Interprozedurale Analyse

Benjamin Langlotz

10. Juli 2009

Inhaltsverzeichnis

1 Grundlagen

1.1 Syntax der Prozedursprache

Die bisher behandelten Datenflussanalysen beschäftigten sich mit einfachen Sprachen ohne Funktionen oder Prozeduren. Komplizierter wird es, wenn man diese mit betrachtet. Schwierigkeiten entstehen dabei, wenn man sicherstellen möchte, dass Prozeduraufruf und Prozedurrückgabe gegenseitig richtig zugeordnet sind, wenn es um Parametermechanismen geht, oder wenn man Prozeduren auch als Parameter übergeben möchte. Im Folgenden werde ich einige der gängigsten Techniken der Interprozeduralen Analyse darstellen. Um möglichst einfach an die vorhergegangenen Kapitel anzuknüpfen erweitern wir einfach die `WHILE` Sprache um Deklarationen von globalen wechselseitig rekursiven Prozeduren, denen ein call-by-value und ein call-by-result Parameter übergeben wird. Die Erweiterung dieser Definitionen um mehrere call-by-value, call-by-result und call-by-value-result ist einfach, wie auch die Erweiterung um lokale Variablen.

Beginnen wir zunächst mit der Erweiterung der `WHILE` Sprache um die benötigten Deklarationen. Bisher hatte die Sprache als Startsymbol das S für Statements, das nun von einem Programm $\mathtt{P}_*$ ersetzt wird, das folgendermaßen aufgebaut ist:

$$\mathtt{begin}\,\mathtt{D}_*\,\mathtt{S}_*\mathtt{end} \text{ wobei } \mathtt{D}_*$$

eine Folge von Prozedurdeklarationen ist:

$$\mathtt{D} ::= \mathtt{proc}\; p(\mathtt{val}\; x, \mathtt{res}\; y)\; \mathtt{is}^{\mathtt{l_n}}\; \mathtt{S}\; \mathtt{end}^{\mathtt{l_x}} \mid D\; D$$

Hierbei bezeichnen die Labels $\mathtt{l}_n$ und $\mathtt{l}_x$ den Eintritt in, bzw. den Ausgang aus dem Rumpf der Prozedur. “p” ist der Name der Prozedur, x ist der Eingabeparameter und in y wird der Rückgabewert gespeichert. Man hat also nicht wie beispielsweise in Java $y = p(x)$ sondern übergibt die Variable mit. Die Syntax der Statements wird ebenso erweitert:

$$S ::= \ldots \mid [call\; p(a,\, z)]^{l_c}_{l_r}$$

Das Call-Statement hat auch zwei Labels, von denen $\mathtt{l}_c$ den Aufruf der Prozedur und $\mathtt{l}_r$ die dazugehörige Rückkehr/Rückgabe kennzeichnet. a und z sind die Parameter die man beim eigentlichen Aufruf der Prozedur übergibt. Der Parametermechanismus für den ersten Parameter ist call-by-value und der für den zweiten ist call-by-result, wobei die Prozeduren wechselseitig rekursiv sein dürfen (das heißt, dass beispielsweise in der Prozedurdeklaration A Prozedur B aufgerufen werden darf, in deren Deklaration aber wiederum A aufgerufen wird).

1.2 Erweiterung der Flussgraphendefinition

Für die folgenden Betrachtungen wollen wir ein paar Vorbedingungen festlegen:
Zunächst soll angenommen werden, dass jedes Label im Programm einmalig ist, sodass das Programm Label-konsistent ist. Des Weiteren wird angenommen, dass $\mathtt{D}_*$ keine zwei Prozeduren mit identischem Namen enthält, sowie dass nur Prozeduren aufgerufen werden, die auch in $\mathtt{D}_*$ deklariert wurden.

Als Nächstes erweitert man die Funktionen init, final, blocks, labels und flow der Flussgraphen, um die Flussgraphen auch für die erweiterte Sprache nutzen zu können. So setzt man beispielsweise für das neue Statement:

$$\mathit{init}([\mathtt{call}\ p\,(\mathtt{a},\ \mathtt{z})]^{\mathtt{l_c}}_{\mathtt{l_r}}) = l_{\mathtt{c}}$$
$$\mathit{final}([\mathtt{call}\ p\,(\mathtt{a},\ \mathtt{z})]^{\mathtt{l_c}}_{\mathtt{l_r}}) = \{l_{\mathtt{r}}\}$$
$$\mathit{blocks}([\mathtt{call}\ p\,(\mathtt{a},\ \mathtt{z})]^{\mathtt{l_c}}_{\mathtt{l_r}}) = \{[\mathtt{call}\ p\,(\mathtt{a},\ \mathtt{z})]^{\mathtt{l_c}}_{\mathtt{l_r}}\}$$
$$\mathit{labels}([\mathtt{call}\ p\,(\mathtt{a},\ \mathtt{z})]^{\mathtt{l_c}}_{\mathtt{l_r}}) = \{l_{\mathtt{s}},\ l_{\mathtt{r}}\}$$
$$\mathit{flow}([\mathtt{call}\ p\,(\mathtt{a},\ \mathtt{z})]^{\mathtt{l_c}}_{\mathtt{l_r}}) = \{(l_{\mathtt{c}}; l_{\mathtt{n}}), (l_{\mathtt{x}}; l_{\mathtt{r}})\}$$

Das Statement für *flow* gilt, wenn $\mathtt{begin}\,\mathtt{D}_*\,\mathtt{S}_*\mathtt{end}$ wobei $\mathtt{D}_*$ in D_* ist.

Hier ist zu beachten, dass $(\mathrm{l}_c;\ \mathrm{l}_n)$ und $(\mathrm{l}_x;\ \mathrm{l}_r)$ zwei neue Flussarten sind. Man differenziert hier um die Flüsse von Prozeduraufruf und -rückkehr von normalen Flüssen unterscheiden zu können, denn
$(\mathrm{l}_c;\ \mathrm{l}_n)$ ist der Fluss entsprechend dem Aufruf einer Prozedur l_c und dem Eintritt in ihren Rumpf bei l_n
und
$(\mathrm{l}_x;\ \mathrm{l}_r)$ ist der Fluss entsprechend dem Verlassen des Rumpfes bei l_x und der Rückkehr - korrenspondierend zum Aufruf - bei l_r

Die Erweiterung der Flussgraphendefinitionen für ein Programm und die Deklarationen finden sich im Anhang, da sie analog zu der Erweiterung der Statements gebildet werden.
Zu bemerken ist allerdings die neu eingeführte Definition für den interprozeduralen Fluss:

$$\mathit{inter\text{-}flow}_* = \{(\mathtt{l_c}, \mathtt{l_n}, \mathtt{l_x}, \mathtt{l_r}) | P_*\ \mathtt{contains}\ [\mathtt{call}\ p(\mathtt{a},\ \mathtt{z})]^{\mathtt{l_c}}_{\mathtt{l_r}}\}$$

für $\mathtt{proc}p(\mathtt{val\,x},\ \mathtt{res\,y})\ \mathtt{is}^{\mathtt{l_n}}\ \mathtt{S}\ \mathtt{end}^{\mathtt{l_x}}$

inter-flow$_*$ stellt klar den Zusammenhang zwischen den labels eines Prozeduraufrufs und der zugehörigen Rückkehr dar, was später dabei hilft Prozeduren präziser zu analysieren. Stellt man sich vor, inter-flow$_*$ enthält (l_c^i, l_n, l_x, l_r^i) für i=1,2, dann enthällt flow (l_c^i, l_n) und (l_x, l_r^i) für i=1,2
Dies führt aber dazu, dass man die zusammengehörigen Aufrufe und Rückgaben nicht mehr korrekt zuordnen kann, das heißt man könnte folgende Kombinationen bilden:

$$(l_c^1, l_n, l_x, l_r^1)$$
$$(l_c^1, l_n, l_x, l_r^2)$$
$$(l_c^2, l_n, l_x, l_r^1)$$
$$(l_c^2, l_n, l_x, l_r^2)$$

Wobei nur das erste und letzte Quadrupel korrekt sind.Aus diesem Grund braucht man inter-flow$_*$

Bezüglich der Monotonen Frameworks nutzt man für die Vorwärtsanalyse F = flow$_*$ und E = init$_*$ wie gehabt, führt aber noch eine neue Metavariable IF = inter-flow$_*$ für den interprozeduralen Fluss ein. Für eine Rückwärtsanalyse verwendet man dementsprechend F = flow$_*$, E = final$_*$ und IF = inter-flow$_*$R. Der Großteil dieses Kapitels beschäftigt sich mit Vorwärtsanalyse.

1.3 Strukturelle Operationale Semantik

Wenn man sich nun ein Programm mit Prozeduren anschaut, so können die Variablen an unterschiedlichen Stellen im Programm verschiedene Werte haben, was uns zu der Erweiterund der Semantik von `WHILE` führt. Um sicherzustellen, dass die Sprache auch lokale Daten bzw. Variablen in Prozeduren unterstützt, muss man zwischen den Werten die den Variablen an verschiedenen Stellen zugewiesen sind unterscheiden können, weswegen wir eine unendliche Menge von Adressen (oder Locations) einführen:

$$\xi \in \mathbf{Loc}$$

Eine Umgebung ρ bildet die Variablen in ihrem momentanen Bereich auf ihre Adressen ab, worauf ein Store ζ die Werte dieser Adressen spezifizieren wird.
$\rho \in \mathbf{Env} = \mathbf{Var}_* \to \mathbf{Loc}$ Umgebungen
$\zeta \in \mathbf{Store} = \mathbf{Loc} \to_{fin} \mathbf{Z}$ Speicher

In diesem Fall ist Var_* die endliche Menge von Variablen, die im Programm auftaucht und Store bezeichnet die Menge der Funktionen von Loc nach Z, die einen endlichen Definitionsbereich haben. Da nun die davor genutzten Zustände $\sigma \in \text{State} = \text{Var}_* \to \text{Z}$ von den Abbildungen ρ und ζ ersetzt werden können sie wie folgt rekonstruiert werden:
Um den Wert einer Variablen x zu erhalten bestimmen wir ihre Adresse $\zeta = \rho(x)$ und daraufhin den Wert $\zeta(\xi)$, der in dieser Adresse gespeichert ist. Damit das funktioniert, ist es wichtig, dass $\zeta o \rho$: $\text{Var}_* \to \text{Z}$ eine totale Funktion ist. Das bedeutet, dass $\text{ran}(\rho)$ eine Teilmenge von $\text{dom}(\zeta)$ ist, wenn $\text{ran}(\rho) = \{\rho(\mathtt{x}) \in \mathbf{Var}_*\}$ sowie $\mathtt{dom}(\zeta) = \{\xi \mid \zeta \text{ ist definiert über } \xi\}$
Wenn nun ein Programm abearbeitet wird, wird immer, wenn einer Variablen ein neuer Wert zugewiesen wird eine neue Adresse erstellt, die die benötigten Informationen enthält, und auf einen Stack gelegt.

2 Intraprozedurale und Interprozedurale Analyse

2.1 Ein einfacher Ansatz

Um zu verstehen, weshalb interprozedurale Analyse komplexer ist als intraprozedurale kann man versuchsweise die Techniken der vorhergegangenen Kapitel mit den Erweiterungen anwenden. Man setzt fest, dass man für jeden Prozeduraufruf zwei Transferfunktionen flc und flr entsprechend Eintritt in und Austritt aus dem Prozedurenrumpf hat. Man nimmt nun ein Monotones Framework(L...) und behandelt die Flüsse $(l_c; l_n)$ und $(l_x; l_r)$ wie den bisher bekannten (l_1, l_2). Als Nächstes nimmt man an, dass alle vier eben genannten Transferfunktionen die Identitätsfunktion sind, wodurch die Parameterübergabe vollständig außer acht gelssen wird, was möglich ist, da ein Monotones Framework alle Transferfunktionen frei interpretieren kann. Dadurch erhält man ein Gleichungssystem wie in den vorhergehenden Kapiteln:

$$A_\bullet(l) = f_l(A_o(l))$$
$$A_o(l) = \sqcup\{A_\bullet(l') \mid (l', l) \in \text{F oder } (l'; l) \in F\} \sqcup \iota^l_E$$

1. Der Status nach dem Label l ist der Status vor dem Label l nach Anwendung der Transferfunktion l
2. Der Status vor dem Label l ist der Status nach dem Vorgängerlabel von l

Wenn man sich dieses Gleichungssystem nun genau anschaut, ist es offensichtlich, dass es keine Sicherheit dafür gibt, dass Information, die von einem Aufruf über $(l_c; l_n)$ und dann von der Prozedur über $(l_x; l_r)$ wieder zum ursprünglichen Aufruf zurückfließt. Um es an einem Beispiel klar zu machen: nichts verhindert den Pfad [9,1,2,4,1,2,3,8,10], der aber vom Programm nicht

durchlaufen werden kann. Das Gleichungssystem enthält zwar auch die korrekten Pfade, jedoch neben diesen viel zu viele falsche, weswegen es sehr unpräzise ist.

2.2 Valide Pfade

Um diese falschen Pfade auszuschließen will man versuchen sich auf die Pfade zu fokussieren, die die korrekte Kapselung von Prozeduraufrufen und Rückgaben haben. Dazu greifen wir nun die früher besprochene MOP-Lösung auf und verfeinern sie so, dass nur noch die validen Pfade berücksichtigt werden, wodurch wir die MVP-Lösung definiert haben.
Gegeben ein Programm $\texttt{P}_*$ der Form `begin` D_* S_* `end`
Ein Pfad ist ein vollständiger Pfad (complete Path) von l_1 nach l_1, wenn er eine korrekte Kapselung aufweist. Diese Pfade werden vom Nicht-Terminalsymbol CP_{l_1,l_2} anhand dieser Produktionen erstellt. Die letzte Produktion sichert die korrekte Kapselung, indem eine Klammerung erzwungen wird, sodass l_c, l_n nur im erstellten Pfad auftaucht, wenn es das kooperierende l_x, l_r gibt und umgekehrt. Ein Pfad wird "valider Pfad" genannt, wenn er an einem Extrempunkt von P_* beginnt und alle Prozedurausgänge zu den -eingängen passen, aber es möglich ist, dass manche Prozeduren betreten, aber nicht verlassen wurden. Das umfasst alle vollständigen Pfade, die in E starten, aber ebenso Berechnungen, die möglicherweise nicht terminieren. Hierfür bilden wir folgende Grammatik:

$$CP_{l_1,l_2} \rightarrow l_1$$
$$CP_{l_1,l_3} \rightarrow l_1, CP_{l_2,l_3}$$
$$CP_{l_c,l} \rightarrow l_c, CP_{l_n,l_x}, CP_{l_r,l}$$

Die letzte Produktion garantiert eine korrekte Klammerung

Das Nichtterminalsymbol VP_* generiert alle validen Pfade. Bei einer Vorwärtsanalyse, die vom Label l_c eines Prozeduraufrufs bis zum Programmpunkt l reicht gibt es 2 Möglichkeiten. Entweder bei l_c getätigte Aufruf terminiert bevor l erreicht wird, oder l wird erreicht bevor der Aufruf terminiert. Mit anderen Worten: entweder l befindet sich im Code nachdem die bei l_c aufgerufene Prozedur vollständig abgelaufen ist (nach l_r) oder l befindet sich innerhalb der Prozedur (zwischen l_c und l_r). Die erste Möglichkeit wird von der vorletzten Produktion realisiert, die das Nichtterminalsymbol CP_{l_n,l_x} verwendet um den Rumpf der Prozedur zu generieren, während die letzte Produktion entsprechen der zweiten Möglichkeit einen validen Pfad innerhalb des Rumpfes erzeugt.

Dies ist die Grammatik der Validen Pfade:

$$VP_* \rightarrow VP_{l_1,l_2}$$
$$VP_{l_1,l_2} \rightarrow l_1$$
$$VP_{l_1,l_3} \rightarrow l_1, VP_{l_2,l_3}$$
$$VP_{l_c,l} \rightarrow l_c, CP_{l_n,l_x}, VP_{l_r,l}$$
$$VP_{l_c,l} \rightarrow l_c, VP_{l_n,l}$$

Ganz offensichtlich sind die Mengen kleiner, als wenn man $(l_c; l_n)$ und $(l_c; l_n)$ gleichgesetzt und die gerade vorhin dargestellten Definitionen verwendet hätte.
Die MVP Solution wird nun wie folgt deklariert:

$$MVP_o(l) = \sqcup\{f_{\bar{l}}(\iota) \in vpath_o(l)\}$$
$$MVP_\bullet(l) = \sqcup\{f_{\bar{l}}(\iota) \in vpath_\bullet(l)\}$$

1. MVP(l) ist die Vereinigung der Pfade, die beim Extrempunkt beginnen und bis vor l reichen
2. MVP(l) ist die Vereinigung der Pfade, die beim Extrempunkt beginnen und bis nach l reichen.
Da diese Pfadmengen kleiner sind als die aus Kapitel 2.4.2 ergibt sich für alle l:

$$MVP_o(l) \sqsubseteq MOP_o(l) \text{ und } MVP_\bullet(l) \sqsubseteq MOP_\bullet(l)$$

Die MVP Lösung ist möglicherweise nicht entscheidbar für Lattices von endlicher Höhe, wie die MOP-Lösung, weswegen wir jetzt die MFP Lösung wieder aufgreifen und überlegen, wie man nicht zu viele invalide Pfade wählt.
Deswegen hat man sich überlegt Informationen über den gewählten Pfad in die Datenflusseigenschaften selbst zu kodieren.

Der Kontext selbst kann beispielsweise einfach eine Kodierung des gewählten Pfades sein. Im Folgenden wird das Monotone Framework erweitert, sodass es den Kontext mit beachtet. Gegeben sei ein Monotones Framework $(L, F, \mathrm{F}, E, \iota, f_.)$ konstruiert man ein erweitertes Monotones Framework $(\hat{L}, \hat{F}, \mathrm{F}, E, \hat{\iota}, \hat{f}_.)$ das den Kontext berücksichtigt. Beginnen wir mit dem einfachen Teil der Definition - dem Teil der unabhängig von der tatsächlichen Wahl des Kontextes Δ ist: dem Teil der interprozeduralen Analyse.
Die Lattice wird mit dem Kontext verknüpft: $\widehat{L} = \Delta \rightarrow L$;
Die Transferfunktionen in $\widehat{F}$ sind monoton.
Jede Transferfunktion $\widehat{f}_l$ ist definiert durch $\widehat{f}_l(\widehat{l})(\delta) = f_l(\widehat{l}(\delta))$

Lässt man die Prozeduren außer Acht, nehmen die Datenflussgleichungen wieder die bekannte Form an.

2.3 Der Interprozedurale Anteil

Nun fehlen noch die Datenflussgleichungen entsprechend den Prozeduren. Für eine Prozedurdefinition gibt es zwei Transferfunktionen:

$$\widehat{f_{l_n}}, \widehat{f_{l_x}} : (\Delta \to L) \to (\Delta \to L)$$

Im Falle unserer einfachen Sprache definieren wir beide Funktionen als Identitätsfunktion

$$\widehat{f_{l_n}}, \widehat{f_{l_x}} = \widehat{l}$$

für alle $\widehat{l} \in L$, da die Auswirkungen des Prozedureintritts und des Prozedurausgangs von der Transferfunktionen für den Prozeduraufruf respektive der Prozedurrückgabe behandelt werden. Bei komplexeren Sprachen, in denen beim Eintritt und Ausgang viele semantische Aktionen stattfinden, sollte man dies überdenken.

Für einen Prozeduraufruf $(l_c, l_n, l_x, l_r) \in IF$ werden zwei Transferfunktionen deklariert, wobei wir uns hier nur auf die Vorwärtsanalyse beschränken wollen.
Die Transferfunktion für den Aufruf

$$\widehat{f^1_{l_c}} : (\Delta \to L) \to (\Delta \to L)$$

wird in der Gleichung

$$A_\bullet(l_c) = \widehat{f^1_{l_c}}(A_o(l_c)) \text{ für alle } (l_c, l_n, l_x, l_r) \in IF$$

verwendet. Das bedeutet die Transferfunktion modifiziert die Datenflusseigenschaften (inkl. Kontext) so, dass sie vom Prozedureintrittspunkt akzeptiert werden. Für die Rückgabe erhält man die Funktion

$$\widehat{f^2_{l_c,l_r}} : (\Delta \to L) \times (\Delta \to L) \to (\Delta \to L)$$

die in der Gleichung

$$A_\bullet(l_r) = \widehat{f^2_{l_c,l_r}}(A_o(l_c), A_o(l_r)) \text{ für alle } (l_c, l_n, l_x, l_r) \in IF$$

verwendet wird.

Der erste Parameter beschreibt die Datenflusseigenschaften am Punkt des

Aufrufes der Prozedur und der zweite Parameter die am Ausgang des Prozedurrumpfes.
Lässt man den ersten Parameter weg, so modifiziert die Transferfunktion die Datenflusseigenschaften (inkl. Kontext) wie sie für die Rückgabe vom Prozedurenausgang benötigt werden. Der Sinn des ersten Parameters ist es einen Teil der Informationen, nämlich sowohl Datenflusseigenschaften als auch Kontext, die vor dem eigentlichen Aufruf vorhanden waren weiterhin verfügbar zu machen. Wie das speziell durchgeführt wird, hängt von der tatsächlich gewählten Menge Δ ab˙
Die Funktionalität und Benutzung der Transferfunktion $\widehat{f_{l_c,l_r}}$ ist allgemein genug um sie an die meisten Szenarien anzupassen. Ein einfaches Beispiel wäre, dass man die Informationen, die vor dem Aufruf vorhanden sind einfach weglässt (siehe Anhang):

$$\widehat{f^2_{l_c,l_r}}(\widehat{l},\widehat{l'}) \;=\; \widehat{f^2_{l_r}}(\widehat{l'})$$

Eine interessante Variation ist die Möglichkeit zu definieren:

$$\widehat{f^2_{l_c,l_r}}(\widehat{l},\widehat{l'}) \;=\; \widehat{f^{2A}_{l_c,l_r}}(\widehat{l}) \sqcup \widehat{f^{2B}_{l_c,l_r}}(\widehat{l'})$$

Was eine Kombination der Informationen vor dem Aufruf und der Information die vom Aufruf zurückkommt, erlaubt (siehe Anhang).

2.4 Kontextsensitivität und Kontextinsensitivität

Bisher haben wir den “naiven Ansatz” kritisiert, weil kein angemessener Zusammenhang zwischen Prozeduraufrufen und Prozedurrückgaben aufgebaut wird.
Ein ähnlicher Kritikpunkt ist, dass nicht zwischen verschiedenen Aufrufen (andere Labels) einer Prozedur unterschieden werden kann. Die Information die in den Zuständen zu denen eine Prozedur aufgerufen wird vorhanden ist, wird gebündelt. Der Rumpf der Prozedur wird nur ein einziges mal mittels dieser gebündelten Information analysiert. Daraus resultiert, dass auch nur eine gebündelte Menge von Informationen in den Rückgabezuständen vorhanden ist. Dieser Umstand wird kontext-insensitiv genannt.
Die Nutzung einer komplexen Kontextinformation setzt beide Kritikpunkte außer Kraft. Betrachtet man zwei verschiedene Aufrufe, die aber beide in verschiedenen Kontexten δ_1 und δ_2 ausgeführt werden, dann wird jede Information, die man nach der Prozedur erhält eindeutig δ_1 bzw. δ_2 zugeordnet, wodurch keine unerwünschte Kombination oder Überschneidung entsteht. Dies wird kontext-sensitiv genannt.
Offensichtlich ist die kontext-sensitive Analyse präziser als die kontext-insensitive,

allerdings meist auch aufwendiger und teurer. Deswegen muss man bei der Wahl der richtigen Technik sorgfältig zwischen Effizienz und Genauigkeit abwägen.

3 Call Strings als Kontext

Um dieses Konzept der Programmanalyse zu vollenden muss nun noch konkret ein Kontext Δ, sowie der Extremwert $\widehat{\iota}$ festgesetzt werden. Außerdem müssen die Transferfunktionen für den Prozeduraufruf definiert werden. Im Folgenden wird auf zwei Ansätze die Call Strings als Kontext verwenden eingegangen, wobei nur die Vorwärtsanalyse in Betracht gezogen wird.

3.1 Call Strings von unbegrenzter Länge

Da in diesem Kapitel das Hauptinteresse auf den Prozeduraufrufen liegt, kodieren wir hier von dem gewählten Pfad nur die Labels der Prozeduraufrufe, genauer gesagt die Flüsse der Form $(l_c; l_n)$, was formal bedeutet:

$$\Delta = \mathbf{Lab}^*$$

Das letzte Label l_c eines Prozeduraufrufs ist dabei am rechten Ende. Diese Ketten von Prozeduraufrufen werden Call Strings genannt, wobei der Extremwert $\widehat{\iota}$ definiert wird als:

$\widehat{\iota}(\delta) = \{\, \iota \text{ wenn } \delta = \Lambda$
$\perp$ andernfalls

Wobei Λ die Leere Menge ist, in diesem Fall am Anfang des Programms, wenn noch kein Prozeduraufruf gespeichert ist.

3.2 Call Strings von begrenzter Länge

Einleuchtenderweise können die Call Strings beliebig lang werden, nicht zuletzt aufgrund der erlaubten Rekursion. Deswegen ist es Usus eine bestimmte obere Grenze $k \geq 0$ für die Länge der Call Strings zu setzen, wobei dann nur die k letzten Labels gespeichert werden. Dies wird so geschrieben:

$$\Delta = \mathbf{Lab}^{\leq k}$$

Wobei der Extrempunkt immer noch gleich bleibt.
Der Spezialfall $k = 0$ ist gleich dem Fall dass gar keine Kontextinformation vorhanden ist.

Für den allgemeinen Fall, dass man Ketten maximal der Länge k hat nimmt die Transferfunktion $f^1_{l_c}$ diese Form an.

$$\widehat{f^1_{l_c}}(\widehat{l})(\delta') \;=\; \sqcup\{f^1_{l_c}(\widehat{l}(\delta))|\delta' \;=\; \lceil\delta, l_c\rceil_k\}$$

Wobei $\lceil\delta, l_c\rceil_k$ den Call String $[\delta, l_c]$ bezeichnet, der aber möglicherweise auf der linken Seite abgeschnitten ist, sodass er maximal k Elemente lang ist. (wenn die Kette ohnehin kleiner ist als k, wird auch nichts abgeschnitten). Da die Funktion, die *delta* auf $\lceil\delta, l_c\rceil_k$ abbildet nicht injektiv ist muss man hier die kleinste Menge über alle δ, die auf den relevanten Kontext δ' abgebildet werden können nehmen. Die Transferfunktion $f^2_{l_c,l_r}$ für die Prozedurrückgabe ist ähnlich neu definiert:

$$\widehat{f^2_{l_c,l_r}}(\widehat{l},\widehat{l'})(\delta) \;=\; f^2_{l_c,l_r}(\widehat{l}(\delta),\widehat{l'},(\lceil\delta, l_c\rceil_k))$$

4 Flusssensitivität und Flussinsensitivität

Alle bisher betrachteten Datenflussanalysen waren flusssensitiv. In klaren Worten bedeutet dies: Man erwartet von der Analyse des Programms S1;S2 ein anderes Ergebnis als von der Analyse des Programms S2;S1.
Wenn die Reihenfolge von Statements für die Analyse keine Rolle spielt, werden auch flussinsensitive Analysen durchgeführt. Man erwartet nun also von den Analysen der Programme S1;S2 und S2;S1 dasselbe Ergebnis. Offensichtlich ist eine flussinsensitive Analyse weit weniger präzise als das flusssensitive Gegenstück, aber dafür ist sie viel billiger. Da Interprozedurale Datenflussanalysen dazu neigen sehr kostspielig zu werden, ist es nützlich verschiedene Konzepte zur Kostensenkung parat zu haben.

Ein Beispiel für eine flussinsensitive Analyse wäre das Folgende: Gegeben sei ein Programm $\mathtt{P}_*$ der Form

$$\mathtt{begin}\,\mathtt{D}_*\,\mathtt{S}_*\mathtt{end}$$

Für jede Prozedur

$$\mathtt{D} ::= \mathtt{proc}\; p(\mathtt{val}\; x, \mathtt{res}\; y)\; \mathtt{is}^{\mathtt{l_n}}\; \mathtt{S}\; \mathtt{end}^{\mathtt{l_x}}$$

in D ist es das Ziel die Menge IAV(p) von globalen Variablen, die wenn p aufgerufen wird direkt oder indirekt einen Wert zugewiesen bekommen, zu berechnen. Um diese Menge zu erhalten braucht man zwei Hilfsmittel. Einmal die Menge AV der direkt aufgerufenen Variablen und die Menge CP der

direkt aufgerufenen Prozeduren.

$$AV([skip]^l) = \emptyset$$
$$AV([x := a]^l) = \{x\}$$
$$AV(S_1; S_2) = AV(S_1) \cup AV(S_2)$$
$$AV(if[b]^l then S_1 else S_2) = AV(S_1) \cup AV(S_2)$$
$$AV(while[b]^l do S) = AV(S)$$
$$AV([callp(a, z)]^{l_c}_{l_r}) = \{z\}$$

AV(S) gibt für jedes Statement die Menge der Variablen zurück, die in dem Statement eine Zuweisung bekommen könnten, ohne jedoch die Variablen innerhalb eines weiteren Prozeduraufrufes zu beachten (es bleibt quasi auf einer Ebene).

$$CP([skip]^l) = \emptyset$$
$$CP([x := a]^l) = \emptyset$$
$$CP(S_1; S_2) = CP(S_1) \cup CP(S_2)$$
$$CP(if[b]^l then S_1 else S_2) = CP(S_1) \cup CP(S_2)$$
$$CP(while[b]^l do S) = CP(S)$$
$$CP([callp(a, z)]^{l_c}_{l_r}) = \{p\}$$

CP(S) liefert die Menge aller Prozedurnamem, die in S aufgerufen werden, wobei weitere Prozeduraufrufe auch wieder nicht beachtet werden. Diese Methode ist offensichtlich flussinsensitiv, das die Anzahl der Variablen nicht durch Vertauschung der Reihenfolge verändert wird.
Die gewünschte Menge ergibt sich nun rekursiv aus dieser Formel:
$IAV(p) = (AV(\backslash\{x\}) \cup \bigcup\{IAV(p') | p' \in CP(S)\}$
wenn proc p(val x, res y) is^{l_n} S end^{l_x} in D_* enthalten ist

Gesucht ist hierbei wieder die kleinstmöglichste Lösung des Gleichungssystems.

5 Fazit

Letztendlich lässt sich sagen, dass die Erweiterung einer Analyse um Prozeduren ein durchaus wichtiger Aspekt bei der Optimierung von Programmen, beziehungsweise Compilern ist. Die Durchführung gestaltet sich jedoch oft kompliziert und ist verständlicherweise teurer als eine Analyse, die Prozeduren nicht in Betracht zieht. Aus diesem Grund ist es wichtig sich genau zu überlegen was man analysieren oder optimieren möchte und anhand diesen Überlegungen die Wahl der geeigneten Analyse zu treffen.

6 Anhang

6.1 Flussgraphendefinitionen

6.1.1 Für Programme

$init_*$ = $init(S_*)$
$final_*$ = $final(S_*)$
$blocks_*$ = $\bigcup$ {$blocks(p)$ | proc p(val x, res y) is^{l_n} S end^{l_x} is in D_*}

$labels_*$ = $\bigcup$ {$labels(p)$ | proc p(val x, res y) is^{l_n} S end^{l_x} is in D_*}
$flow_*$ = $\bigcup$ {$flow(p)$ | proc p(val x, res y) is^{l_n} S end^{l_x} is in D_*}

6.1.2 Für Prozeduren

$init$(p) = l_n
$final$(p) = {l_x}
$blocks$(p) = {is^{l_n}, end^{l_x}} $\cup$ $blocks(S)$
$labels$(p) = {l$_n$, l_x}$\cup$ $labels(S)$
$flow$(p) = {$(l_n, init(S))$} $\cup$ $flow(S)$ $\cup$ {(l,l_x) | $l \in final(S)$ }

6.2 Abbildungen

6.2.1

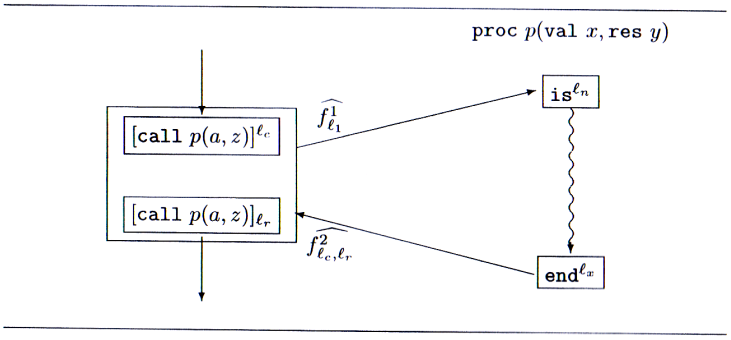

Figure 2.9: Analysis of procedure call: ignoring calling context.

6.2.2

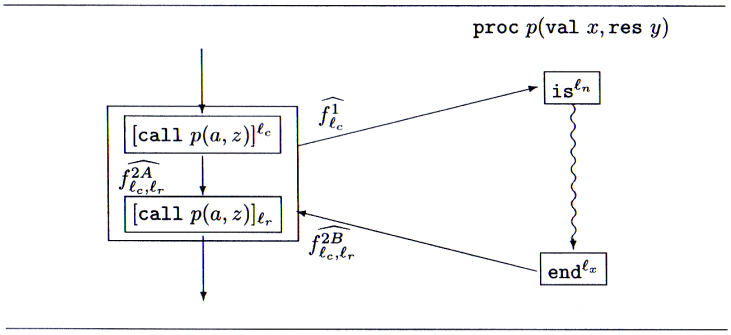

Figure 2.10: Analysis of procedure call: merging of context.