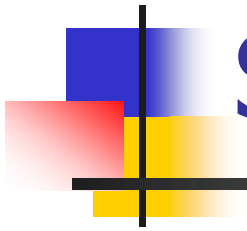
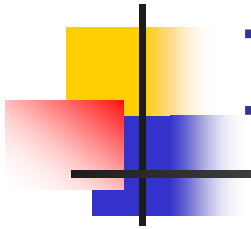


Spec# Boogie

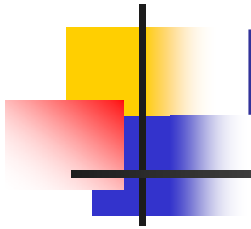
Formale Software-Entwicklung
Seminar SS ’07
Universität Karlsruhe

Stefan Hartte



Inhalt

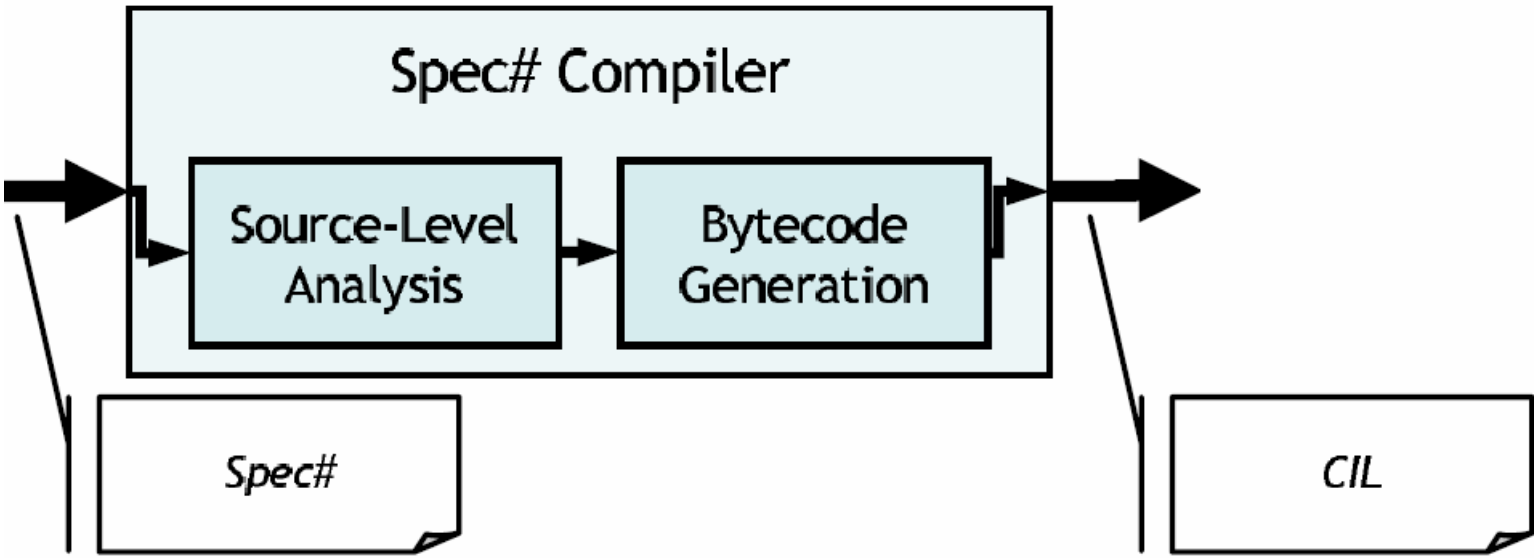
- Einführung
- Überblick
- BoogiePL
- Erzeugung von BoogiePL aus CIL
- Invariantenfolgerung
- Prüfbedingungserzeugung



Einführung

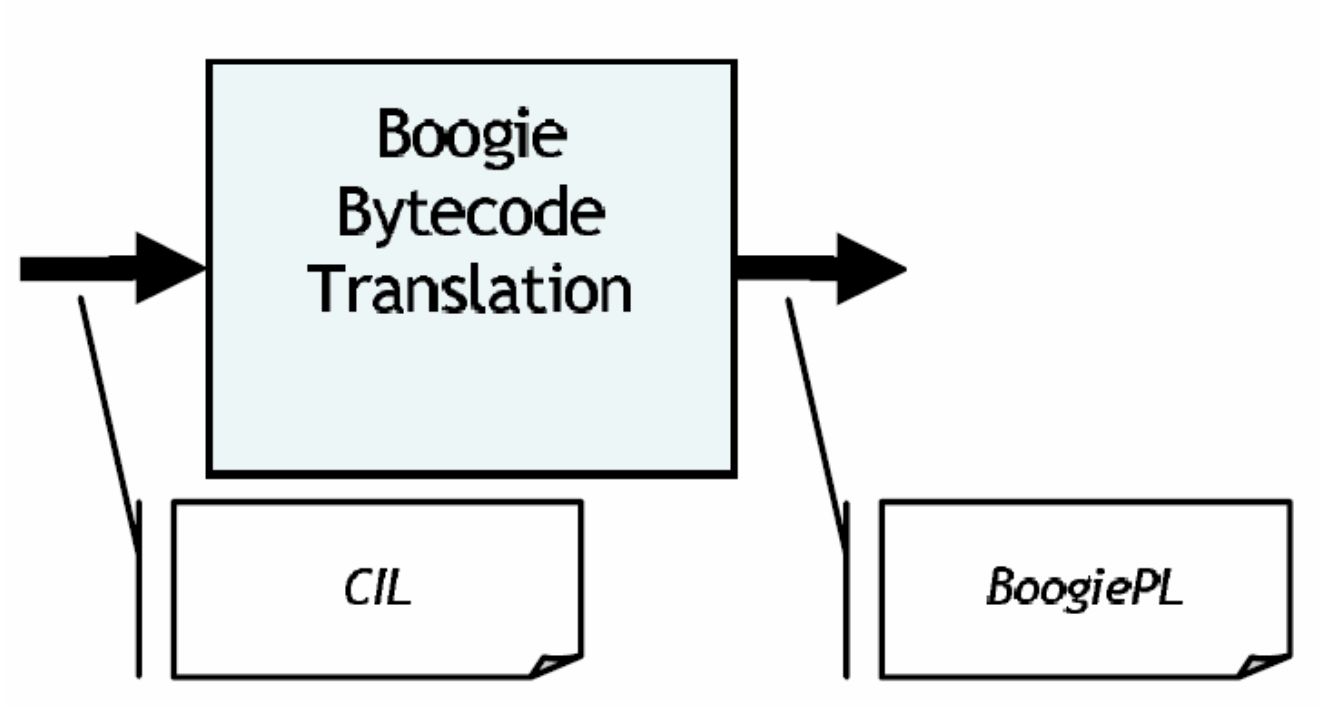
- Boogie ist ein statischer Verifizierer
- erkennt semantische Fehler zur Entwicklungszeit und meldet Fehler im Quelltexteditor
- unabhängig von Quellsprache, da er eigene Sprache benutzt (BoogiePL)

Überblick



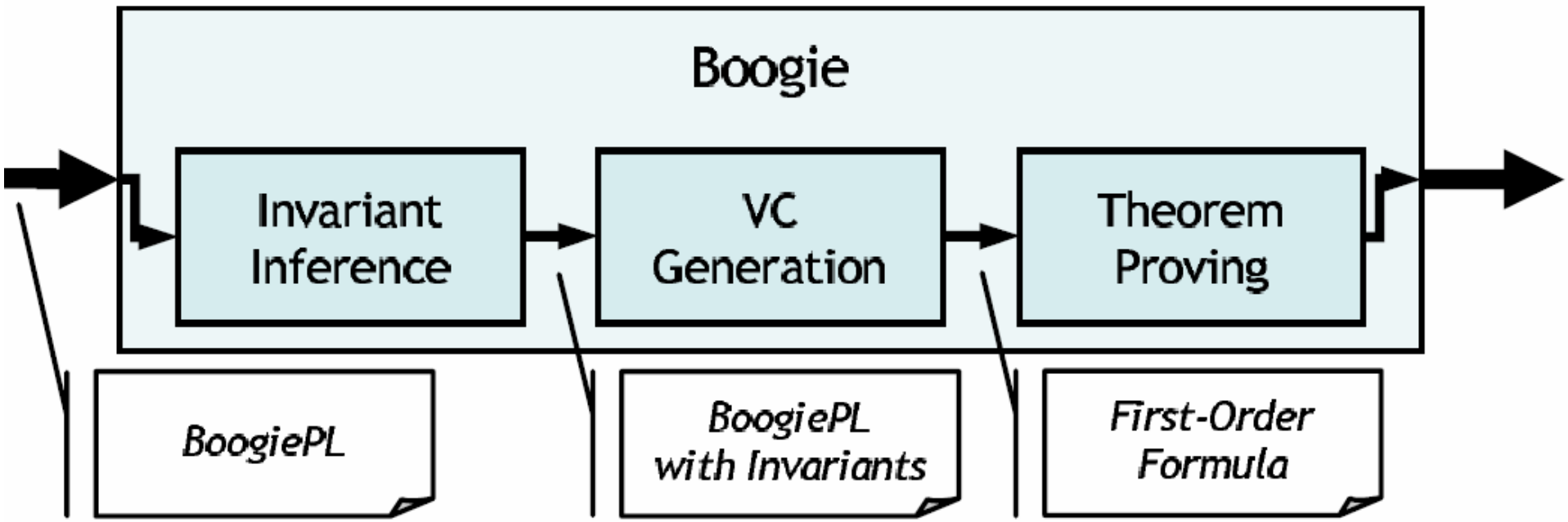
- Spec# Quelltext wird in CIL (Common Intermediate Language) durch den Spec# Compiler übersetzt

Überblick

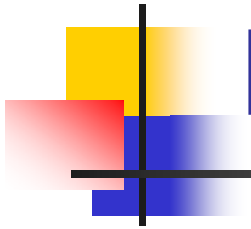


- CIL wird in BoogiePL übersetzt

Überblick

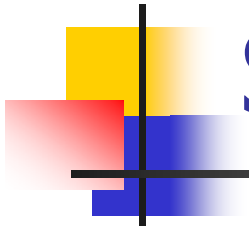


- Eigentliche Verifikation des BoogiePL Programms bzw. des ursprünglichen Programms



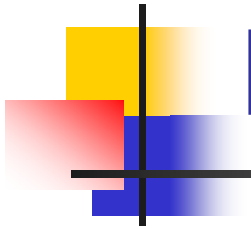
BoogiePL

- Prozedurale Sprache
- nicht komplex, daher leichter umzuwandeln
- Nur Prozeduren (keine Methoden), keine bedingten Sprünge, nur goto, veränderbare Variablen



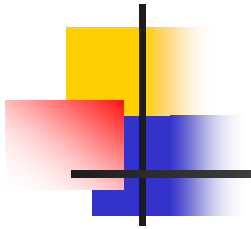
Syntax der BoogiePL

- Typen
- Konstanten
- Funktionsprototyp
- Prozeduren
- Ausdrücke



BoogiePL, Typen

- Variablenarten
  - feste mit *const* deklarierte
  - veränderbare mit *var* deklarierte
- Typen
  - bool — boolischer Typ
  - int — Ganzzahltyp
  - ref — Referenzen
  - name — Bezeichner
  - any — Obertyp von jedem Typ
  - arraytypen — [*type*] *type oder* [*type,type*] *type*
  - selbstdefinierte — *type* typname ;
- Typen können mit *where* Bedingung erhalten



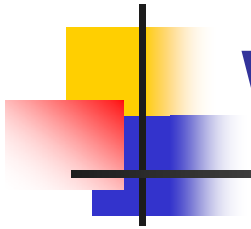
Bezeichnerbeispiel

- Spec#:

```
class Example {
    int x;
    string! s;
... }
```

- BoogiePL:

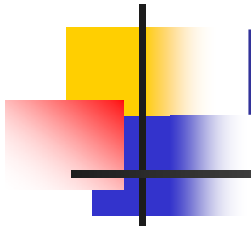
```
const Example : name;
const Example.x : name;
const Example.s : name;
...
```



wichtige Variablen\Konstanten

- Heap wird als Variable deklariert durch:
  - ***var** Heap : [**ref** , **name**]**any**;*
  - Zugriff durch z.B: [e, Example.x]

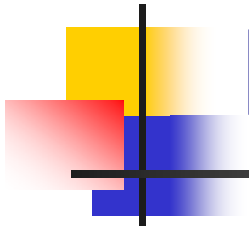
- Ein (allgemeiner) Objekttyp wird deklariert durch:
  - ***const** System.Object* : **name**;



Funktionsprototyp/Funktionen

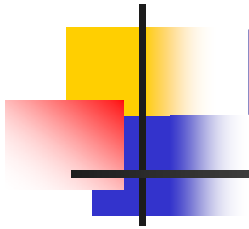
- Funktionsprototyp
  - ***function*** *function (type*)* ***returns*** *(type);*

- wichtige Hilfsfunktionen
  - ***function*** *typeof(obj :* ***ref*** *)* ***returns*** *(class :* ***name****);*
  - ***function*** *StringLength(s :* ***ref*** *)* ***returns*** *(len :* ***int****);*



Prozeduren

- Alle Methoden in Spec# werden in BoogiePL zu Prozeduren
- Prozeduren
  - ***procedure*** *procname (vardecl*)* [***returns*** *(vardecl*)*] ([***free***] ***requires*** *expr ;)*(****modifies*** *var* ;)*([****free***] ***ensures*** *expr;)* [implbody]*
- Prozedurenimplementierungen
  - ***implementation*** *procname (vardecl*)* [***returns*** *(vardecl*)*] *implbody*
- Implementierung
  - *{* Variablendeklarationen *{ label : Befehl* Sprung} }*



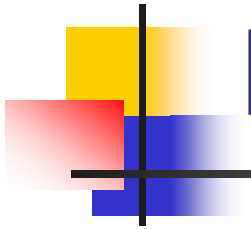
Funktionsbeispiel

- Spec#:

```
class Example {
        public int M (int n) {...}
        ...
}
```

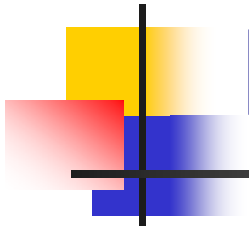
- BoogiePL:

```
...
procedure Example.M (n : int) returns(int)
{ ... }
...
```



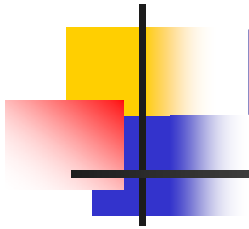
Befehle

- **Befehlsarten**
  - **Passiv**
    - ***assert*** *expr ;*
    - ***assume*** *expr ;*
  - **Zuweisungen**
    - **variablenzugriff *:=* expr*;***
    - ***havoc*** **var⁺*;***
  - **Aufrufe**
    - ***call*** **var*** *:= procname (expr*) ;*
  - **Sprünge**
    - ***goto*** *label⁺ ;*
    - ***return*** *;*



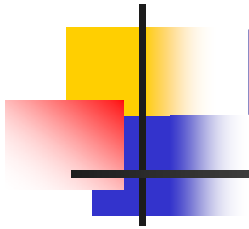
BoogiePL, Ausdrücke

- Ausdrücke:
  - **false**
  - **true**
  - **null**
  - *integer*
  - ($\forall$ *vardecl** • *expr*)
  - ($\exists$ *vardecl** • *expr*)
  - Operator auf Ausdruck angewendet



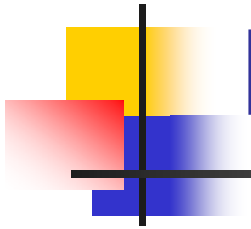
BoogiePL, Ausdrücke

- Operatoren:
  - Boolesch:
    - $\Leftrightarrow$ | $\Rightarrow$ | $\wedge$ | $\vee$ | $\neg$ | $\neq$ | = | $\leq$ | < | <:
  - Arithmetische:
    - + | − | $*$ | / | %
  - bestimmter Typ:
    - cast(expr, type)
    - old(expr) : wird bei *ensure* Klausel benutzt
    - (): Klammerung



BoogiePL, Axiome

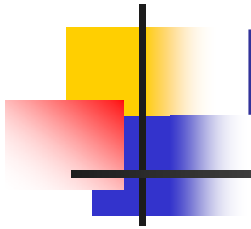
- Axiome:
  - spezifizieren Beschränkung über *Konstanten* und *Funktionen*
  - müssen Rückgabe von Typ Bool haben
  - ***axiom*** *expr* ***;***
- Beispiel:
  - axiom($\forall$ o:ref, T:name • $Is(o,T) $\Leftrightarrow$ o=null $\vee$$typeof(o)<:T)
  - axiom *Example* <: *System.Object*;



Beispiel:

- **In Spec#:** *if (P) S; else T; ...*
- **In BoogiePL:**

Start:	*goto* Then *or* Else
Then:	*assume P;*
	übersetze S in BoogiePL;
	goto After
Else:	*assume ¬P;*
	übersetze T in BoogiePL;
	goto After
After:	...



Benutzung des Boogie

- Übersetzung des CIL Codes in BoogiePL
  - durch Abstrakte Syntax Bäume (AST)
  - durch Umwandlung von
    - Heap, Speicherallokation sowie Klassenvariablen
    - Call-By-Referenz Parametern
    - Methoden und Methodenaufrufen
    - Schleifen
  - Axiomatisierung des Spec# Typsystems

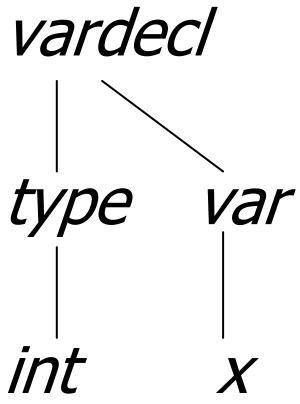
Abstrakter Syntax Baum

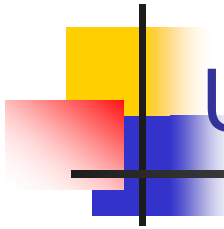
- Syntaktische Analyse
  - Ausführung der syntaktische Analyse durch Parser
  - Das Ziel ist die Struktur des Programms herauszufinden und syntaktische Fehler zu korrigieren (beim Kompilier)
  - Zwischendarstellung des Programms über strukturelle Informationen durch ein Syntaxbaum
- Beispiel: Code

  Spec #: *int x;*

  BoogiePL: *const x: name;*

AST



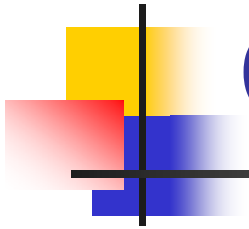


Umwandlung von Heap, Speicherallokation

- Der Heap wird als zweidimensionales Array modelliert, formal von Typ *[ref,name]any*
- Die Speicherallokation wird durch ein Konstante *allocated* vom Typ name modelliert
- Klassenvariablen werden auf dem Heap abgelegt, z.B.:

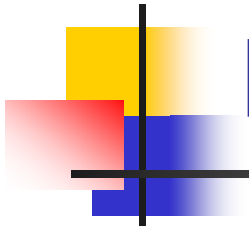
  o.f wird übersetzt *Heap[o,C_f]*,

  wobei *C_f* Konstante von *f* ist



Call-By-Referenz Parametern

- In BoogiePL:
  - arbeiten Methoden mit Ein- und Ausgabeparametern
  - gibt es **kein** Call-By-Referenz Prinzip, aber wird durch Call-By-Value modelliert
    - **Vorgehensweise**:
      - Für jede Referenz wird ein Ein und Ausgabeparameter angelegt
      - Variable der Referenz wird in Eingabeparameter beim Prozedureintritt kopiert
      - Nach der Prozedur wird der Ausgabeparameter wieder in die Variable der Referenz zurückkopiert



Methoden und Methodenaufrufen

- Für jede (überschriebene) Methode wird eine Implementierung in BoogiePL erzeugt:
  - **1.Fall:** Man weiss, welche Methode aufgerufen wird
  - **2.Fall:** Man weiss es nicht, welche Methode aufgerufen wird aufgrund der Polymorphie
    - Für abstrakte Methoden wird eine zusätzliche BoogiePL Prozedur erstellt und dann aufgerufen

Umwandlung von Schleifen

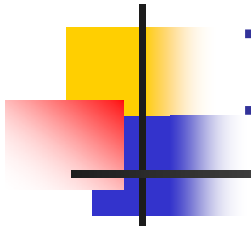
- **In Spec#:** *while(B) { S; }*
- **In BoogiePL:**

LoopHead:	*goto* LoopBody *or* LoopAfter
LoopBody:	*assume B;*
	übersetze S in BoogiePL;
	goto LoopHead
AfterLoop:	*assume ¬B;*
	...

Axiomatisierung des Spec# Typsystems

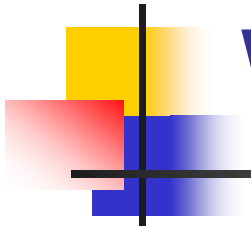
- Ziel:
  - Charakterisieren der Typeigenschaften
  - z.B.: Wertebereich des Datentyps

- Beispiel: (Nicht-Null-Typ)
  - *axiom* ($\forall$ o:ref, T:name • $Is(o, $NotNull(T)) $\Leftrightarrow$ o≠null $\wedge$ $Is(o,T))
  - Semantik:
    - *$Is(o,T)* gibt an, ob Referenz auf Typ T zeigt
    - *$NotNull(T)* gibt den Nicht-Null-Typ T! zurück



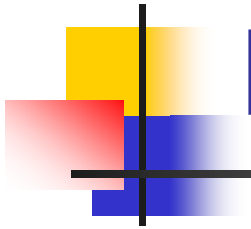
Invariantenfolgerung

- Schleifeninvarianten werden hinzugefügt, durch die Folgerungen mit der weakest Precondition
- Weakest Precondition P=wp(S,Q), heißt das von der Nachbedingung Q, die schwächsten Precondition P gesucht wird



Weakest Precondition

- Regel für die weakest Precondition
  - wp(assert P,Q) $= P \wedge Q$
  - wp(assume P,Q) $= P \Rightarrow Q$
  - wp(S;T;Q) = wp(S,wp(T,Q))



Prüfbedingungserzeugung

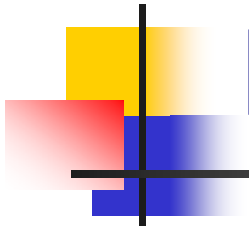
- Umwandlung von Schleifen, so daß, der Kontrollfluß azyklische wird
  - Rückwartskanten im Kontrollflußgraphen löschen, also die entsprechenden goto‘s in BoogiePL
  - havoc x mit Schleifenvariable x
- DSA (dynamic singel assignment)
  - Immer frische (neue) Variablen
- BoogiePL Programm wird in passives Programm umgewandelt
  - var = expr; durch assert var==expr; ersetzt

Umwandlung von Schleifen & DSA

- Azyklischen Kontrolfluss, durch entfernen der Rücksprünge bzw Rückwartskanten im Kontrollflußgraph
- **Beispiel:** wobei I Schleifeninvariante und S Schleifenkörper sind:
  - Vorher:

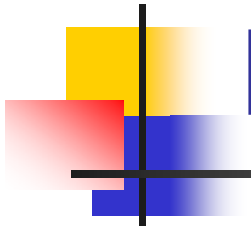
    LoopHead : **assert** I; S; **goto** *LoopHead*;
  - Nachher:

    $x^0_1 := x_1$; . . . $x^0_n := x_n$; **assert** I;
    havoc $x_1, \ldots, x_n$; **assume** I;
    S ; **assert** I ; **assume false**;



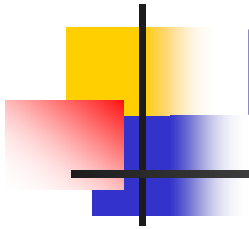
Beispiel(1)

```
static int Abs(int x)
   ensures 0 <= x ==> result == x;
   ensures x < 0 ==> result == -x;
{
if (x < 0) x = -x;
   return x;
}
```



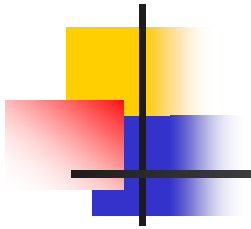
Beispiel(2)

```
procedure Abs(x$in: int) returns ($result: int);
    ensures 0 <= x$in ==> $result == x$in;
    ensures x$in < 0 ==> $result == -x$in;
{ var x       : int, b: bool;
  entry:      x   := x$in; b := x < 0;   goto t, f;
  t:          assume b; x := -x;         goto end;
  f:          assume !b;                 goto end;
  end:        $result := x;              return; }
```



Beispiel(3)

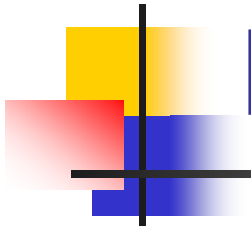
```
procedure Abs(x$in: int) returns ($result: int);
    ensures 0 <= x$in ==> $result == x$in;
    ensures x$in < 0 ==> $result == -x$in;
{ var x1, x2: int, b: bool;
  entry:    x1 := x$in; b := x1 < 0;      goto t, f;
  t:        assume b; x2 := -x1;          goto end;
  f:        assume !b; x2 := x1;          goto end;
  end:      $result := x2;                return; }
```



Beispiel(4)

```
procedure Abs(x$in: int) returns ($result: int);
{ var x1, x2: int, b: bool;

 entry:  assume x1 == x$in;
         assume b == x1 < 0;                    goto t, f;
 t:      assume b; assume x2 == -x1;            goto end;
 f:      assume !b; assume x2 == x1;            goto end;
 end:    assume $result == x2;
         assert 0 <= x$in ==> $result == x$in;
         assert x$in < 0 ==> $result == -x$in;
                                                return; }
```



Beispiel(5)

```
entry &&
  (entry <== (x1 == x$in ==>
                    b == x1 < 0 ==>                  t && f)) &&
  (t <==            (b ==> x2 == -x1 ==>             end)) &&
  (f <==            (!b ==> x2 == x1 ==>             end)) &&
  (end <==      ($result == x2 ==>
                     (0 <= x$in ==> $result == x$in) &&
                     (x$in < 0 ==> $result == -x$in) &&
                                                     true))
```