Der Grid-Occam-Compiler

Übersicht über die OCCAM-Sprachdefinition

OCCAM

- Entwicklung der Fa. Inmos (1988)
  - übernommen von SGS Thomson
  - heute STMicroelectronics
- “aktuelle” Sprachversion ist 2.1
- *die* Sprache für Transputer-Programmierung
  - Sprachentwicklung parallel zur Chipentwicklung
- OCCAM geht nicht von zentralem Hauptspeicher aus
  - Unterstützung der Entwicklung paralleler Programme, auf Basis paralleler Prozesse
- strenges Typsystem

Lexik

- Definitionsmittel für Lexik
  - englische Sprache
  - reguläre Ausdrücke
  - kontextfreie Grammatiken
- Occam: englische Sprache
- Zuordnung zwischen Bytes und Zeichen:
  - Occam: Eingabe ist ASCII

Lexikalische Kategorien

- Bezeichner: Buchstabe, gefolgt von Buchstaben, Zahlen (alphanumeric characters), und Punkten (.), ohne Längenbeschränkung
- Schlüsselwörter: Stets in Großbuchstaben, reserviert (etwa: SEQ, REAL64)
- Sonderzeichen: explizite Terminalsymbole in Grammatik
  - ":=", "(", ")" …
- "informale syntaktische Kategorien":
  - digits: Folge von Ziffern (0-9)
  - hex.digits: Folge von Hexadezimalziffern (0-9A-F)
- "besondere syntaktische Kategorien":
  - Zahlenliterale: definiert in EBNF, zusätzliche Beschränkung, dass sie keine Leerzeichen enthalten dürfen

Freiraum und Kommentare

- Üblicherweise ignoriert
  - 3*4 ist das gleiche wie 3 * 4
- Occam: Einrückung ist lexikalisch relevant
  - Zeilenenden sind ebenfalls lexikalisch relevant
  - Continuation Lines: Zeile setzt sich nach Operatoren, Kommas, Semikolons, Zuweisungsoperatoren und ausgewählten Schlüsselwörtern fort
    - nächste Zeile muss wenigstens genauso viel eingerückt sein
- Zeilenende-Kommentare, eingeleitet durch --
  - Kommentar muss wenigstens genauso eingerückt sein wie aktueller Block

Syntax: Sprachen und Grammatiken

- Alphabet A: Menge von Symbolen (Buchstaben, Terminale, *terminals*)
- A*: Menge aller Folgen von A
  - Leeres Wort: ε
- Sprache L: Menge von Sätzen
  - $L \subset A^*$
- Grammatik: Tupel (A, H, R, s)
  - H: Hilfssymbole (*nonterminals*)
  - Startsymbol $s \in H$
  - Regeln $R \subset H \times (A \cup H)^*$

Chomsky-Hierarchie

- Sprache vs. Grammatik (vs. Parser-Technologie)
- Typ 0: rekursiv aufzählbare Sprachen
  - Akzeptiert durch Turing-Maschine
- Typ 1: kontext-sensitive Sprachen
  - Linke Seite kann Folge von Hilfssymbolen sein
- Typ 2: kontext-freie Sprachen
  - Linke Seite ist ein Hilfssymbol
- Typ 3: reguläre Sprachen
  - Linke Seite ist Hilfssymbol
  - Rechte Seite ist Terminalsymbol, optional gefolgt von Hilfssymbol

Notationen für Grammatiken

- BNF: Backus-Naur-Form
  - Erstmalig verwendet für Algol 58 (Backus Normal Form)
- Regeln der Grammatik: Produktionen
- Linke-seite $\rightarrow$ rechte-seite
- Rechte-seite: Verknüpfung von Alternativen mittels |
- Alternative: Folge von Hilfs- und Terminal-Symbolen

BNF-Beispiel

Expr $\rightarrow$ Term | Expr **+** Term | Expr **-** Term

Term $\rightarrow$ Factor | Term ***** Factor | Term **/** Factor

Factor $\rightarrow$ **number** | **(** Expr **)**

Ableitung: Wiederholte Anwendung von Produktionen

BNF-Erweiterungen

- EBNF (N. Wirth)
- Optionale Symbolfolgen: [sym1 sym2]
- Kleene-Stern: H*
  - Optionale Gruppierung von Symbolen mit runden Klammern
  - Ausschluss von ε: H+
- Occam: Nicht-EBNF-Erweiterung von BNF
  - Wiederholung auf mehreren Zeilen: { H }
  - Wiederholung auf der gleichen Zeile, kommagetrennt, optional: $\{_0$, H }
  - Wiederholung auf der gleichen Zeile, wenigstens einmal: $\{_1$, H }
  - Einrückung in der Grammatikregel ist signifikant

Definition von Semantik

- Statische Semantik: Welche Sätze der Sprache sind korrekte Programme?
- Dynamische Semantik: Was ist das beobachtete Verhalten?
- Occam: ILLEGAL vs. INVALID
  - ILLEGAL: Implementierung lehnt Programm ab
  - INVALID: Prozess ist äquivalent zu STOP
  - Annex E: invalid oder illegal?
- Stop-Modi: Annex F
  - stop process mode
  - halt system mode
  - undefined mode

Primitive Prozesse: Zuweisung

assignment == *variable.list* **:=** *expression.list*

expression.list == {$_1$ *expression* }

| *name* **(** {$_0$ **,** *expression* } **)**

| **(** value.process **)**

- legal: Variablen müssen definiert sein, Datentyp von Variable muss mit dem des Ausdrucks übereinstimmen
- dynamische Semantik: Der Ausdruck wird berechnet; die Variable erhält den ermittelten Wert

Zuweisung: Beispiele

x := y + 2

a, b, c := x, y + 1, z + 2

x, y := y, x

Primitive Prozesse: Input

input == *channel* **?** {$_1$ **;** *input.item* }
| *channel* **?** CASE *tagged.list*
| *timer.input*
| *delayed.input*
| *port* **?** *variable*

input.item == *variable*
| *variable* **::** *variable*

- legal: Kanäle, Variablen müssen definiert sein; Variablentyp muss mit Kanalprotokoll übereinstimmen
- dynamische Semantik: Prozess verharrt in Input, bis korrespondierendes Output erreicht ist, dann erfolgt Datenübergabe an Variable

Primitive Prozesse: Output

output == *channel* **!** {$_1$ **;** *output.item* }
| *channel* **!** *tag*
| *channel* **!** *tag* **;** {$_1$ **;** *output.item* }
| *port* **!** *expression*

output.item == *output*
| *output* **::** *variable*

Primitive Prozesse: SKIP und STOP

process == assignment
| input
| output
| SKIP
| STOP

- legal: keine weiteren Bedingungen
- dynamische Semantik
  - SKIP: kein Effekt
  - STOP: Prozess terminiert nicht
    - genauer: geht in den Stop-Modus über

SKIP, STOP: Beispiele

```
SEQ
    keyboard ? char
    SKIP
    screen ! char

SEQ
    keyboard ? char
    STOP
    screen ! char
```

Konstruierte Prozesse: Sequence

sequence == **SEQ**
{ *process* }
| **SEQ** *replicator*
process

replicator == *name* **=** *base* **FOR** *count*

base == *expression*

count == *expression*

- legal: *base* und *count* haben Typ **INT**, name darf nicht geändert werden
  - *name* hat Typ **INT**
- dynamische Semantik
  - valid: *count* muss nichtnegativ sein
  - Enthaltene Prozesse werden sequentiell ausgeführt
  - Wiederholtes **SEQ**: *name* fängt bei *base* an, *process* wird *count*-mal wiederholt
    - Verschachtelte Konstrukte werden expandiert

SEQ: Beispiele

```
SEQ
  SEQ
    screen ! '?'
    keyboard ? char
  SEQ
    screen ! char
    screen ! cr
    screen ! lf
```

SEQ: Beispiele (2)

```
SEQ i = 0 FOR array.size
    stream ! data.array[i]

SEQ
    stream ! data.array[14]
    stream ! data.array[15]
```

Konstruierte Prozesse: Conditional

conditional == **IF**
{ *choice* }
| **IF** *replicator*
choice

choice == *guarded.choice*
| *conditional*
| *specification*
choice

guarded.choice == *boolean*
process

boolean == *expression*

- legal: boolean muss vom Typ BOOL sein

Conditional: Dynamische Semantik

- *boolean*-Ausdrücke werden von oben nach unten ausgewertet
- falls ein Ausdruck wahr ist, wird der zugehörige *process* abgearbeitet
- falls kein Ausdruck wahr ist, ist der *conditional*-Prozess äquivalent zu STOP
- verschachtelte *conditional*s werden expandiert

Conditional: Beispiele

```
IF
    x < y
      x := x + 1
    x >= y
      SKIP

IF
    IF i = 1 FOR length
      string[i] <> object[i]
        found := FALSE
    TRUE
      found := TRUE
```

Konstruierte Prozesse: Selection

selection == **CASE** *selector*
{ *option* }
selector == *expression*
option == {$_1$ **;** *case.expression* }
process
| **ELSE**
process
| *specification*
option
case.expression == *expression*

- legal: Datentyp von *selector* und *case.expression* müssen gleich sein, müssen integer, byte, oder boolean sein, *case.expression* müssen verschiedene Konstanten sein, es darf höchstens ein ELSE geben

Selection: Dynamische Semantik

- selector wird berechnet und mit case.expression verglichen
- passender *process* ausgewählt
- Falls es keinen passenden Prozess gibt, wird **ELSE**-*process* ausgeführt
- Falls es keinen **ELSE**-*process* gibt, wird wird **STOP** ausgeführt.

Selection: Beispiele

```
CASE direction
    up
      x := x + 1
    down
      x := x - 1
CASE letter
    'a', 'e', 'i', 'o', 'u'
      vowel := TRUE
    ELSE
      vowel := FALSE
```

Konstruierte Prozesse: WHILE loop

loop == **WHILE** *boolean*
process

- Dynamische Semantik: *process* wird ausgeführt bis *boolean* **FALSE** ist

```
WHILE buffer <> eof
    SEQ
      in ? buffer
      out ! buffer
```

Konstruierte Prozesse: Parallel

parallel == **PAR**
{ *process* }
| **PAR** *replicator*
process
| **PRI PAR**
{ *process* }
| **PRI PAR** *replicator*
process

Parallel: valid vs. legal

- Eine Variable darf nicht in mehreren Prozessen geschrieben werden
  - 2.5.1: Verletzung führt zu INVALID program
  - Annex E (usage rules check list): Verwendung gleicher Variablen ist illegal
    - "usage checking can usually be performed at compile time."
- Ein Kanal darf nur in einem Prozess für output und in einem Prozess für input verwendet werden
  - CSP: Viele Schreiber, viele Leser
- Dynamische Semantik: Die Prozesse werden nebenläufig ausgeführt

Parallel: Beispiele

```
PAR
  farmer ()
  PAR i = 0 FOR 4
    worker (i)
```

Konstruierte Prozesse: Alternation

alternation == **ALT**
{ *alternative* }
| **ALT** *replicator*
alternative
| **PRI ALT**
{ *alternative* }
| **PRI ALT** *replicator*
alternative

alternative == *guarded.alternative*
| *alternation*
{ *alternative* }
| *channel* **? CASE**
{ *variant* }
| *specification*
alternative

Konstruierte Prozesse: Alternation (2)

guarded.alternative == *guard*
process

guard == *input*
| *boolean* **&** *input*
| *boolean* **& SKIP**

- dynamische Semantik: Falls *boolean* **TRUE** ist und *input* bereit, wird *input* ausgeführt, und danach der zugehörige *process*
  - sind mehrere Alternativen bereit, wird eine beliebige ausgewählt
  - bei **PRI ALT** wird die textuell erste ausgewählt

Alternative: Beispiele

```
ALT
    left ? packet
       stream ! packet
    right ? packet
       stream ! packet
```

Alternative: Beispiele (2)

```
ALT
    ALT i = 0 FOR number.of.workers
      free.worker[i] & gen.to.reg ? packet
        SEQ
          to.workers[i] ! packet
          free.worker[i] := FALSE
    ALT i = 0 FOR number.of.workers
      from.workers[i] ? result
        SEQ
          reg.to.gen ! result
          free.worker[i] := TRUE
```