienen dazu Constraints zu modellieren, die aus Zuweisungen mit Derefenzierungen stammen; dazu werden aus Elementen aus Lösungsmengen $L(v)$ neue Constraints konstruiert. Die genaue Semantik dieser Constraints wird weiter unten in den Anforderungen an eine Lösung gegeben.

Das Problem heißt „positiv“, da die Constraints so ausgelegt werden können, dass sie nur fordern, dass gewisse Elemente in Menge enthalten sind - es wird nicht verlangt, dass bestimmte Elemente nicht enthalten sind. Darauf folgt insbesondere, dass es immer die triviale Lösung gibt, in der jede Menge alle Elemente enthält, d.h. $L(v) := V \forall v \in V$.

Definition 2.2. Sei $L : V \to 2^V$ ein „Lösungsversuch“ für das Problem (V, C):

- Für $v \in V^k : l(L, v) := \{(x_1, \ldots, x_k) \mid x_i \in L(v_i)\}$
  (Dies sind alle Kombinationen von Points-To-Zielen für ein Tupel von Variablen.)

- Das Auswerten der erweiterten Constraints zu einfachen Constraints:

$$\begin{aligned}\hat{S}_L &: E \to 2^B \\ \hat{S}_L(\text{“}x \in y\text{“}) &:= \{\text{“}x \in y\text{“}\} \\ \hat{S}_L(\text{“}x \subseteq y\text{“}) &:= \{\text{“}x \subseteq y\text{“}\} \\ \hat{S}_L((v,f)) &:= \{f(x) \mid x \in l(L,v)\} \\ S_L &: 2^E \to 2^B \\ S_L(C) &:= \bigcup_{c \in C} \hat{S}_L(c)\end{aligned}$$

  Auf Teilmengen von einfachen Constraints ist $S_L|_{2^B}$ also die Identität.

- L heißt Lösung von (V, C), wenn die Constraints $S_L(C)$ für L erfüllt sind, d.h:
  - $\forall \text{“}x \in y\text{“} \in S_L(C) : x \in L(y)$
  - $\forall \text{“}x \subseteq y\text{“} \in S_L(C) : L(x) \subseteq L(y)$
- Sei $L' : V \to 2^V$ ein weiterer Lösungsversuch. $L \sqsubseteq L' :\Leftrightarrow \forall v \in V : L(v) \subseteq L'(v)$
  $\sqsubseteq$ definiert damit eine partielle Ordnung auf der Menge aller Lösungsversuche.

Das Ziel ist nun, einfache Constraints zu einem Problem hinzuzufügen, ohne dessen Lösungsmege zu verändern, um eine Menge von Constrains zu erreichen, aus der man eine Lösung ablesen kan.

Satz 2.3. Sei U eine untere Schranke aller Lösungen für das Problem (V, C), dann gilt für alle $L : V \to 2^V$:

$$L \text{ ist Lösung für } (V,C) \iff L \text{ ist Lösung für } (V, C \cup S_U(C))$$

Beweis.

- $\forall C, D : S_L(C \cup D) = S_L(C) \cup S_L(D)$, folgt direkt aus der Definition von S_L.
- $\Rightarrow \forall C \subseteq D : S_L(C) \subseteq S_L(C \cup D) = S_L(D)$
- $\forall C, L_1 \sqsubseteq L_2 : S_{L_1}(C) \subseteq S_{L_2}(C)$, da $\forall v \in V^k : l(L_1, v) \subseteq l(L_2, v)$
- $\Rightarrow S_U(C) \subseteq S_L(C) \Rightarrow S_L(S_U(C)) \subseteq S_L(S_L(C)) = S_L(C)$
- $\Rightarrow S_L(C \cup S_U(C)) = S_L(C) \cup S_L(S_U(C)) = S_L(C)$

Für beide Probleme müssen Lösungen also dieselbe Menge an Constraints erfüllen. □

Definition 2.4. Schrittweise Auswertung der einfachen Teilmengen Constraints:

$$\begin{aligned} T : 2^E &\to 2^E : \\ C &\mapsto C \cup \{\text{“}x \in z\text{“} \mid \text{“}x \in y\text{“}, \text{“}y \subseteq z\text{“} \in C\}\end{aligned}$$

Offensichtlich gilt $C \subseteq T(C)$.

Satz 2.5. Für alle $L : V \to 2^V$ gilt:

$$L \text{ ist Lösung für } (V, C) \iff L \text{ ist Lösung für } (V, T(C))$$

Beweis. „$\Longleftarrow$“: klar.
„$\Longrightarrow$“: Sei L Lösung für (V, C). Betrachte die neuen Constraints $T(C) \setminus C$:

- “$x \in z$“ $\in (T(C) \setminus C)$, wobei “$x \in y$“, “$y \subseteq z$“ $\in C$:
  $\Rightarrow x \in L(y) \wedge L(y) \subseteq L(z)$
  $\Rightarrow x \in L(z)$

L erfüllt also auch diese Constraints, und ist also auch Lösung für $(V, T(C))$. □

Definition 2.6. Zu einer Menge von Constraints $C \subseteq E$ kann man eine untere Schranke $M(C)$ der Lösungen ablesen: $M(C) : V \to 2^V$:

$$M(C)(v) := \{x \mid \text{“}x \in v\text{“} \in C\}$$

Definition 2.7. Die Menge von Constraints in einem Problem kann nun iterativ vergrößert werden:

$$I(C) := T(C \cup S_{M(C)}(C))$$

Die Probleme (V, C) und $(V, I(C))$ sind äquivalent, d.h. sie haben die gleichen Lösungen. Zudem gilt $C \subseteq I(C)$.

Satz 2.8. Die Kette $C_0 := C \subseteq C_1 := I(C_0) \subseteq C_2 := I(C_1) \subseteq \ldots$ wird stationär, d.h. $\exists k \in \mathbb{N} : C^* := C_k = I(C_k) = C_{k+1}$. Das Problem wird dabei nicht verändert, d.h. (V, C) und (V, C^*) sind äquivalent.

Beweis. Die Differenz zweier aufeinanderfolgenden Kettenglieder liegt in B: $D_n := C_{n+1} \setminus C_n \subseteq B$, da I nur Elemente aus B hinzufügt.
Die Mengen D_n sind disjunkt für verschiedene n, da Elemente nur einmal „neu“ hinzukommen können.
Gäbe es nun kein $k : C_k = I(C_k)$, so wären die D_n alle nicht leer. Also gäbe es unendlich viele nicht leere disjunkte Teilmengen D_n von B - dies ist ein Widerspruch, da B endlich ist.

Der zweite Teil des Satzes folgt mit Induktion über k. □

Satz 2.9. Gilt $I(C) = C$, dann ist $M(C)$ die kleinste Lösung von (V, C).

Beweis. $M(C)$ ist auf jeden Fall eine untere Schranke. Bleibt zu zeigen, dass $M(C)$ eine Lösung ist.

$C \cap B \subseteq S_{M(C)}(C) \subseteq T(C \cup S_{M(C)}(C)) \cap B = C \cap B$
$\Rightarrow S_{M(C)}(C) = C \cap B$
$C \subseteq T(C) \subseteq I(C) = C$
$\Rightarrow T(C) = C$

Prüfe nun, ob $M(C)$ alle Constraints $S_{M(C)}(C) = C \cap B$ erfüllt:

- "$x \in y$" $\in (C \cap B) : x \in M(C)(y)$ nach Definition von $M(C)$.
- "$x \subseteq y$" $\in (C \cap B)$:
  $\forall v \in M(C)(x)$: "$v \in x$" $\in C$ nach Definition von $M(C)$
  $\Rightarrow$ "$v \in y$" $\in T(C) = C \Rightarrow v \in M(C)(y)$
  $\Rightarrow M(C)(x) \subseteq M(C)(y)$

Also ist $M(C)$ die kleinste Lösung von (V, C). □

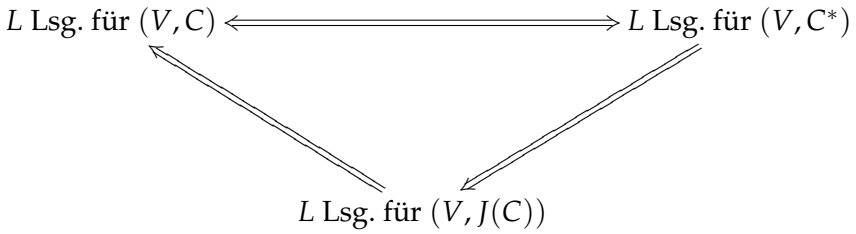
Korollar 2.10. Für ein positives Set-Constraint-Problem (V, C) gibt es eine Vervollständigung zu einem äquivalenten Problem (V, C^*), $C \subseteq C^*$, so dass $M(C^*)$ die kleinste Lösung dieser Probleme ist.

Anmerkung 2.11. Anstatt I kann auch eine andere Iterator-Funktion J verwendet werden, sofern sie folgende Anforderungen erfüllt:

- Monoton und beschränkt: $C \subseteq J(C) \subseteq C^*$
- Erreicht die Schranke: $C \subsetneq I(C) \Rightarrow C \subsetneq J(C)$

Eine solche Iterator-Funktion hat nämlich folgende Eigenschaften, die oben verwendet wurden:

- (V, C) und $(V, J(C))$ sind äquivalente Probleme:



- Die Kette $C \subseteq J(C) \subseteq J^2(C) \ldots$ wird stationär, d.h. $\exists k \in \mathbb{N} : J^{k+1}(C) = J^k(C)$; das k ist natürlich von der verwendeten Iteratorfunktion J abhängig, aber der erreichte Fixpunkt ist der gleiche: $C^* = J^k(C)$:
  Da $J(C)$ durch C^* beschränkt ist, kann kein Kettenglied größer sein. Da $J(C) \supsetneq C$ ist, solange $I(C) \supsetneq C$ ist, kann kein kleinerer Fixpunkt erreicht werden.
  Mit demselben Endlichkeitsargument wie oben folgt jedoch die Existenz.

Es wird also dieselbe Vervollständigung (V, C^*) erreicht, und damit auch die kleinste Lösung $M(C^*)$ gefunden.

2.1 Optimierungen

Unter bestimmten Voraussetzungen können Ersetzungen und andere Transformationen vorgenommen werden, die das Problem vereinfach können.

Definition 2.12. Definition des Ersetzungsoperator $[y/z]$ für $y, z \in V$.

Auf V^k für $k = 1$:

- $(y)[y/z] := (z)$
- $x \neq y : (x)[y/z] := (x)$

Auf V^k für $k > 1$:

- $(v_1, v_2, \ldots, v_k)[y/z] := (v_1', v_2', \ldots, v_k')$, wobei $(v_1') = (v_1)[y/z]$ und $(v_2', \ldots, v_k') = (v_2, \ldots, v_k)[y/z]$

Auf einfachen Constraints:

- "$x \in y$"$[y/z] :=$ "$x \in z$"
- $w \neq y :$ "$x \in w$"$[y/z] :=$ "$x \in w$"
- $x \neq y :$ "$x \subseteq y$"$[y/z] :=$ "$x \subseteq z$"
- $x \neq y :$ "$y \subseteq x$"$[y/z] :=$ "$z \subseteq x$"
- "$x \subseteq x$"$[x/z] :=$ "$z \subseteq z$"
- $x, w \neq y :$ "$x \subseteq w$"$[y/z] :=$ "$x \subseteq w$"

Auf Mengen elementweise:

- $C[y/z] := \bigcup_{c \in C} c[y/z]$

Auf erweiterten Constraints:

- $k > 0, v \in V^k, f : V^k \to B :$
  $f' : V^k \to B : x \mapsto (f(x))[y/z]$
  $(v, f)[y/z] := (v[y/z], f')$

y wird damit an fast allen Stellen ersetzt, nur nicht auf der linken Seite in "$y \in x$". Bei der Entscheidung, ob $L : V \to 2^V$ eine Lösung für $(V, C[y/z])$ ist, werden dabei im Vergleich zu (V, C) alle Vorkommen von $L(y)$ durch $L(z)$ ersetzt.

Satz 2.13. Wenn für zwei verschiedene Variablen $x, y \in V$ für jede Lösung L von (V, C) gilt, dass die Mengen $L(x)$ und $L(y)$ gleich sind, dann kann (V, C) zu einem ähnlichen Problem $(V, C[x/y])$ transformiert werden. Eine Lösung L für (V, C) ist dabei auch eine Lösung für $(V, C[x/y])$, denn mit $L(x) = L(y)$ erfüllt L auch alle Bedingungen, in denen $L(x)$ durch $L(y)$ ersetzt wurde.

Ist L eine Lösung für $(V, C[x/y])$, so ist $L' : v \mapsto \begin{cases} L(y) \text{ für } v = x \\ L(v) \text{ sonst} \end{cases}$ eine Lösung für (V, C), denn alle Anforderungen an $L'(x)$ in (V, C) wurden von $L(y) = L'(x)$ in $(V, C[x/y])$ bereits erfüllt.

Durch solche Transformationen kann das Problem kleiner werden, da z. Bsp. verschiedene Constrains zu einem werden können. Außerdem muss eine Menge $L(x)$ weniger berechnet werden; im transformierten Problem spielt sie keine Rolle.

Satz 2.14. Existieren paarweis verschiedene Variablen $v_1, \ldots, v_n \in V, n > 1$, so dass für $1 \leq i < n$: "$v_i \subseteq v_{i+1}$" $\in C$ und "$v_n \subseteq v_1$" $\in C$, so gilt für jede Lösung L von (V, C):

$$L(v_1) = \ldots = L(v_n)$$

Beweis. Folgt direkt aus $L(v_1) \subseteq L(v_2) \subseteq \ldots \subseteq L(v_n) \subseteq L(v_1)$. □

Solche Zyklen können mit dem Algorithmus von Tarjan zur Bestimmung starker Zusammenhangskomponenten [Tar72] gefunden werden; dazu wird der Graph $(V, \{(x, y) \mid "x \subseteq y" \in C\})$ benötigt. Die Menge C der dafür verwendeten Constraints muss dabei nicht notwendigerweise von der verwendeten Iteratorfunktion durchlaufen werden, es kann auch jede andere Menge von Constraints C' mit $C' \subseteq C^*$ verwendet werden.
In der Praxis bedeutet dies, dass die Zyklen parallel zum Iterieren berechnet werden können und kein „konsistenter" Schnappschuss der Constraints benötigt wird.

3 Constraint-Generierung

3.1 Speichermodell

Jedes Speicherfeld wird durch eine natürliche Zahl identifiziert; es werden nur endlich viele Speicherfelder unterschieden.

Ein Objekt mit mehreren Felder belegt aufeinanderfolgende Speicherfelder, die Komponenten werden durch den Offset zum ersten Feld oder durch einen relativen Offset zu einem vorigen Feld identifiziert. Die abstrakte Darstellung kennt keine Unterobjekte von Objekten; jedes Speicherfeld ist also genau einem Objekt zugeordnet. Mit Hilfe dieser Zuordnung kann später geprüft werden, ob ein Komponentenoffset nicht über ein Objekt hinausgeht und damit gültig ist.

Eine Funktion wird ähnlich wie ein Objekt modelliert; die erste Komponente steht für die Funktion selbst (um Zeiger auf die Funktion modellieren zu können), dann folgen der Rückgabewert und die Parameter. Die einzelnen Komponenten dieses Funktionsobjekts können wiederum normale Objekte sein. Jedes Speicherfeld kann also zusätzlich zur Zuordnung zu einem Objekt auch eine Zuordnung zu einer Funktion enthalten.

Eine Dereferenzierung trägt eine zusätzliche Information, die angibt, ob das derefenzierte Objekt als Funktionsobjekt betrachtet werden soll oder nicht; dementsprechend fallen Überprüfungen, ob der Offset für eine Komponente gültig ist, anders aus.

3.2 Zuweisungen

Es gibt acht Zuweisungsmuster, die wie folgt als Constraints interpretiert werden. Für jede Zuweisung wird dabei immer geprüft, ob die Komponentenoffsets für die jeweiligen Objekte gültig ist, also nicht über die Objektgrenzen hinausgeht. Ist einer der Offsets nicht gültig, wird eine Warnung ausgegeben und die Zuweisung ignoriert.

Für Zuweisungen mit Dereferenzierung kann die entsprechende Prüfung erst stattfinden, nachdem ein Points-To-Ziel eingesetzt wurde. Da die formale Definition der erweiterten Constraints immer die Rückgabe eines einfachen Constraints verlangt, kann man sich hier die Rückgabe eines trivialen Constraints vorstellen („$0 \subseteq 0$“), wenn der Komponentenoffset für das eingesetzte Ziel nicht gültig ist und die Zuweisung daher ignoriert werden soll. Dies kann beim Casten in der Objektorientierung auftreten, das normalerweise durch eine Typprüfung geschützt ist.

Zusätzlich können weitere Überprüfungen vorgenommen werden, zum Beispiel ob ein Speicherfeld überhaupt ein Zeiger ist und eine nicht-leere Points-To-Menge enthalten darf.

Die acht Zuweisungsmuster sind entweder einfache Kopieranweisungen oder das Speichern einer Adresse in einem Speicherfeld. Dementsprechend führen die Zuweisungen entweder zu Constraints, die fordern, dass die Points-To-Menge der rechten Seite Teilmenge der Points-To-Menge der linken Seite ist bzw. dass die Points-To-Menge der linken Seite die rechte Seite enthält.

- `X.a = Y.b`: "$(Y + b) \subseteq (X + a)$"
- `X.a = Y->b`: $((Y), f), f : V \to B : y \mapsto$ "$(y + b) \subseteq (X + a)$"
- `X->a = Y.b`: $((X), f), f : V \to B : x \mapsto$ "$(Y + b) \subseteq (x + a)$"
- `X->a = Y->b`: $((X, Y), f), f : V^2 \to B : (x, y) \mapsto$ "$(y + b) \subseteq (x + y)$"
- `X.a = & Y.b`: "$(Y + b) \in (X + a)$"
- `X.a = & Y->b`: $((Y), f), f : V \to B : y \mapsto$ "$(y + b) \in (X + a)$"
- `X->a = & Y.b`: $((X), f), f : V \to B : x \mapsto$ "$(Y + b) \in (x + a)$"
- `X->a = & Y->b`: $((X, Y), f), f : V^2 \to B : (x, y) \mapsto$ "$(y + b) \in (x + y)$"

3.3 Beispiele

Beispiel 3.1.

```
int* foo(int *x) {
    return x;
}

int main() {
    int* (*bar)(int *x);
    int i;
    int *q;

    bar = foo;
    q = bar(&i);
    return 0;
}
```

Generierte Zuweisungen: („formale" Dereferenzierungen beziehen sich auf Funktionsobjekte)

- `8.0 = 9.0`
- `22.0 = & 7.0`

- `(formal) 22->2 = & 20.0`
- `21.0 = (formal) 22->1`

Die „formalen“ Dereferenzierungen werden also benötigt, wenn Funktionen über Funktionszeiger aufgerufen werden.

Speicherausschnitt mit den Ergebnissen der Points-To-Analyse:

Index	Kontext	Beschreibung	Points-To-Menge
7	foo	Abstrakte Funktion	∅
8	foo	Rückgabewert	{20}
9	foo	Parameter x	{20}
20	main	Variable i	∅
21	main	Variable q	{20}
22	main	Variable bar	{7}

3.4 Probleme

C-Programme sind nicht besonders gut geeignet als Gegenstand der Points-To-Analyse. Es gibt viele Konstrukte in C, die nicht sinnvoll zu analysieren sind.

Das offensichtlichste Problem ist, dass in C Zeigerarithmetik erlaubt ist. Es gibt allerdings noch einige weitere Probleme, die nicht auf den ersten Blick zu erkennen sind.

Memcpy

`memcpy` wird normalerweise in Assembler programmiert, um optimale Performance zu erzielen. Eine äquivalente Implementierung in C könnte etwa so aussehen:

```
void *memcpy(void *s1, const void *s2, size_t n) {
    char *c1 = s1;
    const char *c2 = s2;
    int i;
    for (i = 0; i < n; i++) c1[i] = c2[i];
    return s1;
}
```

Eine solche Implementierung hat das Problem, dass die Constraint-Generierung nur eine Zuweisung `*s1 = *s2;` ausgibt, und damit nur die erste Komponente von Objekten kopiert - der Index fällt bei Arrayzugriffen weg.

Beispiel 3.2. In der aktuellen Implementierung der Constraint-Generierung wird `memcpy` komplett ausgelassen. Als Folge davon ist in folgendem Beispiel die Points-To-Menge von `y` leer anstatt `a` zu enthalten.

```
int main() {
    int a = 1;
    int *x = &a, *y = 0;

    memcpy(&y, &x, sizeof(int*));

    return 0;
}
```

Wenn `memcpy` verwendet wird, um den Inhalt eines Objekt in ein anderes Objekt vom selben Typ zu kopieren, könnte der obige Aufruf in der Analyse durch `y = x;` ersetzt werden.

Casting von Zeigern auf Strukturen

In C sind viele Strukturen in gewisser Hinsicht kompatibel; auch wenn die Semantik für diese Programme nach dem Standard evtl. undefiniert ist, treten ähnliche Beispiele in der Praxis durchaus auf, da sie mit den üblichen Compilern funkionieren. Zum Beispiel kann ein Zeiger auf eine `struct sockaddr` in einen Zeiger auf `struct sockaddr_in` konvertiert werden, da die erste Struktur normalerweise groß genug ist, dass die zweite Platz darin hat.

Beispiel 3.3. Folgendes Beispiel zeigt das Problem. Die Constraint-Generierung reserviert für `addr` nur zwei Speicherfelder (eins für `sa_family` und eins für das `char`-Array `sa_data`). Die Komponente `sin_addr.s_addr` ist allerdings das dritte Speicherfeld in `struct sockaddr_in`, und die Constraint-Generierung versucht auf eine nicht existierende Komponente zuzugreifen.

```
int main() {
    struct sockaddr addr;

    in_addr_t *p = &((struct sockaddr_in *)&addr)->sin_addr.s_addr;

    return 0;
}
```

Beispiel 3.4. Ein weiteres Problem wird von `unions` und der Feldinsensitivität ausgelöst:

```
struct A {
    int *x, *y;
};

struct B {
    int *ptrs[2];
};

union C {
```

```
    struct A a;
    struct B b;
};

int main() {
    int i, j;
    union C c;

    c.a.x = &i;
    c.a.y = &j;
    int *p = c.b.ptrs[1];

    return 0;
}
```

Die Points-To-Menge von `p` sollte auf `j` zeigen, aber sie zeigt auf `i` - das Array in Struktur `B` ist nur ein Speicherfeld groß, und alle Array-Einträge entsprechen also der Komponente `x` in `struct A`.

Negative Komponentenoffsets

Die Constraint-Generierung sieht bislang nicht vor mit negativen Komponentenoffsets zu arbeiten. Allerdings könnte dies an einigen Stellen nützlich sein, zum Beispiel um das umgebende Objekt für ein Teilobjekt zu finden (gerade im Linux-Kernel erfreut sich das container_of() Macro [KH05], das zu einem Zeiger auf eine Komponente eines Objekts den Zeiger auf das Objekt zurückgibt, großer Beliebtheit).

4 Algorithmus

Die Grundidee des Algorithmus ist aus Kapitel 2 bereits bekannt: ausgehend von einer Menge an Constraints wird diese iterativ um Constraints erweitert, die Schlussfolgerungen aus bereits bekannten Constraints sind, so dass das Problem an sich nicht verändert wird. Am Ende können dann aus dieser Constraint-Menge die Points-To-Mengen ausgelesen werden.

Es ist jeoch ungünstig, die Menge dieser Constraints als einfache Menge abzuspeichern. Deutlich sinnvoller ist es, diese Menge nach bestimmten Kriterien zu ordnen und in Teilmengen zu zerlegen. Insbesondere ist wichtig, dass der Algorithmus nicht die komplette Menge an Constraints absuchen muss, um für die Iteration neue Constraints zu finden, sondern dass er weiß, an welchen Stellen er noch suchen muss.

Außerdem ist gewünscht, dass der Algorithmus auch mehrere Prozessoren nutzen kann; es sollen also mehrere Threads laufen, die parallel nach neuen Constraints suchen.

Damit sich die verschiedenen Threads sich nicht gegenseitig blockieren, wird die Arbeit auf Speicherfelder verteilt. Jedesmal, wenn ein neuer Constraint gefunden wird, wird er einem Speicherfeld zugeordnet und das Speicherfeld auf die Todo-Queue gesetzt. Wenn ein Thread später dann dieses Speicherfeld abarbeitet, hat er eine Menge von neuen Constraints, die er in die abstrakte Menge aller Constraints einarbeitet, und dabei weitere neu gefundene Constraints wieder auf Speicherfelder verteilt.

Das Abarbeiten eines Speicherfeldes kann einige Zeit in Anspruch nehmen, deswegen darf dieses nicht die Zuordnung neuer Constraints zu diesem Speicherfeld blockieren. Stattdessen extrahiert ein Thread am Anfang der Abarbeitung alle neuen Constraints für das jeweilige Speicherfeld; danach können wieder neue hinzukommen, die erst in einem späteren Durchlauf bearbeitet werden.

Für jede Art von Constraints gibt es drei Möglichkeiten der Verwaltung:

- Es ist bekannt, dass es keine neuen Constraints geben kann, und die bekannten werden nur intern in der Abarbeitung eines Speicherfeldes benötigt.
- Es gibt eine Liste von eventuell neuen Constraints, und eine interne Liste von bereits abgearbeiteten Constraints. Beim Abarbeiten wird geprüft ob der jeweilige Eintrag tatsächlich neu oder bereits bekannt ist, und bereits bekannte werden übersprungen um Endlosschleifen zu vermeiden.
- Es gibt eine Liste von bereits gesehenen (aber evtl. noch nicht abgearbeiteten) Constraints, eine für wirklich neue und eine interne von bereits abgearbeiteten. Bevor ein Eintrag in die Liste für neue Constraints kommt, wird geprüft, ob der Eintrag bereits

gesehen wurde. Diese Überprüfung muss entsprechend schnell machbar sein, da sonst die Threads sich gegenseitig zu lange blockieren können.

Ein Speicherfeld kann nur von einem Thread gleichzeitig abgearbeitet werden, anderere Threads müssen warten; dies stellt aber kein Problem dar. Die Verwaltung in der Todo-Queue stellt sicher, dass die Threads versuchen verschiedene Speicherfelder zu bearbeiten, solange genügend Speicherfelder zum Abarbeiten markiert sind. Wenn nicht mehr viele Speicherfelder zu bearbeiten sind, ist die gleichzeitige Auslastung aller Threads nicht mehr besonders relevant, da der Algorithmus dann so gut wie fertig ist.

4.1 Kategorisierung der Constraints

Um die Constraints sinnvoll abspeichern zu können, werden sie in folgende Kategorien sortiert und jeweils einem Speicherfeld zugeordnet.

- `X.a = & Y.b`: „Points_To“ (von $X + a$ nach $Y + b$), wird im Speicherfeld $X + a$ gespeichert. Diese Constraints sind auch genau diese, die am Ende die Lösung in Form der Points-To-Mengen definieren.
- `X.a = Y.b`: „Copy_To“ (von $Y + b$ nach $X + a$), wird im Speicherfeld $Y + b$ gespeichert, da für neue „Points_To“ Ziele diese an die „Copy_To“ Ziele propagiert werden müssen.
- `X->a = Y.b, X->a = & Y.b`: „LHS_Deref“, werden im Speicherfeld X gespeichert, da X für neue „Points_To“ Ziele derefenziert werden muss. Für Points-To-Ziele x von X werden dann Constraints `x.a = Y.b` bzw. `x.a = & Y.b` erzeugt.
- `X.a = Y->b, X.a = & Y->b`: „RHS_Deref“, werden im Speicherfeld Y gespeichert. Für Points-To-Ziele y von Y werden dann Constraints `X.a = y.b` bzw. `X.a = & y.b` erzeugt.
- `X->a = Y->b, X->a = & Y->b`: Kann prinzipiell im Speicherfeld X oder Y gespeichert werden. Der Algorithmus implementiert hier eine schrittweise Einsetzung der „Points_To“ Ziele, und beginnt dabei auf der rechten Seite, daher werden diese Constraints als „RHS_Deref_Both“ im Speicherfeld Y gespeichert. Nach Einsetzen eines Y „Points_To“ Zieles bleibt ein „LHS_Deref“ Constraint übrig, der im Speicherfeld X gespeichert wird.

4.2 Abarbeitung neuer Constraints

Nachdem einem Speicherfeld neue Constraints zugeordnet wurden, muss einer der Threads diese abarbeiten. Da beim Einsetzen keine neuen „RHS_Deref“- bzw. „RHS_Deref_Both“-Constraints estehen können, bleiben folgende Fälle. Die neuen Constraints werden dabei in die jeweiligen Mengen der bereits abgearbeiteten Constraints eingefügt.

Die dabei neu generierten Constraints sind alle, die in einem Schritt des Iterators aus Kapitel 2 aus den betrachteten Constraints entstehen können, und die Abarbeitung der neuen Constraints erfüllt damit die Anforderungen an einen alternativen Iterator, der zu einer minimalen Lösung des Constraint-Problems führt.

Neue „Points_To“-Ziele

Ein neues „Points_To“-Ziel x für das Speicherfeld X wird in dessen bereits abgearbeiteten „LHS_Deref“-, „RHS_Deref“- und „RHS_Deref_Both“-Constraints eingesetzt und die dabei neu entstehenden Constraints wieder einsortiert.

Außerdem wird es an die „Copy_To“-Ziele von X propagiert, d.h. für ein „Copy_To“-Ziel c wird ein neuer Constraint `c = & x` („Points_To“ von c nach x) eingetragen.

Neue „Copy_To“-Ziele

Bereits abgearbeitete „Points_To“-Ziele werden an neue „Copy_To“ Ziele propagiert.

Neue „LHS_Deref“-Constraints

Bereits abgearbeitete „Points_To“-Ziele werden in die neuen „LHS_Deref“-Constraints eingesetzt.

4.3 Todo-Queue

Jeder Thread bekommt von der Todo-Queue immer wieder ein Speicherfeld zum Abarbeiten zugewiesen, bis es keine mehr gibt. Ein Thread setzt beim Abarbeiten eines Speicherfeldes wieder andere auf die Todo-Queue, wenn er diesen neue Constraints zugeordnet hat.

Als Spezialfunktion löst die Todo-Queue nach jedem Durchlauf durch alle Speicherfelder das Finden und Kontrahieren der „Copy_To“ Zyklen aus.

4.4 Kontraktion der „Copy_To“-Zyklen

Wie am Ende des 2. Kapitels beschrieben, können „Copy_To“-Zyklen kontrahiert werden.

Dazu wird ein Thread beauftragt, anstatt der Abarbeitung eines Speicherfelds diese Zyklen zu suchen und anschließend zu kontrahieren. Zur Suche wird der Tarjan-Algorithmus zum Finden starker Zusammenhangskomponenten [Tar72] in Graphen verwendet, als Kanten werden die „Copy_To“ Constraints verwendet.

Zur Kontraktion eines Zyklus werden jeweils zwei Speicherfelder vereinigt. Dazu werden sowohl die neuen als auch bereits abgearbeiteten Constraint-Mengen des einen Speicherfelds als neue Constraints dem anderen Speicherfeld zugeordnet; da der Algorithmus nicht die Abarbeitung von neuen „RHS_Deref“- bzw. „RHS_Deref_Both“-Constraints vorsieht, werden diese sofort ausgewertet.

Außerdem wird in allen „RHS_Deref_Both“-Constraints der Zeiger der linken Seite durch den äquivalenten neuen Zeiger ersetzt.

5 Implementierung

Da die Implementierung im Rahmen des Projekt Bauhaus, das zentrale Forschungvorhaben der Abteilung Programmiersprachen des Instituts für Softwaretechnologie an der Universität Stuttgart, weiter verwendet und gewartet werden soll, wurde als Implementierungssprache Ada [TDB+06] vorgegeben.

In Ada werden sogenannte „protected types“ verwendet, um den Zugriff auf Daten von mehreren Threads aus zu synchronisieren. Dabei kann nur ein Unterprogramm eines „protected objects“ gleichzeitig aktiv sein. Diese Objekte werden verwendet, um die Abarbeitung der Speicherfelder und das Zuordnen neuer Constraints zu Speicherfeldern zu synchronisieren. Auch die Todo-Queue wird mit Hilfe eines „protected types“ implementiert.

5.1 Datenstrukturen

Die Struktur für ein Speicherfeld (im Quellcode `Pointsto_Object`) ist ein einfacher (nicht synchronisierter) record mit folgenden Komponenten:

- `Prot`: das interne geschützte Objekt, das für die Abarbeitung neuer Constraints verwendet wird. Es enthält die bereits abgearbeiteten Constraints, aus denen am Ende die Points-To-Menge für ein Speicherfeld extrahiert wird. Diese Constraints werden in einfachen, nicht synchronisierten Containern gespeichert (der Zugriff ist bereits durch das umgebende Objekt synchronisiert).
- `Pointsto, Copyto`: in diesen werden sowohl alle bereits gesehen „Points_To“- bzw. „Copy_To“-Ziele als auch die neuen gespeichert. Beim Abarbeiten wird aus diesen Komponenten die Menge der jeweils neuen Constraints extrahiert. Das Einfügen und Extrahieren neuer Einträge muss synchronisiert geschehen, in Bitvektoren kann jedoch ein schneller unsynchronisierter Test vorgeschaltet werden, ob ein Eintrag bereits vorhanden ist.
- `New_LHS_Deref`: die Liste von eventuell neuen „LHS_Deref“-Constraints. Beim Abarbeiten wird geprüft, ob die jeweiligen Einträge tatsächlich neu sind. Das Einfügen und Extrahieren neuer Einträge muss synchronisiert geschehen.
- `SCC_Copyto`: Index des äquivalenten Speicherfelds, das nach einer „Copy_To“-Zyklenkontraktion anstatt diesem Feld verwendet werden soll, und wird mit dem eigenen Index initialisiert. Dieses Feld wird nur geändert, wenn alle anderen Threads keine Speicherfelder bearbeiten (die Synchronisation dazu wird von der Todo-Queue verwaltet).

- Konstante Informationen über das Speicherfeld:
  - `Object_First, Routine_First`: Index des ersten Speicherfeldes für normale Objekte bzw. Funktionsobjekte. Wird verwendet um zu prüfen, ob zwei Speicherfelder zum selben Objekt gehören, und damit ob ein Komponentenoffset gültig ist.
  - `Allows_Address`: gibt an, ob Zeiger auf dieses Speicherfeld zeigen dürfen.
  - `Is_Pointer`: gibt an, ob dieses Speicherfeld ein Zeiger ist.

Dadurch ist der Zugriff auf alle Daten entsprechend synchronisiert und es gibt keine Race-Conditions; jeder neue Constraint, der generiert wird, wird aus sicheren Informationen generiert. Auf der anderen Seite sind in der Abarbeitung neuer Constraints alle Fälle berücksichtigt, durch die weitere neuen Constraints entstehen können.

In den geschützten Containern wird nicht auf externe Daten zugegriffen, diese können also keine Deadlocks auslösen. Das interne geschützte Objekt eines Speicherfeldes greift nur auf die öffentlichen geschützten Kompontenten anderen Speicherfelder zu, und kann deshalb auch zu keinem Deadlock führen.

Nur während der „Copy_To“-Zyklenkontraktion gibt es hier eine Ausnahme: das interne geschützte Objekt eines Speicherfeldes greift auf das eines anderen Speicherfeldes zu, um auf einige Constraints direkt zugreifen zu können. Dieser Zugriff geschieht jedoch immer nur von den Speicherfeldern aus, die zu einem anderen vereinigt werden sollen, auf das Speicherfeld, mit dem sie vereinigt werden sollen. Dabei gibt es also keine Zyklen und damit keine Deadlocks, und in der aktuelle Implementierung ist an dieser Stelle sowieso nur ein Thread aktiv.

5.2 Container

Es werden verschiedene Container benötigt, um die Daten abzuspeichern. Zusätzlich zu den bekannten Containern aus Ada wurden folgende Container implementiert. Im nächsten Kapitel wird untersucht, wie sich die Auswahl verschiedener Container auf die Laufzeit und den Speicherverbrauch auswirkt.

5.2.1 Safe-Set

Dieser Container bietet synchronisierte Operationen für `Ada.Containers.Ordered_Sets`; er unterstützt das synchronisierte Einfügen neuer Elemente und das synchronisierte Extrahieren und Kopieren aller Elemente (entweder in eine synchronisierte und unsynchronisierte Menge). Der Wrapper unterstützt nicht die Aufzählung mit Hilfe eines Cursors.

5.2.2 Ordered-Vectors

Ordered-Vectors sind einige Zusatzmethoden um in `Ada.Containers.Vectors` geordnete Mengen zu verwalten, die Einträge nur einmal enthalten. Dieser Container ist geeignet für Daten, die selten geändert aber oft gelesen werden, und bei denen eine Ordnung sinnvoll ist, zum Beispiel die „RHS_Deref“- und „RHS_Deref_Both“-Constraint-Mengen (sie werden nur am Anfang und während einer Zyklenkontraktion geändert).

5.2.3 Monoton-Natural-Set

Diese Container sind für Mengen von natürlichen Zahlen gedacht, in die nur neue Element eingefügt werden, oder alle Elemente gelöscht werden. Das Löschen einzelner Elemente ist nicht vorgesehen. Es gibt zwei grundlegende Container:

- Einen Bitvektor: für diesen muss die größte natürliche Zahl, die eingefügt werden soll, bereits bei der Initialisierung bekannt sein. Ein Bitvektor ist dabei ein Vektor von größeren Wörtern, und das Einfügen eines Elements entspricht dem Setzen eines Bits in einem dieser Wörter.

  Der Bitvektor ist für die Fälle geeignet, in denen schnell (und eventuell unsynchronisiert) geprüft werden soll, ob ein Eintrag bereits in der Menge enthalten ist. Er hat dafür einen großen, aber konstanten Speicherverbrauch.

- Ein Bit-Baum als Mischung von Bitvektor und Bäumen: es werden die Einträge eines Bitvektors, die nicht Null sind, in denen also mindestens ein Bit gesetzt ist, in einem Baum (`Ada.Containers.Ordered_Maps`) gespeichert.

  Der Bit-Baum ist in Fällen geeignet, in denen nur wenige Einträge aus einem großen Bereich gesetzt werden, und die auch öfters mit einem Cursor durchlaufen werden sollen.

Zu jedem dieser Container gibt es eine synchronisierte und eine unsynchronisierte Variante.

Dann gibt es noch eine synchronisierte Kombination von zwei Containern, um eine Menge zu verwalten, und neue Einträge in diese Menge in einer separaten Menge zu speichern; diese neuen Einträge können dann später extrahiert werden. Die Menge der neuen Einträge wird dabei immer in einem Bit-Baum gespeichert, für die normale Menge der Einträge gibt es eine Variante mit Bitvektor und eine mit Bit-Baum.
Diese Kombination wird für die „Pointsto“ und „Copyto“ Mengen eines Speicherfelds verwendet.

Die Kombination, die einen Bitvektor verwendet, hat dabei den Vorteil, dass sie unsynchronisiert überprüfen kann, ob ein Eintrag bereits in der Menge enthalten ist.

Um das Austauschen von Containern zu vereinfachen, gibt es nur einen Cursor-Typ für diese Container.

5.2.4 Todo-Queue

Die Todo-Queue besteht aus einem Bitvektor; ein gesetztes Bit gibt an, dass der entsprechende Eintrag in der Todo-Queue steht. Um die Zugriffe auf die Todo-Queue zu minimieren, bekommt jeder Thread nicht ein einzelnes Bit als Auftrag, sondern ein größeres Wort, einen Eintrag aus dem Vektor. Zum Durchlaufen der Einträge wird ein Cursor verwendet; neue Einträge für die Todo-Queue werden nicht direkt zurückgeschrieben, sondern werden zunächst im Cursor zwischengespeichert und erst beim nächsten Synchronisieren des Cursors in die Todo-Queue eingefügt.

Zusätzlich kann der Cursor einem Thread signalisieren, dass zusätzliche Arbeit zu tun ist; an dieser Stelle wird die „Copy_To"-Zyklensuche und -Kontraktion ausgelöst. Für einige Teile der Kontraktion müssen alle anderen Threads angehalten werden, und auch dies wird über die Todo-Queue verwaltet: die anderen Threads werden jeweils beim Warten auf neue Einträge blockiert.

Die Zusatzarbeit wird nur einem Thread signalisiert - erst wenn dieser die erfolgreiche Bearbeitung meldet, kann sie erneut vergeben werden.

5.3 „Copy_To"-Zyklensuche

Wie in Kapitel 2 bereits gezeigt, ist es nicht wichtig, einen konsistenten Snapshot der „Copy_To" Kanten für die Zyklensuche zu verwenden. Stattdessen kann eine Kante zu dem Zeitpunkt gelesen werden, an dem sie benötigt wird. Zum Beispiel kann man einen unsynchronisierten Cursor auf einem Bitvektor verwenden, um Lock Contentions zu vermeiden; dafür ist das Durchlaufen eines dünn besetzten Bitvektors zum Finden der gesetzten Bits nicht besonders effizient. Das synchronisierte Durchlaufen eines Bit-Baums hat auch einige Probleme, da zum Beispiel laut Ada Standard der Cursor für einen Baum beim Einfügen neuer Elemente ungültig wird (auch wenn dies in der Realität nicht der Fall ist); man kann also nicht mit einem Cursor durchlaufen, sondern muss jedesmal einen neuen Lookup im Baum starten. Als Alternative bleibt noch das Kopieren eines Bit-Baums in den Cursor; dies erhöht zwar den Speicherverbrauch merklich, da die Zyklensuche rekursiv abläuft und deshalb viele Cursors und deren Kopien von Bit-Bäumen im Speicher gehalten werden, minimiert aber die Zeit in denen die synchronisierten Bit-Bäume blockiert sind und bietet effizientes Durchlaufen von dünn besetzten Bitmengen.

Der Tarjan Algorithmus [Tar72] selbst ist ohne weitere Besonderheiten generisch implementiert.

5.4 „Copy_To"-Zyklenkontraktion

Nach dem Zyklen gefunden wurden, können die Ersetzungen vorgenommen werden, die nach Kapitel 2 möglich sind. Dazu werden die Constraint-Mengen der zu kontrahierenden

Speicherfeldern vereinigt, und in den „RHS_Deref_Both“-Mengen auf der linken Seite die Ersetzungen vorgenommen.
Letzteres kann auf jedem Speicherfeld einzeln vorgenommen werden, ohne die anderen Threads komplett anzuhalten; es darf nur das jeweilige Speicherfeld nicht gerade abgearbeitet werden. Um Synchronisationskonflikte möglichst zu vermeiden, werden die Speicherfelder rückwärts durchlaufen. Dadurch sollte die Kontraktion mit jedem anderen Thread höchstens zweimal zusammenstoßen, anstatt einem anderen Thread hinterherzulaufen ohne überholen zu können.

Für die Vereinigung der Speicherfelder werden alle anderen Threads angehalten. Dann wird jeweils ein Quell-Speicherfeld in ein Ziel-Speicherfeld vereinigt; das Quellfeld wird danach nicht mehr aktiv verwendet, es zeigt dann in der `SCC_Copyto` Komponente auf das Zielfeld. Auch in einer neuen Zyklensuche werden solche Felder übersprungen.
Im Prinzip sollen alle Constraints des Quellfeldes als neue Constraints im Zielfeld bearbeitet werden. Da es keine Mengen für neue „RHS_Deref“ bzw. „RHS_Deref_Both“ gibt, müssen diese direkt abgearbeitet werden, d.h. die „Points_To“ Ziele des Zielfeldes eingesetzt werden, die nicht auch „Points_To“ Ziele des Quellfeldes waren.
Die „Points_To“-, „Copy_To“- und „LHS_Deref“-Constraints des Quellfeldes können einfach als neue Constraints dem Zielfeld zugeordnet werden.

6 Benchmark

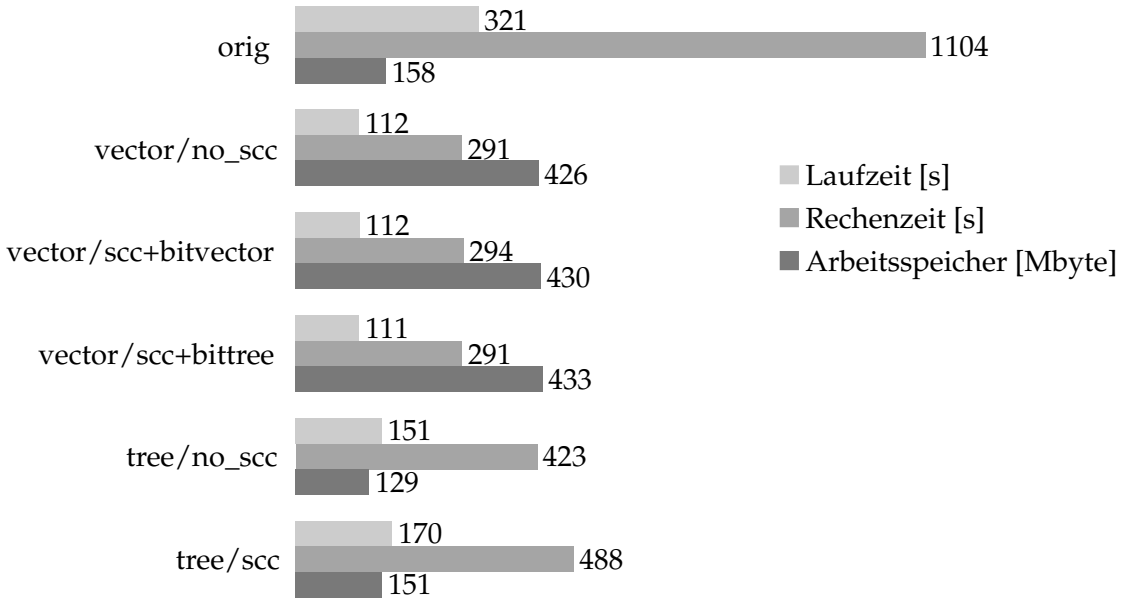
Als Testeingabe wurde der Apache httpd 1.3.27 [Apa02] verwendet. Er wurde mit Tools aus dem Projekt Bauhaus in die Zwischendarstellung übersetzt. Dann wurde die Zeit und der Speicherverbrauch des Points-To-Analysetools mit verschiedenen Implementierungsvarianten auf dieser Zwischendarstellung gemessen. Aufgrund einer Limitierung des Testsystems konnte nur der aktuelle Speicherverbrauch aus dem Überwachungstool „top“ abgelesen werden, so dass der maximale Speicherverbrauch geringfügig größer gewesen sein könnte. Die Zeiten wurde mit dem Tool „time“ [Fre96] gemessen, und der Durchschnitt von jeweils 3 Testläufen verwendet; die Laufzeit ergibt sich aus dem „elapsed time“ Wert, und die Rechenzeit aus der Summe der „user“ und „system“ Zeiten.

Das Testsystem hatte 4 Prozessorkerne, die während der Benchmarks abgesehen von der Points-To-Analyse keine nennenswerten Berechnungen ausgeführt haben.

Die Ausgabe aller Varianten wurde verglichen und war übereinstimmend.

Folgende Varianten wurden getestet:

- „orig“: Die bereits existierende Implementierung in C. Alle anderen Varianten sind in Ada neu geschrieben worden.
- „vector/no_scc“: Verwendet Bit-Vektoren um zu testen, ob „Points_To“- bzw. „Copy_To“-Constraints bereits gesehen wurden. Es findet keine „Copy_To“-Zyklensuche statt.
- „vector/scc+bitvector“: Verwendet Bit-Vektoren, und sucht die „Copy_To“-Zyklen mit Hilfe eines unsynchronisierten Cursors auf dem Bit-Vektor.
- „vector/scc+bittree“: : Verwendet Bit-Vektoren, und sucht die „Copy_To“-Zyklen mit Hilfe von Kopien der bereits bearbeiteten „Copy_To“-Constraints, die als Bit-Baum gespeichert sind.
- „tree/no_scc“: Verwendet Bit-Bäume um zu testen, ob „Points_To“- bzw. „Copy_To“-Constraints bereits gesehen wurden. Es findet keine „Copy_To“-Zyklensuche statt.
- „tree/scc“: Verwendet Bit-Bäume, und sucht die „Copy_To“-Zyklen mit Hilfe von Kopien dieser Bit-Bäume.



Alle Varianten der Neuimplementierung sind deutlich schneller als die existierende Implementierung. Während die Varianten mit den Bit-Vektoren klar die schnellsten sind, und wenig überraschend auch am meisten Speicher verwenden, ist die Ausnutzung der 4 Kerne mit den Bit-Vektoren deutlich kleiner als bei den anderen Varianten. Dies deutet auf einen weiteren Flaschenhals hin, der erst mit den Bit-Vektoren sichtbar wird, da die Bit-Vektoren selbst die Anzahl der Zugriffskonflikte nur verringern.

Interessant ist auch die Auswirkung der Zyklenkontraktion. Während in der Zusammenarbeit mit Bit-Vektoren kein Unterschied zu sehen ist, wird die Variante mit Bit-Bäumen bei Verwendung des Zyklenkontraktion deutlich langsamer.

Ein Test mit größeren Projekten könnte mehr Klarheit darüber bringen, welche Container-Auswahl effizienter ist; leider stößt man mit größeren Projekten zumindest mit den Bit-Vektoren schnell an Speicherlimits, und die Testläufe benötigen auch deutlich mehr Zeit.

Die erhoffte Performancesteigerung durch die Zyklen-Kontraktion war in diesem Test noch nicht zu sehen, und auch hier könnte ein Test mit größeren Projekten bessere Ergebnisse zeigen. Zusätzlich wäre zu untersuchen, ob nicht weitere Teile der Zyklenkontraktion parallel ausgeführt werden können.

7 Zusammenfassung und Ausblick

Die Neuimplementierung in Ada ist zumindest mit dem getesteten Projekt deutlich schneller als die existierende Implementierung in C. Weitere Benchmarks mit größeren Projekten wären sinnvoll, um die verschiedenen Container zu testen und zu sehen, welche Kombination die beste Performance für große Projekte bietet.

Auch mit größeren Projekten sollte die Zyklenkontraktion getestet werden, und die Implementierung einer parallelen Variante der Zyklenkontraktion sollte weitere Performancegewinne bringen.

Literaturverzeichnis

[And94] L. Andersen. Program analysis and specialization for the C programming language. Technical report, Technical Report 94-19, University of Copenhagen, 1994.

[Apa02] Apache Software Foundation. Apache httpd 1.3.27. `http://archive.apache.org/dist/httpd/apache_1.3.27.tar.gz`, 2002. [Online; geprüft am 22. September 2011].

[Fre96] Free Software Foundation. GNU time. `http://ftp.gnu.org/gnu/time/`, 1996. [Online; geprüft am 22. September 2011].

[KH05] G. Kroah-Hartman. container_of(). 2005. [Online; geprüft am 22. September 2011].

[Plo08] E. Ploedereder. Skript zur Vorlesung Programmanalysen und Compilerbau, 2008.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[TDB+06] S. Taft, R. Duff, R. Brukardt, E. Ploedereder, P. Leroy. Ada 2005 Reference Manual: Language and Standard Libraries. LNCS 4348, 2006.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Stefan Bühler)