

Für die bedingte Verzweigung verifizieren wir:

$$\mathbf{WP}[[x>1]] (B, D) \equiv (x \leq 1 \wedge l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}) \vee \\ (x > 1 \wedge x = l - 1 \wedge m_1 = m_0 = 1)$$

$$\Leftarrow x > 0 \wedge x = l - 1 \wedge m_0 = m_1 = 1$$

:-))

1.6 Prozeduren mit lokalen Variablen

- Prozeduren `f()` modifizieren globale Variablen.
- Die Werte der lokalen Variablen des Aufrufers **vor** und **nach** dem Aufruf sind unverändert **:-)**

Beispiel:

```
{int y= 17; double(); write(y);}
```

Vor und nach dem Aufruf von `double()` gilt: $y = 17$ **:-)**

- Der Erhaltung der lokalen Variablen tragen wir **automatisch** Rechnung, wenn wir bei der Aufstellung der globalen Hypothese beachten:
 - Die Vor- und Nachbedingungen: $\{A\}, \{B\}$ für Prozeduren sprechen nur über globale Variablen !
 - Die h werden nur für die **globalen** Variablen eingesetzt !!

- Der Erhaltung der lokalen Variablen tragen wir **automatisch** Rechnung, wenn wir bei der Aufstellung der globalen Hypothese beachten:
 - Die Vor- und Nachbedingungen: $\{A\}, \{B\}$ für Prozeduren sprechen nur über globale Variablen !
 - Die h werden nur für die **globalen** Variablen eingesetzt !!
- Als neuen Spezialfall der Adaption erhalten wir:

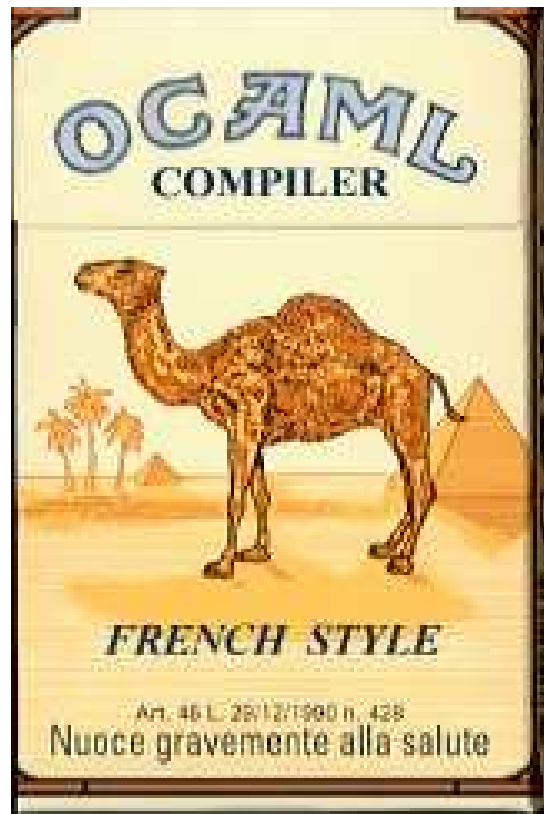
$$\frac{\{A\} \text{ f}(); \{B\}}{\{A \wedge C\} \text{ f}(); \{B \wedge C\}}$$

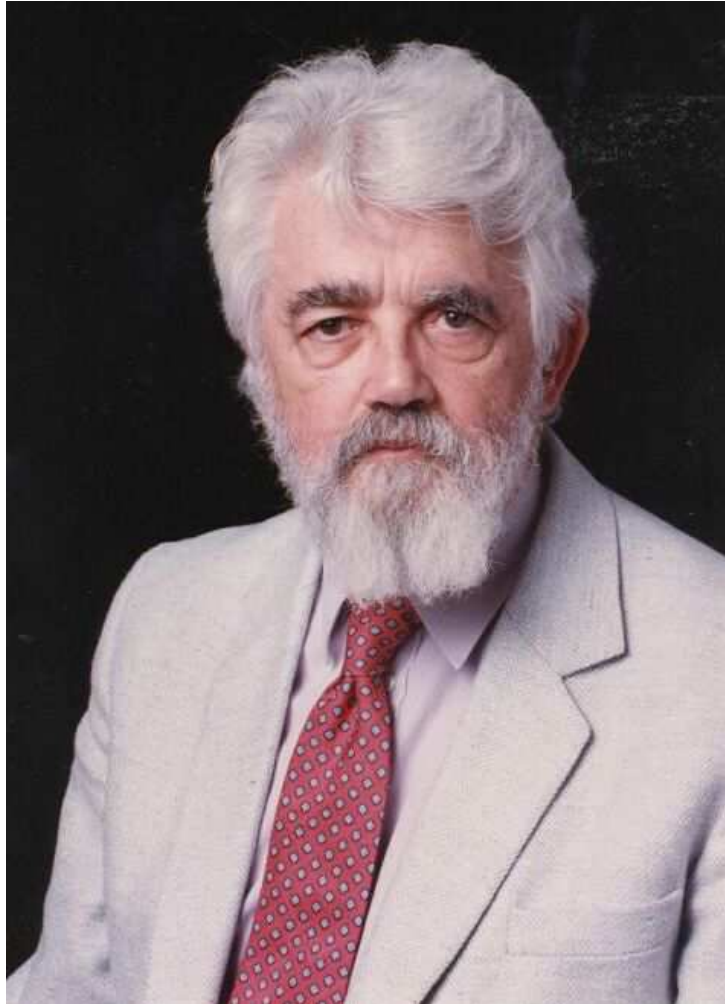
falls C nur über logische Variablen oder lokale Variablen des Aufrufers spricht :-)

Abschluss:

- Jedes weitere Sprachkonstrukt erfordert neue Methoden zur Verifikation :-)
- Wie behandelt man dynamische Datenstrukturen, Objekte, Klassen, Vererbung ?
- Wie geht man mit Nebenläufigkeit, Reaktivität um ??
- Erlauben die vorgestellten Methoden alles zu beweisen \implies Vollständigkeit ?
- Wie weit lässt sich Verifikation automatisieren ?
- Wieviel Hilfe muss die Programmiererin und/oder die Verifiziererin geben ?

Funktionale Programmierung





John McCarthy, Stanford



Robin Milner, Edinburgh



Xavier Leroy, INRIA, Paris

2 Grundlagen

- Interpreter-Umgebung
- Ausdrücke
- Wert-Definitionen
- Komplexere Datenstrukturen
- Listen
- Definitionen (Forts.)
- Benutzer-definierte Datentypen

2.1 Die Interpreter-Umgebung

Der Interpreter wird mit `ocaml` aufgerufen...

```
seidl@linux:~> ocaml
                Objective Caml version 3.09.3
#
```

Definitionen von Variablen, Funktionen, ... können direkt eingegeben werden **:-)**

Alternativ kann man sie aus einer Datei einlesen:

```
# #use "Hallo.ml";;
```

2.2 Ausdrücke

```
# 3+4;;  
- : int = 7  
# 3+  
  4;;  
- : int = 7  
#
```

- Bei `#` wartet der Interpreter auf Eingabe.
- Das `;;` bewirkt Auswertung der bisherigen Eingabe.
- Das Ergebnis wird berechnet und mit seinem Typ ausgegeben.

Vorteil: Das Testen von einzelnen Funktionen kann stattfinden, ohne jedesmal neu zu übersetzen :-)

Vordefinierte Konstanten und Operatoren:

Typ	Konstanten: Beispiele	Operatoren
int	0 3 -7	+ - * / mod
float	-3.0 7.0	+. -. *. /.
bool	true false	not &&
string	"hallo"	^
char	'a' 'b'	

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

```
# -3.0/.4.0;;
- : float = -0.75
# "So"^" "^"geht"^" "^"das";;
- : string = "So geht das"
# 1>2 || not (2.0<1.0);;
- : bool = true
```

2.3 Wert-Definitionen

Mit `let` kann man eine **Variable** mit einem Wert belegen.

Die Variable behält diesen Wert **für immer** :-)

```
# let seven = 3+4;;  
val seven : int = 7  
# seven;;  
- : int = 7
```

Achtung: Variablen-Namen werden **klein** geschrieben !!!

Eine erneute Definition für `seven` weist **nicht** `seven` einen neuen Wert zu, sondern erzeugt eine **neue** Variable mit Namen `seven`.

```
# let seven = 42;;  
val seven : int = 42  
# seven;;  
- : int = 42  
# let seven = "seven";;  
val seven : string = "seven"
```

Die alte Definition wurde **unsichtbar** (ist aber trotzdem noch vorhanden **:-)**)

Offenbar kann die neue Variable auch einen **anderen Typ** haben **:-)**

2.4 Komplexere Datenstrukturen

- Paare:

```
# (3,4);;  
- : int * int = (3, 4)  
# (1=2,"hallo");;  
- : bool * string = (false, "hallo")
```

- Tupel:

```
# (2,3,4,5);;  
- : int * int * int * int = (2, 3, 4, 5)  
# ("hallo",true,3.14159);;  
-: string * bool * float = ("hallo", true, 3.14159)
```

Simultane Definition von Variablen:

```
# let (x,y) = (3,4.0);;  
val x : int = 3  
val y : float = 4.
```

```
# let (3,y) = (3,4.0);;  
val y : float = 4.0
```

Records:

Beispiel:

```
# type person = {vor:string; nach:string; alter:int};;
type person = { vor : string; nach : string; alter : int; }
# let paul = { vor="Paul"; nach="Meier"; alter=24 };;
val paul : person = {vor = "Paul"; nach = "Meier"; alter = 24}
# let hans = { nach="kohl"; alter=23; vor="hans"};;
val hans : person = {vor = "hans"; nach = "kohl"; alter = 23}
# let hans_i = {alter=23; nach="kohl"; vor="hans"}
val hans_i : person = {vor = "hans"; nach = "kohl"; alter = 23}
# hans=hans_i;;
- : bool = true
```

Bemerkung:

- ... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)
- ... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.
- ... Typ-Namen und Record-Komponenten werden `klein` geschrieben :-)

Bemerkung:

- ... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)
- ... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.
- ... Typ-Namen und Record-Komponenten werden `klein` geschrieben :-)

Zugriff auf Record-Komponenten

... per Komponenten-Selektion:

```
# paul.vor;;  
- : string = "Paul"
```

... mit Pattern Matching:

```
# let {vor=x;nach=y;alter=z} = paul;;  
val x : string = "Paul"  
val y : string = "Meier"  
val z : int = 24
```

... und wenn einen nicht alles interessiert:

```
# let {vor=x} = paul;;  
val x : string = "Paul"
```

Fallunterscheidung: `match` und `if`

```
match n
  with 0 -> "Null"
       | 1 -> "Eins"
       | _ -> "Soweit kann ich nicht zaehlen!"
```

```
match e
  with true  -> e1
       | false -> e2
```

Das zweite Beispiel kann auch so geschrieben werden (-:

```
if e then e1 else e2
```