Ludwig Maximilian Universität München

Haupt-/Bachelorseminar: Programmanalyse

Seminararbeit

Thema: Shape Analyse

eingereicht von: Daniel Fritsch ‹dfritsch86@arcor.de›

eingereicht am: 24. Juni 2009

Dozent: Prof. Martin Hofmann, Ph.D.

Betreuer: Lennart Beringer

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im folgenden wird die Shape Analyse erläutert, eine Analyse, die es erlaubt Informationen über diese Datenstrukturen zu gewinnen. Dies geschieht indem man durch die Shape Analyse die potentiell unendlichen Zustände des Heaps in eine endliche Darstellung überführt. Mögliche Anwendungen einer solchen Analyse sind dann:

- Erkennung von Deferenzierung von Null-Pointern
- Erkennung von Zugriffen auf deallocated storage
- Bestimmung der Erreichbarkeit einer Heap-Zelle (Garbage Collection..)
- etc

1.1 Aufbau des Papers

Zunächst wird die While-sprache die im Buch *Principles of Program Analysis* (Flemming Nielson et al.) verwendet wird so modifiziert, dass sie das Erstellen von Datenzellen im Heap erlaubt. Die neu eingeführten Syntax Elemente werden in der Einleitung erläutert. Im Abschnitt 2 werden dann die einzelne Elemente von Shape Graphen vorgestellt, bevor dann im Abschnitt 3 die Analyse selbst besprochen wird. Um das Ganze verständlicher und nicht allzu theoretisch zu belassen, wird ein laufendes Beispiel durch alle Abschnitte hindurch fortgesetzt.

1.2 Neue Elemente

Die grundlegende Struktur der While-Sprache die hier benutzt wird, wird als bekannt vorausgesetzt. Falls dies nicht der Fall sein sollte, vergleiche man *Principles of Program Analysis* (Flemming Nielson et al.). Diese wird nun so erweitert, dass sie es erlaubt strukturierte Datenzellen im Heap zu erstellen. Diese Zellen können dann Daten oder Pointer zu anderen Zellen beinhalten. Die Daten in solchen Zellen werden mit Selektoren abgerufen, also gibt es eine nicht-leere Menge von Selektorennamen:

sel $\in$ **Sel** Selektorennamen

Im folgenden Paper werden die Lisp-ähnlichen Selektorennamen car und cdr verwendet. x.car würde dann zuerst die Zelle auf die die Variable x zielt adressieren und dann den Wert im car Feld (= Head) zurückgeben. Ähnlich

für x.cdr wo dann der Wert des cdr-Feldes (=Tail) zurückgegeben wird. Mit Hilfe von Selektoren werden dann Pointer-Ausdrucke

p $\in$ **PExp**

definiert durch :

p ::= x | x.sel

Zusätzlich wird noch ein Syntax-Block malloc definiert, womit neue Zellen im Heap erstellt werden. Die Komplette Syntax der While-Sprache die hier verwendet wird lautet dann:

a ::= $p \mid n \mid a_1\ op_a\ a_2 \mid$ `nil`
b ::= `true` | `false` | `not` $b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid op_p\ p$
S ::= $[p{:=}a]^l \mid [$`skip`$]^l \mid S_1; S_2 \mid$
`if` $[b]^l$ `then` S_1 `else` $S_2 \mid$ `while` $[b]^l$ `do` $S \mid [$`malloc` $p]^l$

1.3 Operationelle Semantik

Um die obere Sprache auch modellieren zu können, werden nun einige Begriffe eingeführt, nämlich die der Locations, States und Heaps.

1.3.1 Locations

Locations sind eine unendliche Menge von Speicher-Orte/Adressen der Heap Zellen:

$\xi \in$ **Loc** Locations

1.3.2 States

Jede Variable wird also entweder einen Wert beinhalten (in dieser Sprache also eine ganze Zahl) oder eine Location oder $\diamond$ (NULL Wert). Somit werden States (Zustände) definiert als:

$\sigma \in$ **State** = $\mathbf{Var}_\star \rightarrow (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$

wo $\mathbf{Var}_\star$ die endliche Menge aller Variablen ist, die im betrachteten Programm vorkommen.

1.3.3 Heaps

Wie oben schon erwähnt, können Zellen im Heap mehrere Felder haben die mit verschiedenen Selektoren aufgerufen werden. Jedes Feld kann wiederum entweder ein Integer oder eine Location oder NULL sein. Dies wird so formalisiert:

$$\mathcal{H} \in \mathbf{Heap} = (\mathbf{Loc} \times \mathbf{Sel}) \rightarrow_{fin} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

wo $\rightarrow_{fin}$ darauf hindeutet, dass nicht alle Selektor Felder definiert sein müssen.

Mit Locations, States und Heaps kann man nun die neue Elemente so definieren:

Pointer Expression: Bei gegebenen State und Heap muss ein Pointer-Ausdruck p als Element von Z + Loc + $\{\diamond\}$ berechnet werden. Dafür wird die Funktion

$$\varrho : \mathbf{PExp}_\star \rightarrow (\mathbf{State} \times \mathbf{Heap}) \rightarrow_{fin} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

eingeführt, mit:

$$\varrho[[x]](\sigma, \mathcal{H}) = \sigma(x)$$

$$\varrho[[x.sel]](\sigma, \mathcal{H}) = \begin{cases} \mathcal{H}(\sigma(x), sel) \\ \quad \text{wenn } \sigma(x) \in \mathbf{Loc} \text{ und } \mathcal{H} \text{ ist definiert auf } (\sigma(x), sel) \\ undef \\ \quad \text{wenn } \sigma(x) \notin \mathbf{Loc} \text{ oder } \mathcal{H} \text{ ist undefiniert auf } (\sigma(x), sel) \end{cases}$$

Der erste Teil der Definition deckt den Fall, dass p eine einfache Variable ist. Durch den State wird dann ihr Wert ermittelt, was in der hier benutzten Sprache nur ein Integer, eine Location oder der NULL Wert sein kann.
Die zweite Hälfte deckt den Fall, dass p die Form x.sel besitzt. In solchen Fällen muss vorher der Wert von x ermittelt werden. Nur wenn x das Feld sel besitzt macht es dann Sinn diesen auch zu lesen. Deshalb ist der zweite Fall auch in zwei Teile gesplittet.

Malloc: Das malloc Konstrukt ist für das Erstellen neuer Zellen im Heap verantwortlich. Ähnlich wie für die Funktion ϱ wird es je nach Form von p unterschiedlich formalisiert:

$\Big\langle [\texttt{malloc } x]^l, \sigma, \mathcal{H} \Big\rangle \rightarrow \langle \sigma[x \mapsto \xi], \mathcal{H} \rangle$
dabei kommt ξ weder in σ noch $\mathcal{H}$ vor

$\Big\langle [\texttt{malloc } x.sel]^l, \sigma, \mathcal{H} \Big\rangle \rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \xi] \rangle$
dabei kommt ξ weder in σ noch $\mathcal{H}$ vor und $\sigma(x) \in \mathbf{Loc}$

Dabei erstellen beide Teile eine neue Location ξ, $\mathcal{H}(\xi$, sel) wird aber nicht initialisiert. Dies könnte man auch anders machen, die in diesem Paper benutzt Sprache wird aber so festgelegt. Weiterhin zu beachten ist, dass in der letzten Klausel kontrolliert wird ob eine Location die zu x entspricht auch tatsächlich existiert sodass man eine Verlinkung zur neuen Location erstellen kann.

Weitere Änderungen:
Im Vergleich zu vorangegangenen Kapiteln des Buches *Principles of Program Analysis* werden nun auch Statements und arithmetische bzw. boolsche Operatoren so geändert, dass sie mit Pointer und Selektoren umgehen können. So kann z.B. in einer Variable nun neben Zahlen und den NULL Wert auch ein Pointer gespeichert werden, oder es ist nun durch boolsche Operatoren möglich festzustellen ob zwei Pointer gleich sind, oder von NULL unterschiedlich.

1.4 Laufendes Beispiel

Um die neu eingeführten Konzepte zu verdeutlichen werden sie nun anhand eines Beispiels nochmals erläutert. Betrachtet wird folgendes Programm, das eine verkettete Liste die in der Variable x anfängt umdreht und sie dann in der Variable y anfangen lässt:

```
[y:= nil]1
while[not is-nil(x)]2 do
    ([z:= y]3;[y:= x]4;[x:= x.cdr]5;[y.cdr:= z]6);
[z:=nil]7
```

In Abbildung 1 ist die Funktionsweise des Programms grafisch dargestellt, wobei anfangs in x eine 5-elementige Liste gespeichert ist und die Variablen y und z undefiniert sind. In Zeile 0 wird das Heap gezeigt, bevor das Programm die while-Schleife erreicht. Die Zeile 5 zeigt den Heap nach dem letzten Ausführen der While-Schleife. Grundsätzlich gilt, dass Zeile n den Heap vor dem (n+1)-ten Durchlauf der Schleife darstellt.
Die ovalen Felder stellen dabei die Zellen im Heap dar, die mit ihrer Location/Adresse beschriftet sind. Die unbeschrifteten Pfeile sollen für den State σ

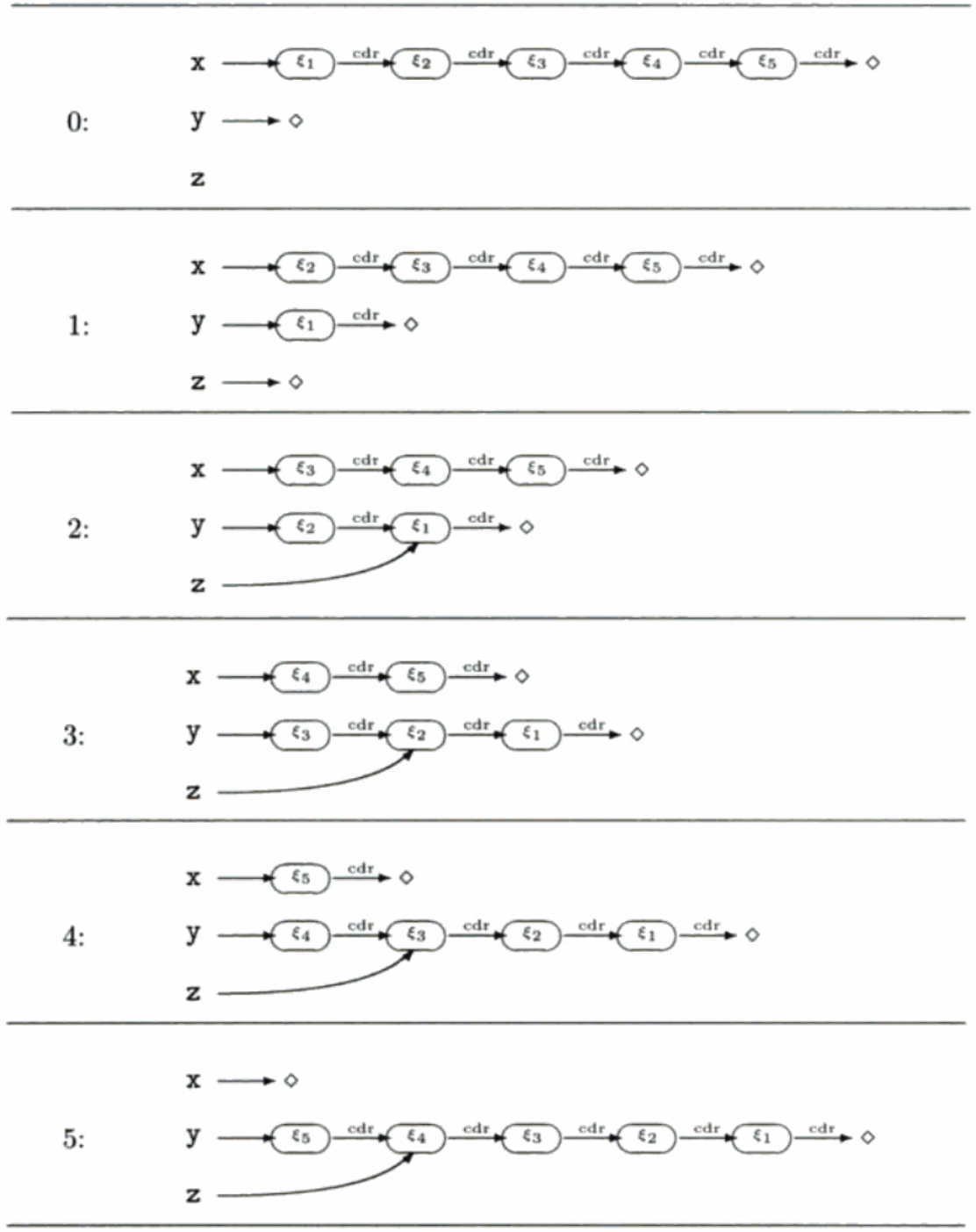

Abbildung 1: Drehung einer Liste mit 5 Elementen,
Quelle: F. Nielson et al, 2005

stehen. Eine unbeschriftete Verbindung von einer Variablen x zu einer Location ξ bedeutet also, dass $\sigma(\mathrm{x}) = \xi$. Die beschrifteten Pfeile modellieren den Heap $\mathcal{H}$. Eine mit sel markierte Verbindung von einer Location ξ zu einer Location ξ' bedeutet also, dass es einen sel-Pointer zwischen beiden Zellen gibt, mit $\mathcal{H}(\xi, \mathrm{sel}) = \xi$'.

2 Shape Graphen

Es ist offensichtlich, dass Programme existieren, für die das Heap unbegrenzt wachsen kann. Da das Ziel der Shape Analyse das Festlegen von bestimmten Eigenschaften des Heaps ist, muss das Heap endlich dargestellt werden können. Um dies zu erreichen werden abstrakte Locations, abstrakte States (`S`) und abstrakte Heaps (`H`) eingeführt. Die Grundidee die dahinter steckt, ist Locations mit ähnlichen Eigenschaften zu einer größeren abstrakten Location zusammenzufassen. Da dabei aber Informationen verloren gehen, wird auch die sharing Information (`is`) eingeführt, um einen Teil der Informationen wiederherstellen zu können. Im Folgenden wird nun erklärt wie ein gegebener State σ und ein Heap $\mathcal{H}$ einen Shape Graphen (`S`, `H`, `is`) erzeugen. Dabei werden `S`, `H` und `is` auch genau definiert und insgesamt 5 Invarianten aufgestellt.

2.1 Abstrakte Locations

Die abstrakte Locations haben die Form n_X, dabei ist X eine Teilmenge von $\mathrm{Var}_\star$, also den im Programm vorkommenden Variablen:

$$\mathbf{ALoc} = \{\mathrm{n}_X \mid \mathrm{X} \subseteq \mathbf{Var}_\star\} \qquad \text{abstrakte Locations}$$

Die Grundidee ist, dass wenn $\mathrm{x} \in \mathrm{X}$, dann stellt n_X (unter anderen) die Location $\sigma(\mathrm{x})$ dar. Zusätzlich wird eine sog. abstrakte zusammenfassende Location $\mathrm{n}_\emptyset$ definiert, die alle Locations darstellt, die nicht von einem State erreicht werden können, ohne dass man den Heap konsultiert. D.h. die Mengen von Locations n_X und $\mathrm{n}_\emptyset$ sind überschneidungsfrei wenn $\mathrm{X} \neq \emptyset$.
Verstärkend wird noch gefordert, dass 2 verschiedene abstrakte Locations n_X und n_Y überschneidungsfrei sind. Dies bedeutet also, dass entweder $\mathrm{X} = \mathrm{Y}$ oder $\mathrm{X} \cap \mathrm{Y} = \emptyset$. Die Invariante kann also folgendermaßen formuliert werden:

> **Invariante 1**: Wenn 2 abstrakte Locations n_X und n_Y im gleichen Shape-Graph vorkommen, so ist entweder $\mathrm{X} = \mathrm{Y}$ oder $\mathrm{X} \cap \mathrm{Y} = \emptyset$

Beispiel: Wenn man Zeile 2 von Abbildung 1 nochmals betrachtet, so ist deutlich zu erkennen, dass die Variablen x, y und z auf verschiedene Locations zeigen (nämlich ξ_3, ξ_2 und ξ_1). In einem Shape-Graph würden diese also durch die abstrakte Locations $\mathrm{n}_{\{x\}}$, $\mathrm{n}_{\{y\}}$ und $\mathrm{n}_{\{z\}}$ dargestellt werden. Die Locations ξ_4 und ξ_5 können nicht direkt von einem State erreicht werden, folglich werden sie durch die abstrakte zusammenfassende Location $\mathrm{n}_\emptyset$ dargestellt.

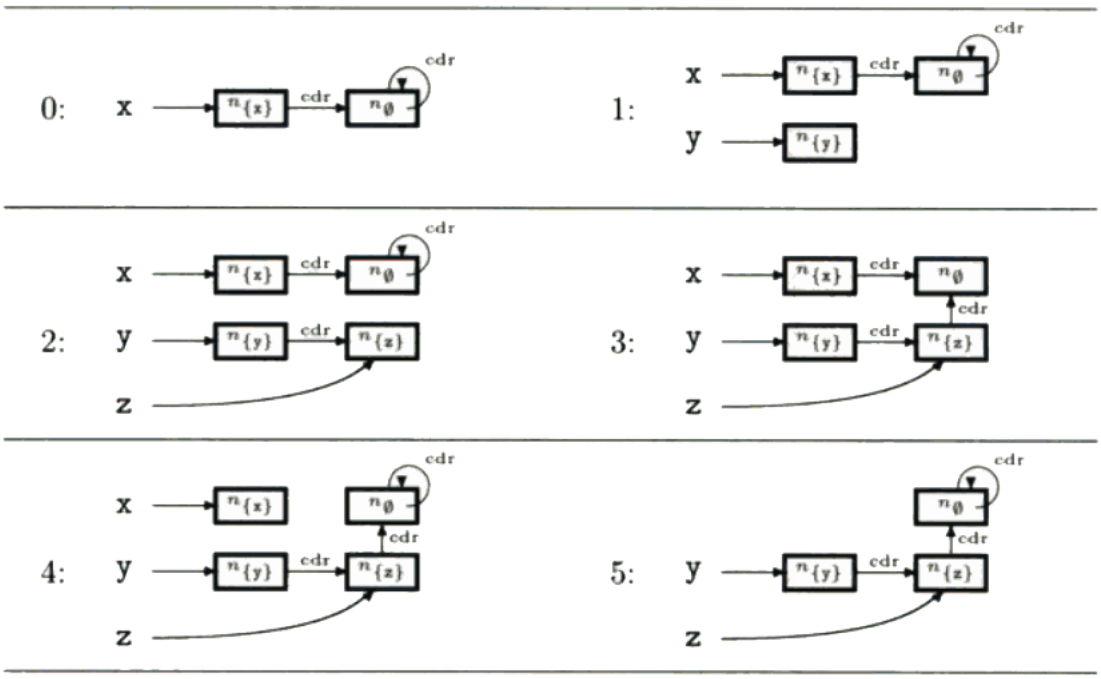

Abbildung 2: Shape Graphen der Abbildung 1,
Quelle: F. Nielson et al, 2005

2.2 Abstrakte States (S)

In einem Shape-Graph ordnet ein abstraktes State eine Variable x einer abstrakten Location n_X zu, eben genauso wie States eine Variable einer Location zuordnen. Deswegen wird folgende Invariante gefordert:

> **Invariante 2:** Wenn die Variable x einer abstrakten Location n_X zugeordnet wird, so ist $x \in X$

Zusammen mit Invariante 1 folgt daraus, dass jede Variable in höchstens einer abstrakten Location vorkommen kann. Abschliessend wird also festgehalten, dass ein abstraktes State als Element von

$S \in \mathbf{AState} = \mathcal{P}(\mathbf{Var}_\star \times \mathbf{ALoc})$ betrachtet werden kann, wobei
$\mathrm{ALoc}(S) = \{n_X \mid \exists x: (x, n_X) \in S\}$

2.3 Abstrakte Heaps (H)

Ähnlich wie Heaps die Verbindung zwischen zwei Locations spezifizieren, so spezifiziert ein abstrakter Heap die Verbindung zwischen zwei abstrakten

Locations. Solche Verbindungen werden mit Tripeln del Form (n_X, sel, n_Y) festgehalten, sodass man abstrakte Heaps H als Element folgender Menge definieren kann:

$H \in \mathbf{AHeap} = \mathcal{P}(\mathbf{ALoc} \times \mathbf{Sel} \times \mathbf{ALoc})$ wobei
$ALoc(H) = \{n_X, n_Y \mid \exists sel : (n_X, sel, n_Y) \in H\}$

Dabei soll es so sein, dass wenn $\mathcal{H}(\xi_1, sel) = \xi_2$ und ξ_1 von n_X und ξ_2 von n_Y dargestellt werden, dann ist $(n_X, sel, n_Y) \in H$. In einem einem Heap $\mathcal{H}$ wird es höchstens eine Location ξ_2 geben mit $\mathcal{H}(\xi_1, sel) = \xi_2$. In einem abstrakten Heap ist dies nur begrenzt der Fall, da die abstrakte Location $n_\emptyset$ mehrere Locations darstellen kann. Trotzdem müssen abstrakte Heaps folgender Invariante gerecht werden:

Invariante 3: Wenn (n_X, sel, n_Y) und (n_X, sel, n_Y') in einem abstrakten Heap sind, so ist entweder $X = \emptyset$, oder Y = Y'

Beispiel: Um das obige Beispiel weiterzuführen, betrachte man Zeile 2 der Abbildung 1. Der abstrakte State S2 von Zeile 2 ist:

$S2 = \{(x, n_{\{x\}}), (y, n_{\{y\}}), (z, n_{\{z\}})\}$

Der abstrakte Heap von Zeile 2 ist dann:

$H2 = \{(n_{\{x\}}, cdr, n_\emptyset), (n_\emptyset, cdr, n_\emptyset), (n_{\{y\}}, cdr, n_{\{z\}})\}$

Das erste Tripel stellt dar, dass das Heap ξ_3 und cdr mit ξ_4 verlinkt, dabei ist ξ_3 durch $n_{\{x\}}$ dargestellt und ξ_4 durch $n_\emptyset$. Das zweite zeigt, dass ξ_4 und cdr nach ξ_5 verlinkt, und ξ_4 und ξ_5 beide mit $n_\emptyset$ dargestellt werden. Das letzte Tripel bedeutet dann, dass ξ_2 und cdr nach ξ_1 verlinken, und dabei ist ξ_2 durch $n_{\{y\}}$ und ξ_1 durch $n_{\{z\}}$ dargestellt.
In Abbildung 2 sind die abstrakten Locations, die abstrakten Heaps und die abstrakten States für alle Zeilen von Abbildung 1 dargestellt. Dabei stehen die Rechtecke für abstrakte Locations, die unbeschrifteten Pfeile von einer Variablen zu einer abstrakten Location für abstrakte States und die beschrifteten Pfeile zwischen abstrakten Locations für abstrakte Heaps. Dabei wurden abstrakte States die nicht zu einer abstrakten Location führen einfach weggelassen.

2.4 Sharing Information (is)

Man betrachte die oberste Zeile von Abbildung 3. Der abstrakte Heap und abstrakte State auf der rechten Seite stellen den Heap und State der linken

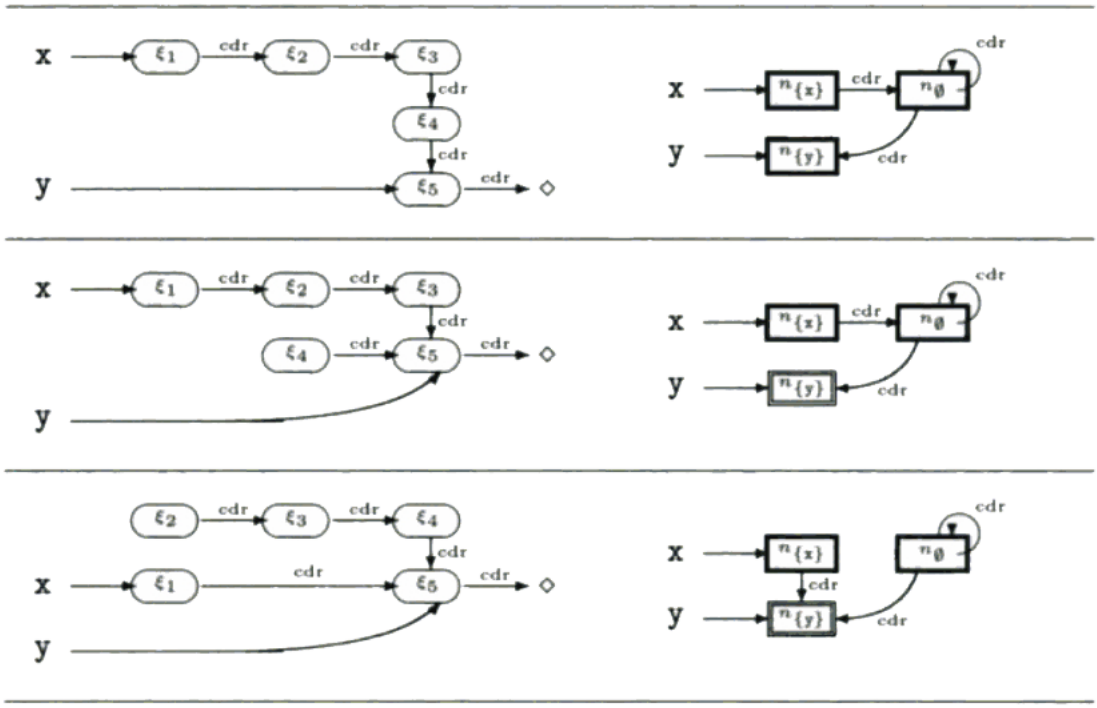

Abbildung 3: Sharing Information,Quelle: F. Nielson et al, 2005

Seite dar, aber auch den Heap und State auf der linken Seite der zweiten Zeile. Um dies zu unterscheiden wird die Sharing Information (is) eingeführt. Die Idee ist, eine Teilmenge is von abstrakten Locations zu definieren, die eine Location darstellen, die geteilt (im Sinne von share) wird. Eine abstrakte Location n_X wird also Element von is sein, wenn sie eine Location darstellt, die von mehr als ein Pointer gezielt wird. In der ersten Zeile von Abbildung 3 stellt $n_{\{y\}}$ die Location ξ_5 dar, und ξ_5 ist nicht geteilt (wie immer im Sinne von share), d.h. nur ein Heap-Pointer zeigt auf ξ_5, also ist hier $n_{\{y\}} \notin$ is. In der zweiten Zeile hingegen ist die Location ξ_5 geteilt, da ξ_3 und ξ_4 auf ihr zeigen, hier ist also $n_{\{y\}} \in$ is. Um geteilte abstrakte Locations zu verdeutlichen, erhalten diese ein doppeltes Rechteck, während nicht geteilte abstrakte Locations ein dickes Rechteck bekommen. Da die abstrakten Heaps selbst implizit Informationen über die Teilung von Locations beinhalten, muss sichergestellt werden, dass diese implizite Information mit der expliziten (von der is Menge) konsistent ist. Dazu werden 2 Invarianten eingeführt:

Invariante 4: Wenn $n_X \in$ is, dann ist entweder:
a) ($n_\emptyset$, sel, n_X) ist im abstrakten Heap für mindestens ein sel, oder
b) es existieren zwei verschiedene Tripel (n_Y, sel_1, n_X) und ($n_{Y'}$, sel_2, n_X)

im abstrakten Heap (also ist entweder $\text{sel}_1 \neq \text{sel}_2$ oder Y $\neq$ Y')

Diese Invariante sichert, dass die Information in der sharing-Komponente is auch im abstrakten Heap widergespiegelt wird. Fall a) kümmert sich darum, dass verschiedene Locations die durch $\text{n}_\emptyset$ dargestellt werden auf n_X zielen können, Fall b) deckt den Fall, dass n_X Ziel verschiedener Quellen oder verschiedener Selektoren sein kann.
Die nächste Invariante hingegen sichert dass die sharing Information im abstrakten Heap auch in der sharing-Komponente widergespiegelt wird:

> **Invariante 5:** Wenn zwei verschiedene Tripel (n_Y, sel_1, n_X) und ($\text{n}_{Y'}$, sel_2, n_X) im abstrakten Heap existieren und $\text{n}_X \neq \text{n}_\emptyset$, so ist $\text{n}_X \in$ is.

Diese deckt den Fall, dass n_X eine einzige Location darstellt, die von zwei oder mehr Heap-Pointers gezielt wird.

2.5 Zusammenfassung

Der Übersicht wegen wird hier nun zusammengefasst.
Ein Shape Graph ist ein Tripel welches aus einem abstrakten State S, einem abstrakten Heap H und einer Menge is geteilter abstrakter Locations mit:

$\text{S} \in \mathbf{AState} = \mathcal{P}(\mathbf{Var}_\star \times \mathbf{ALoc})$

$\text{H} \in \mathbf{AHeap} = \mathcal{P}(\mathbf{ALoc} \times \mathbf{Sel} \times \mathbf{ALoc})$

$\text{is} \in \mathbf{IsShared} = \mathcal{P}(\mathbf{ALoc})$

besteht, wobei $\mathbf{ALoc} = \{\text{n}_X \mid \text{X} \subseteq \mathbf{Var}_\star\}$. Ein Shape Graph (S, H, is) heißt *kompatibel* wenn es die fünf oben genannten Invarianten erfüllt:

1. $\forall n_X, n_Y \in ALoc(S) \cup ALoc(H) \cup is : (X = Y) \vee (X \cap Y = \emptyset)$
2. $\forall (x, n_X) \in S : x \in X$
3. $\forall (n_X, sel, n_Y), (n_X, sel, n_{Y'}) \in H : (X = \emptyset) \vee (Y = Y')$
4. $\forall n_X \in is : (\exists sel : (n_\emptyset, sel, n_X) \in H) \vee (\exists (n_Y, sel_1, n_X), (n_{Y'}, sel_2, n_X) \in H) : sel_1 \neq sel_2 \vee Y \neq Y'$
5. $\forall (n_Y, sel_1, n_X), (n_{Y'}, sel_2, n_X) \in H : ((sel_1 \neq sel_2 \vee Y \neq Y') \wedge X \neq \emptyset) \Rightarrow n_X \in is$

Die Menge der kompatiblen Shape Graphen wird als

$\mathbf{SG}$ = { (S, H, is) | (S, H, is) ist kompatibel }

bezeichnet.

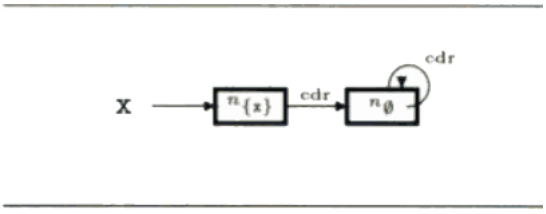

Abbildung 4: Der Shape Graph für den Entrem-Wert ι
für das Beispielprogramm,
Quelle: F. Nielson et al, 2005

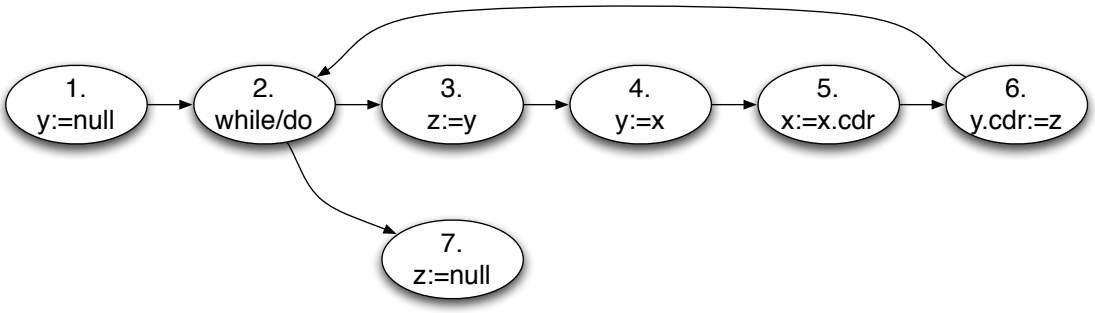

Abbildung 5: Graph für das Beispielprogramm

3 Analyse

Nachdem alle nötigen Begriffe eingeführt und erklärt worden sind, kann nun mit der Analyse angefangen werden. Diese wird eine Instanz eines *monotonen Frameworks* sein, der als bekannt vorausgesetzt wird (falls dies nicht der Fall sein sollte, so vergleiche man *Principles of Program Analysis* (Flemming Nielson et al.)). Für jedes korrekte Programm $S_\star$ erhält man eine Menge von Gleichungen der Form

$$Shape_\circ(l\,) = \begin{cases} \iota & \text{für } l \;= init(S_\star), \text{ sonst:} \\ \cup\{Shape_\bullet(l'\,) \mid (l',\, l\,) \in flow(S_\star)\} \end{cases}$$

$$Shape_\bullet(l\,) = f_l^{SA}(Shape_\circ(l))$$

wobei $\iota \in \mathcal{P}(\mathbf{SG})$ der Extremwert beim Eingang in $S_\star$ ist, und f_l^{SA} Transfer-Funktionen sind, die noch zu spezifizieren sind.
Die Analyse wird eine *forward Analyse* sein (wegen *flow*($S_\star$) und eine *may Analyse*, da $\cup$ als Kombinations-Operation benutzt wird.

Beispiel: Man betrachte das Beispiel-Programm zum Umkehren von verketteten Listen auf Seite 7 und man nehme an, dass zu Anfang x auf einer nicht-geteilten Liste mit mindestens zwei Elementen zielt, und dass y und z undefiniert sind. Der einzige Shape Graph der diesem Zustand entspricht wird in Abbildung 4 gezeigt und stellt den Extrem-Wert ι dar. In Abbildung 5 hingegen wird die Form des Programms dargestellt, dank der die Mengen $\text{Shape}_\circ(l\ )$ und $\text{Shape}_\bullet(l\ )$, die den Status des Heaps vor und nach dem Eintreten im Programmblock l beschreiben, berechnet werden können. Die Gleichungen für $\text{Shape}_\bullet(l\ )$ sind

$$\begin{aligned}
\text{Shape}_\bullet(1) &= \text{f}_1^{SA}(\text{Shape}_\circ(1)) = \text{f}_1^{SA}(\iota)\\
\text{Shape}_\bullet(2) &= \text{f}_2^{SA}(\text{Shape}_\circ(2)) = \text{f}_2^{SA}(\text{Shape}_\bullet(1) \cup \text{Shape}_\bullet(6))\\
\text{Shape}_\bullet(3) &= \text{f}_3^{SA}(\text{Shape}_\circ(3)) = \text{f}_3^{SA}(\text{Shape}_\bullet(2))\\
\text{Shape}_\bullet(4) &= \text{f}_4^{SA}(\text{Shape}_\circ(4)) = \text{f}_4^{SA}(\text{Shape}_\bullet(3))\\
\text{Shape}_\bullet(5) &= \text{f}_5^{SA}(\text{Shape}_\circ(5)) = \text{f}_5^{SA}(\text{Shape}_\bullet(4))\\
\text{Shape}_\bullet(6) &= \text{f}_6^{SA}(\text{Shape}_\circ(6)) = \text{f}_6^{SA}(\text{Shape}_\bullet(5))\\
\text{Shape}_\bullet(7) &= \text{f}_7^{SA}(\text{Shape}_\circ(7)) = \text{f}_7^{SA}(\text{Shape}_\bullet(2))
\end{aligned}$$

dabei werden die Transferfunktionen f_l^{SA} unten definiert.
Die Transferfunktion $f_l^{SA} : \mathcal{P}(\mathbf{SG}) \to \mathcal{P}(\mathbf{SG})$ hat folgende Form:

$$f_l^{SA}(SG) = \cup\{\phi_l^{SA}((\text{S, H, is})) \mid (\text{S, H, is}) \in SG\}$$

wobei $\phi_l^{SA} : \mathbf{SG} \to \mathcal{P}(\mathbf{SG})$ festlegt, wie ein *einzelner* Shape Graph in $\text{Shape}_\circ(l)$ zu einer *Menge* von Shape Graphen in $\text{Shape}_\bullet(l)$ wird. Es werden nun die verschiedenen Elementar-Blöcke der While-Sprache untersucht, und für jeden von ihnen die Transferfunktion ϕ_l^{SA} bestimmt, beginnend bei boolschen Ausdrücken und den Skip-Statement.

3.1 Transferfunktion für [b]^l und [skip]^l

Da weder bool'sche Tests, noch die Anweisung `skip` den Heap verändern gilt:

$$\phi_l^{SA}((\text{S, H, is})) = \{(\text{S, H, is})\ \}$$

Das heißt hier ist die Transferfunktion die Identität.

Beispiel: Im laufenden Beispiel ist z.B. $\text{Shape}_\bullet(2) = f_2^{SA}(\text{Shape}_\bullet(1) \cup \text{Shape}_\bullet(6))$. Da die Anweisung 2 ein bool'scher Test ist, gilt also $f_2^{SA} =$ Identität und somit $f_2^{SA}(\text{Shape}_\bullet(1) \cup \text{Shape}_\bullet(6)) = \text{Shape}_\bullet(1) \cup \text{Shape}_\bullet(6)$

3.2 Transferfunktion für $[x{:=}a]^l$

Dabei soll a die Form n, a_1 op_a a_2 oder `nil` haben. Die Wirkung dieser Anweisung ist es, die Verbindung von x zu entfernen und alle abstrakte Locations so umzubenennen, dass sie nicht mehr die Variable x enthalten. Dies geschieht durch die Funktion

$$k_x(n_Z) = n_{Z\setminus\{x\}}$$

und es ist

$$\phi_l^{SA}((\text{S, H, is})) = \{kill_x((\text{S, H, is}))\}$$

Wobei $kill_x((\text{S, H, is})) = (\text{S', H', is'})$ folgendermaßen definiert ist:

$$\text{S'} = \{(z, k_x(n_Z)) \mid (z, n_Z) \in S \wedge z \neq x)\}$$
$$\text{H'} = \{(k_x(n_V), sel, k_x(n_W)) \mid (n_V, sel, n_W) \in H\}$$
$$\text{is'} = \{(k_x(n_X)) \mid (n_X) \in \texttt{is}\}$$

Beispiel: Im Beispielprogramm hat die Anweisung `[y:=nil]`^{1} genau die hier betrachtete Form. Da y in ι (Abbildung 4) nicht vorkommt, ist Shape$_\bullet(1)$ identisch zu ι.
Interessanter wird es, wenn $(x, n_{\{x\}}) \in$ S, da in diesem Falle die abstrakten Locations $n_{\{x\}}$ und $n_\emptyset$ vereint werden. Die Sharing information wird dann so aktualisiert, dass $n_\emptyset$ ganz sicher nicht-geteilt ist, wenn vor der Vereinigung $n_\emptyset$ und $n_{\{x\}}$ nicht geteilt waren

Beispiel: Die Anweisung `[z:=nil]`^{7} des List-Reversal Programm stellt diese Situation dar. Für alle Shape Graphen von Shape$_\bullet(2)$ wird die abstrakte Location $n_{\{z\}}$ mit $n_\emptyset$ vereint und erzeugt somit einen der Graphen von Shape$_\bullet(7)$.

3.3 Transferfunktion für $[x{:=}y]^l$

Im Falle x = y ist die Transferfunktion trivialerweise die Identität. Man nehme nun an, dass x $\neq$ y. Die erste Wirkung dieser Anweisung ist es, die alten Verbindungen zu x zu entfernen. Dafür wird die oben eingeführte Funktion $kill_x$ benutzt. Dann werden die neuen Verbindungen zu x erstellt, indem die abstrakten Locations, die y enthalten so umbenannt werden, dass sie auch x enthalten. Dies passiert durch die Funktionen:

$$g_x^y(n_Z) = \begin{cases} n_{Z\cup\{x\}} & \text{wenn } y \in Z \\ n_Z & \text{sonst} \end{cases}$$

Es soll sein:

$$\phi_l^{SA}((\text{S, H, is})) = \{(\text{S", H", is"})\}$$

mit (S', H', is') $= kill_x((\text{S, H, is}))$ und

$$\begin{aligned}
\text{S"} &= \{(z, g_x^y(n_Z)) \mid (z, n_Z) \in S'\} \\
&\quad \cup \{(x, g_x^y(n_Y)) \mid (y', n_Y) \in S' \wedge y' = y\} \\
\text{H"} &= \{(g_x^y(n_V), sel, g_x^y(n_W)) \mid (n_V, sel, n_W) \in H'\} \\
\text{is"} &= \{(g_x^y(n_Z) \mid n_Z \in is'\}
\end{aligned}$$

Dabei erstellt die zweite Klausel in der Formel für S" die neuen Verbindungen zu x.

Beispiel: Die Anweisung `[y:=x]`^{4} besitzt die hier betrachtete Form. Jeder der Graphen in Shape$_\bullet$(3) wird zu einem Graph in Shape$_\bullet$(4). Ähnliches gilt für `[z:=y]`^{3}

3.4 Transferfunktion für $[x{:=}y.sel]^l$

Man nehme an x = y. Dann kann man die Anweisung folgendermaßen umschreiben:

$$[t := y.sel]^{l_1}; [x := t]^{l_2}; [t := \texttt{nil}]^{l_3}$$

Dabei ist t eine neue (temporäre) Variable, und l_1, l_2, l_3 neue Labels. Die Transferfunktion f_l^{SA} ist dann:

$$f_l^{SA} = f_{l_3}^{SA} \circ f_{l_2}^{SA} \circ f_{l_1}^{SA}$$

$f_{l_2}^{SA}$ und $f_{l_3}^{SA}$ sind dabei aus obigen Beispielen schon bekannt. Es soll nun die Transferfunktion $f_{l_1}^{SA}$ untersucht werden oder die gleichbedeutende f_l^{SA} im Falle x $\neq$ y.

Beispiel: Die Anweisung `[x:=x.cdr]`^{5} im Beispielprogramm wird dann zu `[t:=x.cdr]`^{51}; `[x:=t]`^{52}; `[t:=x.nil]`^{53}.
Man betrachte nun also den Fall x $\neq$ y. Wie schon in vorangegangenen Beispielen wird zunächst die Funktion $kill_x$ aufgerufen, um die alten Verbindungen zu x zu entfernen:

$$(\text{S', H', is'}) = kill_x((\text{S, H, is}))$$

Im nächsten Schritt muss die abstrakte Location die y.sel entspricht so umbenannt werden, dass sie auch x beinhaltet und die Verbindung von x zu dieser Location erstellt werden. Es gibt dann drei Fälle:

1. Es existiert keine abstrakte Location n_Y mit $(y, n_Y) \in$ S' oder es gibt eine abstrakte Location n_Y mit $(y, n_Y) \in$ S' aber kein n_Z mit $(n_Y, sel, n_Z) \in$ H'. In solchen Fällen stellt der Shape Graph einen State und Heap dar, in denen y oder y.sel entweder ein Integer oder `nil` oder undefiniert sind.

2. Es existiert eine abstrakte Location n_Y mit $(y, n_Y) \in$ S' und es existiert eine abstrakte Location $n_U \neq n_\emptyset$ mit $(n_Y, sel, n_U) \in$ H'. Dies ist der Fall, wenn der Shape Graph einen State und Heap darstellt, in denen die Location die von y.sel gezielt wird auch von anderen Variablen (in U) gezielt wird.

3. Es gibt eine abstrakte Location n_Y mit $(y, n_Y) \in$ S' und $(n_Y, sel, n_\emptyset) \in$ H'. Hier stellt der Shape Graph einen State und Heap dar, in denen keine andere Variable zur Location zielt, auf die y.sel zielt

Fall 1: Man betrachte die Anweisung [x:=y.sel]^l (mit $x \neq y$) wobei es keine abstrakte Location n_Y mit $(y, n_Y) \in$ S' gibt. Dann gibt es auch keine abstrakte Location für y.sel, also keine abstrakte Location die umbenannt werden muss und zu der Verbindungen erstellt werden müssen. Also ist hier:

$$\phi_l^{SA}((\text{S, H, is})) = \{kill_x((\text{S, H, is}))\}$$

Oder aber es gibt eine abstrakte Location n_Y mit $(y, n_Y) \in$ S' aber keine abstrakte Location n_Z mit $(n_Y, sel, n_Z) \in$ H'. Aus den Invariante folgt, dass n_Y eindeutig ist und es gibt weiterhin keine abstrakten Locations zum umbenennen oder zu der Verbindungen erstellt werden müssen. Also gilt hier wieder:

$$\phi_l^{SA}((\text{S, H, is})) = \{kill_x((\text{S, H, is}))\}$$

Zu beachten ist, dass im ersten Fall verhindert wird, dass ein `NULL`-Pointer derefenziert wird, und im zwieten Fall verhindert wird, dass ein nicht-existierendes Selektor-Feld derefenziert wird.

Fall 2: Man betrachte die Anweisung [x:=y.sel]^l (mit $x \neq y$) wobei es eine abstrakte Location n_Y mit $(y, n_Y) \in$ S' gibt und es existiert eine abstrakte Location $n_U \neq n_\emptyset$ mit $(n_Y, sel, n_U) \in$ H'. Dann sind n_Y und n_U wegen der Invarianten eindeutig. Die abstrakte Location n_U wird dann so umbenannt, dass sie die Variable x beinhaltet. Dies geschieht durch Anwendung der Funktion:

$$h_x^U(n_Z) = \begin{cases} n_{U\cup\{x\}} & \text{wenn } Z = U \\ n_Z & \text{sonst} \end{cases}$$

Es soll sein:

$$\phi_l^{SA}((\text{S, H, is})) = \{(\text{S", H", is"})\}$$

mit (S', H', is') = $kill_x$((S, H, is)) und

$$\text{S"} = \{(z, h_x^U(n_Z)) \mid (z, n_Z) \in S'\} \cup \{(x, h_x^U(n_U))\}$$
$$\text{H"} = \{(h_x^U(n_V), sel', h_x^U(n_W)) \mid (n_V, sel', n_W) \in H'\}$$
$$\text{is"} = \{(h_x^U(n_Z) \mid n_Z \in is'\}$$

Das Einbinden von $(x, h_x^U(n_U))$ in S" erstellt die neuen Verbindungen. Die Definition von is" stellt sicher, dass die Sharing Information bei der Operation erhalten bleibt. Es gilt dann, dass $n_{U\cup\{x\}}$ in H" geteilt ist, nur genau dann, wenn n_U in H' geteilt wird.

Fall 3: Da Fall 3 (es gibt eine abstrakte Location n_Y mit $(y, n_Y) \in$ S' und $(n_Y, sel, n_\emptyset) \in$ H') eine neue abstrakte Location einführt, und dies den abstract Heap modifiziert (was recht kompliziert werden kann) wird hier auf eine Herleitung verzichtet. Interessierte können diese in *Principles of Program Analysis* (Flemming Nielson et al.) nachlesen.

3.5 Transferfunktion für $[x.sel := a]^l$

Dabei soll a die Form n, $a_1\ op_a\ a_2$ oder `nil` haben. Man nehme zuerst an, dass es kein n_X mit $(x, n_X) \in$ S gibt. Das würde heißen, dass x zu keiner Zelle im Heap zielt, und die obige Anweisung wird also keinen Einfluss auf die Form des Heaps haben, sodass die Transferfunktion die Identität ist.
Als Nächstes nehme man an, dass es ein (eindeutiges, wegen der Invariante) n_X mit $(x, n_X) \in$ S gibt, aber kein n_U mit $(n_X, sel, n_U) \in$ H. Das bedeutet, dass die Zelle auf die *sel* zielt, auf keine weitere Zelle zielt. Also wird auch hier die Anweisung den Heap nicht ändern, sodass die Transferfunktion wieder die Identität ist.
Interessant wird es, wenn es abstrakte Locations n_X und n_U gibt, mit $(x, n_X) \in$ S und $(n_X, sel, n_U) \in$ H. Wegen der Invariante werden diese abstrakten Locations eindeutig sein. Die Wirkung der Anweisung wird es sein, das Tripel (n_X, sel, n_U) aus H zu entfernen:

$$\phi_l^{SA}((\text{S, H, is})) = \{kill_{x.sel}((\text{S, H, is}))\}$$

wobei $kill_{x.sel}$((S, H, is)) = (S', H', is') so definiert wird:

$$\text{S'} = \text{S}$$

$$\text{H'} = \{(n_V, sel', n_W) \mid (n_V, sel', n_W) \in H \wedge \neg(X = V \wedge sel = sel')\}$$

$$\text{is'} = \begin{cases} \text{is}\backslash\{n_U\} & \text{wenn } n_U \in \text{is} \wedge \#into(n_U, H') \leq 1 \wedge \\ & \quad \neg\exists sel' : (n_\emptyset, sel', n_U) \in H' \\ \text{is} & \text{sonst} \end{cases}$$

Die Shared Information bleibt also gleich, ausser wenn nach dem Entfernen des Pointers (in x.sel) nur noch ein Pointer auf n_U zielt und dieser nicht aus $n_\emptyset$ stammt. In diesem Falle ist dann n_U nicht mehr geteilt und es wird aus is entfernt. Dabei soll $\#into(n_U, H')$ die Anzahl der Pointer zurückgeben, die auf n_U zielen.

3.6 Transferfunktion für $[x.sel := y]^l$

Man nehme an x = y. Dann kann man die Anweisung folgendermaßen umschreiben:

$$[t := y]^{l_1}; [x.sel := t]^{l_2}; [t := \texttt{nil}]^{l_3}$$

Dabei ist t eine neue (temporäre) Variable, und l_1, l_2, l_3 neue Labels. Die Transferfunktion f_l^{SA} ist dann:

$$f_l^{SA} = f_{l_3}^{SA} \circ f_{l_2}^{SA} \circ f_{l_1}^{SA}$$

$f_{l_1}^{SA}$ und $f_{l_3}^{SA}$ sind dabei aus obigen Beispielen schon bekannt. Es soll nun die Transferfunktion $f_{l_2}^{SA}$ untersucht werden oder die gleichbedeutende f_l^{SA} im Falle x $\neq$ y.
Sei also x $\neq$ y. Falls es kein n_X mit $(x, n_X) \in$ S gibt, so ist die Transferfunktion die Identität, da die Anweisung den Heap nicht ändern kann. Man nehme also an, dass n_X die Bedingung $(x, n_X) \in$ S erfüllt. Der Fall in dem es kein n_Y mit $(x, n_Y) \in$ S gibt entspricht dem Fall, dass y entweder ein Integer oder `nil` oder undefiniert ist und ähnelt also dem Fall $[x.sel$`:= nil`$]^l$:

$$\phi_l^{SA}(\text{(S, H, is)}) = \{kill_{x.sel}(\text{(S, H, is)})\}$$

Interessant wird es erst, wenn x $\neq$ y, $(x, n_X) \in$ S und $(y, n_Y) \in$ S. Im ersten Schritt wird dann die alte Verbindung für $x.sel$ entfernt, mit der Funktion $kill_{x.sel}$ Im zweiten Schritt wird dann die neue Verbindung für $x.sel$ gesetzt. Also ist:

ϕ_l^{SA}((S, H, is)) = {(S", H", is")}

wobei (S', H', is') = $kill_x$((S, H, is)) und

S" = S' (= S)

H" = H' ∪{(n_X, sel, n_Y)}

$$\text{is"} = \begin{cases} \text{is} \cup \{n_Y\} & \text{wenn } \#into(n_Y, H') \geq 1 \\ \text{is} & \text{sonst} \end{cases}$$

Hier ist zu beachten, dass n_Y geteilt werden kann, wenn wir einen neuen Pointer darauf zeigen lassen.

3.7 Transferfunktion für $[x.sel{:=}y.sel']^l$

Man kann die Anweisung folgendermaßen umschreiben:

$[t := y.sel']^{l_1}; [x.sel := t]^{l_2}; [t := \texttt{nil}]^{l_3}$

Dabei ist t eine neue (temporäre) Variable, und l_1, l_2, l_3 neue Labels. Die Transferfunktion f_l^{SA} ist dann:

$f_l^{SA} = f_{l_3}^{SA} \circ f_{l_2}^{SA} \circ f_{l_1}^{SA}$

$f_{l_1}^{SA}$, $f_{l_2}^{SA}$ und $f_{l_2}^{SA}$ sind dabei schon in den vorangegangenen Fällen spezifiziert worden, sodass hier die Transferfunktion bekannt ist.

3.8 Transferfunktion für [malloc p]^l

Man betrachte zuerst die Anweisung `[malloc` x`]`^{l}, wo man zunächst die Verbindung von x entfernen muss und dann eine neue (nicht-geteilte) Location einführen muss, auf die x dann zielt. Also wird definiert:

ϕ_l^{SA}((S, H, is)) = {(S' ∪ {$(x, n_{\{x\}})$}, H', is')}

mit (S', H', is') = $kill_x$(S, H, is). Die Anweisung `[malloc` $(x.sel)$`]`^{l} entspricht dann der Folge

$[\texttt{malloc } t]^{l_1}; [x.sel := t]^{l_2}; [t := \texttt{nil}]^{l_3}$

Dabei ist t eine neue (temporäre) Variable, und l_1, l_2, l_3 neue Labels. Die Transferfunktion f_l^{SA} ist dann:

$f_l^{SA} = f_{l_3}^{SA} \circ f_{l_2}^{SA} \circ f_{l_1}^{SA}$

$f_{l_1}^{SA}$, $f_{l_2}^{SA}$ und $f_{l_3}^{SA}$ sind dabei schon in den vorangegangenen Fällen spezifiziert worden, sodass hier die Transferfunktion bekannt ist.

4 Zusammenfassung

Ziel dieses Papers ist es, die Grundkonzepte der Shape Analyse zu erläutern. Dafür wurde die aus dem Buch *Principles of Program Analysis* (Flemming Nielson et al.) schon bekannte While-Sprache so erweitert, dass sie das Erstellen von Datenzellen im Heap erlaubt. Die Begriffe Heap, Location und State, die somit eingeführt worden sind, wurden dann spezifiziert. Da es Programme gibt, für die die Struktur des Heaps potentiell unendlich wachsen kann, wurde es notwendig ein Instrument einzuführen, welches die potentiell unendlichen Zustände des Heaps in einer endlichen Struktur überführt, die Shape Graphen. Die Grundidee besteht darin, Elemente so zusammenzufassen, dass man Heaps nun endlich darstellen kann, aber nicht zu sehr als dass man keine interessanten Eigenschaften des Heaps mehr erkennen kann. Nachdem die einzelnen Teile eines Shape Graphen eingeführt und erklärt worden sind, wurde mit der Analyse begonnen.

Jede Anweisung eines Programms kann die Form des Shape Graphen ändern, es existieren also sog. Transferfunktionen die diese Änderungen beschreiben, die also eine Menge von Graphen vor der Anweisung in einer Menge von Graphen nach der Anweisung überführen. Dadurch ist es zu jedem Zeitpunkt möglich, bestimmte Aussagen über den Heap zu treffen, was für verschiedene Aufgaben nützlich sein kann.