Prolog Compilierung

Interpreter ⟶ Compiler:

- Durch Partielle Auswertung des Interpreters

  Ein Interpreter hat zwei Eingaben: Programm und Eingabedaten. Wie kann man den Interpreter für ein gegebenes Programm spezialisieren? D.h. welche Teile kann man schon auswerten?

- Andere Vorstellung: Modifiziere Interpreter so, daß er, anstelle die vom Programm verlangten Aktionen (z.B. Addition, Zuweisung) direkt auszuführen, Befehle dafür ausgibt (erfordert natürlich spezielle Überlegungen bei Fallunterscheidungen, Schleifen, Prozeduraufrufen).

Warren Abstract Machine (WAM):

- Vorgeschlagen von D.H.D. Warren (1983)

  D.H.D. Warren hat 1977 den ersten Prolog-Compiler entwickelt.

- Zwischencode

  (Prolog → WAM → Maschinensprache).

  Vergleichbar mit P-Code für Pascal. WAM ist „de facto Standard".

- Viele Prolog-Compiler übersetzen nur in WAM-Code und interpretieren diesen dann.

- WAM: Register, Speicherbereiche, Instruktionen

- Register: Argument-Register, 10 spezielle Register (Stackpointer etc.)

- Speicherbereiche: WAM-Code, Heap, Stack (Activation Records + Variablen-Frames, Choice-Points), Trail, Unifikations-Stack.

- 39 Instruktionen (Maschinenbefehle).

Beispiel (1): Argument-Übergabe

Übersetzung des Rumpf-Literals p(a, X, X):

- Beim Aufruf von p/n: Argumente in Register r1–rn.
- putCon r1, a

  Speichert die Konstante a in das Register r1, d.h. $\mathrm{r1} := \langle \mathrm{CON}, \mathrm{a} \rangle$. Register und Speicherzellen sind „tagged" ($\cong$ varianten Records).
- putVar r2, #1

  Register E enthält Anfangsadresse des aktuellen „Environment Frame". Darin ab Offset 2 die Variablen, d.h. X (#1) an Adresse $\mathrm{E} + 2$. Es wird zu „unbound" initialisiert (in der WAM: Zeiger auf sich selbst, vgl. z.B. getVar). Also $*(\mathrm{E} + 2) := \langle \mathrm{REF}, \mathrm{E} + 2 \rangle$. Dann: $\mathrm{r2} := \langle \mathrm{REF}, \mathrm{E} + 2 \rangle$.
- putVal r3, #1 % $\mathrm{r3} := *(\mathrm{E} + 2)$.

Übersetzung des Kopf-Literals p(Y, b, Y):

- getVar r1, #1

  Der Wert von r1 wird in den 1. Platz des Variablen-Frames (Y) kopiert.
- getCon r2, b

  Ist der dereferenzierte Wert von r2 eine ungebundene Variable, so weise ihr b zu (und ihre Adresse auf Trail). Sonst backtracke, falls nicht b.
- getVal r3, #1

  Bisher war die Unifikation einfach, weil einer der beiden Terme zur Compilezeit vollständig bekannt war (ungebundene Variable bzw. bestimmte Konstante). Hier können in r3 und #1 aber beliebige Terme stehen. Deswegen muß die allgemeine Unifikation aufgerufen werden.

Beispiel (2): Strukturierte Terme

Übersetzung des Rumpf-Literals p(f(a,X)):

- putStr r1, f

  Im Register H steht die nächste freie Adresse im Heap. Dorthin wird f gespeichert ($*\mathrm{H} := \langle \mathrm{CON}, \mathrm{f}/2 \rangle$), dann: $\mathrm{r1} := \langle \mathrm{STR}, \mathrm{H} \rangle$; $\mathrm{H} := \mathrm{H} + 1$.
- Strukturen werden durch $\langle \mathrm{STR}, \mathrm{A} \rangle$ repräsentiert, an Adresse A (im Heap) steht der Funktor f/n, an Adresse $\mathrm{A} + 1$ bis $\mathrm{A} + \mathrm{n}$ die Argumente.
- setCon a

  $*\mathrm{H} := \langle \mathrm{CON}, \mathrm{a} \rangle$; $\mathrm{H} := \mathrm{H} + 1$. (In der WAM uniCon im write-Mode).
- setVar #1

  $*\mathrm{H} := \langle \mathrm{REF}, \mathrm{H} \rangle$; $*(\mathrm{E} + 2) := \langle \mathrm{REF}, \mathrm{H} \rangle$; $\mathrm{H} := \mathrm{H} + 1$.
- Bei Funktoren innerhalb von Funktoren funktioniert das so nicht, weil für ein Argument immer nur eine Zelle zur Verfügung steht. Lösung: Übersetze f(g(a),b) wie X=g(a), f(X,b).

Übersetzung des Kopf-Literals q(g(a,X,X)):

- getStr g, r1

  Falls r1 zu ungebundener Variable dereferenziert wird: Lege g auf Heap, binde Variable daran, setze Modus Register auf „write“. Falls Funktor g: Setze Modus auf „read“, Register S („Structure Pointer“) auf Adresse des ersten Arguments. Sonst: Backtracken.
- uniCon a

  Im „write“-Modus wie setCon. Im „read“-Modus: Dereferenziere Heap an Adresse S, weiter entsprechend getCon, schließlich $\mathrm{S} := \mathrm{S} + 1$.
- Dann uniVar #1 (X initialisieren) uniVal #1 (allgemeine Unifikation).

Beispiel (3): Ablauf-Steuerung

Übersetzung von p(X,Y,Z) :- q(Z), r(Y), s(Z):

- p:

  Erste Regel über p bekommt Label p:, folgende Regeln p_2, ...,p_m.

- alloc

  Das Register E zeigt noch auf das Environment des Aufrufers. Es wird inkrementiert auf den nächsten freien Platz im Stack. Dort wird der alte Wert von E und der Wert des Registers CP (s.u.) abgelegt.

- getVar r1, #1 % X
- getVar r2, #2 % Y
- getVar r3, #3 % Z
- putVal r1, #3
- call q:, 2

  Das Register CP (Program Continuation, Rücksprung-Adresse) wird auf die Adresse der folgenden Instruktion gesetzt (CP := P + 1) (P ist der Program Counter). Das aktuelle Environment wird auf 2 Variablen verkürzt (Z ist nicht mehr nötig). Dann wird zum Label q: gesprungen.

- putVal r1, #2
- call r:, 1 % Nur noch die Variable X (#1) wird benötigt.
- putVal r1, #1
- dealloc

  Das aktuelle Environment wird freigegeben, dabei werden E und CP wieder auf ihre alten Werte gesetzt („Last Call Optimization“).

- exec s:

  Es wird zum Label s: gesprungen, d.h. P := „Adresse von s:“.

Beispiel (4): Stack-Benutzung

Übersetzung von s(a):

- s: getCon r1, a

  Fakten brauchen keinen Environment-Frame, also „alloc“ nicht nötig.

- proceed

  Es wird zu der Rücksprungadresse in CP gesprungen (P := CP.)

Übersetzung von p(X,Y) :- q(X,Z), p(Z,Y):

- p: alloc
- getVar r2, #1

  X wird nicht im Stack, sondern in einem Register (r1) gehalten („temporäre Variable“). Möglich, weil X nicht über „call“ hinaus nötig.

- putVar r2, #2

  X steht schon in r1, es braucht also überhaupt nicht kopiert zu werden.

- call q:, 2
- putUVal r1, #2

  „put unsafe value“: wie putVal, aber: Wenn Z von q nicht gebunden wurde, würde r1 anschließend eine Adresse aus dem aktuellen Stackframe enthalten, der gleich freigegeben wird. Dann legt putUVal auf dem Heap eine neue Variable an, und weist r1 deren Adresse zu.

- putVal r2, #1
- dealloc
- exec p:

Beispiel (5): Backtracking

Übersetzung mehrerer Regeln über p/2:

- p: tryMeElse 2, p_2:

  Dieser Befehl legt einen Choice-Point-Frame auf dem Stack ab. Alle Environment-Frames darunter sind geschützt und werden nicht freigegeben, solange dieser Choice-Point-Frame existiert. Im Choice-Point-Frame werden folgende Register gesichert: r1, r2 (entsprechend der Stelligkeit 2 von p), E, CP, H, TR (Trail-Top), B („Backtrack“, letzter Choice-Point-Frame). Außerdem enthält er die Adresse p_2: der nächsten Alternative.

- Code für erste Regel, z.B.:
  alloc
  getCon r1, a
  getVar r2, #1
  ...

- Falls Backtracking nötig: springe zu Alternativ-Adresse in oberstem Choice-Point-Frame. Seine Adresse steht in Register B.

- p_2: retryMeElse 2, p_3:

  Dieser Befehl wird nur durch das Backtracking angesprungen. Er stellt den durch „TryMeElse“ gesicherten Zustand wieder her (außer Register B selbst). Die auf dem Trail gemerkten Variablen werden wieder ungebunden gemacht. Anschließend muß nur die nächste Alternative im Choice-Point-Frame auf „p_3:“ gesetzt werden.

- Code für zweite Regel

- p_3: trustMeElseFail 2

  Wie „retryMeElse“, aber Choice-Point-Frame anschließend löschen.

- Code für letzte Regel

Zusammenfassung zur WAM (1)

Format von Speicherworten:

- $\langle \mathrm{CON}, \mathrm{C} \rangle$: Konstante (bzw. Funktor). C identifiziert den Funktor mit Namen und Stelligkeit. Auch integers etc. werden so repräsentiert.
- $\langle \mathrm{REF}, \mathrm{A} \rangle$: Variable (gebunden/ungebunden). A: Adresse des Terms, an den die Variable gebunden ist. Ungebunden falls eigene Adresse.
- $\langle \mathrm{STR}, \mathrm{A} \rangle$: A ist die Adresse der Funktorzelle einer Struktur. In den n folgenden Zellen sind die n Argumente abgespeichert.
- $\langle \mathrm{LIS}, \mathrm{A} \rangle$: Adresse eines Listenknotens (Head-Zelle, danach Tail-Zelle).

Speicherbereiche:

- Code-Bereich: Enthält das Programm aus WAM-Befehlen. Register P und CP zeigen in den Code-Bereich.
- Heap: Enthält konstruierte Terme (Strukturen oder Kopien unsicherer Variablen). Freigabe bei Backtracking in Stack-Manier. Register H, S (und HB) beziehen sich auf den Heap.
- Stack: Enthält Environments (Parent-Env., Rücksprung-Adresse, Variablenwerte) und Choice-Points (gesicherte Registerwerte, alternative Code-Adresse). Die Freigabe von Environments geschieht normalerweise vor dem Aufruf des letzten Rumpf-Literals, sofern nicht durch darüberliegenden Choice-Point geschützt. Choice-Points werden nur beim Backtracking freigegeben. Siehe Register E, B (und B0).
- Trail: Enthält Adressen von Variablen (REF-Zellen), die beim Backtracking ungebunden gemacht werden müssen. Die Freigabe geschieht beim Backtracking. Zum Trail gehört Register TR.
- PDL (Push-Down-List): Dieser Stack wird implizit von der allgemeinen Unifikationsprozedur benutzt. Er enthält Paare von noch zu unifizierenden Termen (bzw. Zeiger darauf).

Zusammenfassung zur WAM (2)

Register:

- P: „Program Counter“. Adresse des nächsten auszuführenden Befehls.
- CP: „Program Continuation“. Adresse des Befehls, zu dem nach Abarbeitung der aktuellen Prozedur zurückgekehrt werden soll.
- E: Anfangsadresse des aktuellen „Environment Frame“ auf dem Stack.
- B: „Backtrack Point“. Endadresse des letzten Choice-Point-Frames.
- Aus E und B kann man den „Top of Stack“ berechnen, sofern man die Anzahl Variablen im obersten Environment-Frame kennt. Diese steht aber im call-Befehl, auf den CP verweist (CP zeigt direkt dahinter).
- B0: „Cut Register“: Wert von B beim letzten Prozeduraufruf (für !).
- H: „Heap Pointer“ Nächste freie Adresse im Heap.
- S: „Structure Pointer“ (+ Modus Bit). Nächstes zu unifizierendes Argument einer Struktur. Siehe getStr, set/uni-Befehle.
- HB: Wert von H beim Anlegen des letzten Choice-Points. Nur darunter müssen Variablen-Bindungen auf dem Trail notiert werden.
- TR: „Trail Pointer“ Nächste freie Adresse auf dem Trail.
- r1, . . . , rn: Argument-Register. Enthalten Argumente des aktuellen Prozeduraufrufs und temporäre Variablen.

Klassen von Befehlen:

- put: Lade Argumente in Register.
- get: Unifiziere Argumente aus Registern mit Regelkopf.
- set, unify: Für Argumente von Strukturen.
- alloc, dealloc, call, exec, proceed: Kontrolle.
- tryMeElse etc.: Choice-Points anlegen, Backtracking.
- Weitere Instruktionen für Cut und Index über erstem Argument.