Three-valued logic for software specification.

An ERASMUS course

University Gent

Eötvös-Loràn University Budapest

University Patras

A. Hoogewijs

October 1999

... everywhere in mathematics, the explicit formulation marks only the end of a sometimes long period of non-formulated operativeness, and in such a period the student must read in the teachers' face what his mind knows only semiconsciously.

[Freudental 1973, p20]

Contents

1 Introduction

Predicates play an important role in programming and in the formal specification of software systems. To clarify this statement, it suffices to consider **predicates** as functions which assume **BOOLEAN** values = {T,F}.

- they appear as *guards* for conditional expressions,

  `while` $x \neq y$ `do`

  `if` $x > y$ `then` $x := x - y$

  `else` $y := y - x$;

- the are used to impose *constraints on domains*,

  $-32768 \leq integer \leq 32767$

- they allow to express properties of software,

  `Getdate(var year, month, day, dayofweek:word)`

  `then` $1980 \leq$ `year` ≤ 2099, $1 \leq$ `month` ≤ 12, $1 \leq$ `day` ≤ 31, $0 \leq$ `dayofweek` ≤ 6

- they can be used in axiomatic specifications of software.

  Example:

  "**invariants**" i.e. *predicates P_i that are true before and after each iteration of a loop*

  e.g. in the extended Euclides algorithm we look for (u_1, u_2, u_3) s.t.

  $u * u_1 + v * u_2 = u_3 = gcd(u, v)$

  We use (v_1, v_2, v_3) and (t_1, t_2, t_3) that satisfy

  $$\begin{cases} u * t_1 & + & v * t_2 & = & t_3 \\ u * u_1 & + & v * u_2 & = & u_3 \\ u * v_1 & + & v * v_2 & = & v_3 \end{cases}$$

  The algorithm ends with $v_3 = 0$

- PROLOG: Consider the programming statement

```
devide(A,B,Q,R) if
   A>B       and
   Q=A div B and
   R=A-Q*B .
```

From a classical point of view, *predicates* are considered as **total functions** with two possible values $\{T, F\}$. We also say that *sentences* build out of predicates are either **true** or **false**. Note however that it is not difficult to find *predicates* that don't satisfy this description: e.g.

- *books smell yellow*,
- *the king of the America is black*,
- $\sqrt{-1} < 2$
- for $a : \texttt{array}[1..10]$ `of integer` $a[10] > a[11]$.

It is clear that those expressions are senseless, the one more trivially than the other.

Two-valued logic is the basis for the theory of mathematical proofs. The latter is a sequence of formulas, which are **valid** within their interpretation domains. Senseless expressions are not allowed in proofs, and thus excluded from the theory. In the construction of a proof however, potentially meaningless expressions may occur, such as

$\sqrt{x} < 2$, $a[i] > a[i+1]$

Those problems are solved however, by *strengthening the assumptions.* e.g.

$x \geq 0$, $i \leq 9$

This formalism however is not expressible in classical proof theory. It is part of the folklore of Mathematics, and belongs to the *non-formulated operativeness.*

It is clear that we cannot go on with this game, if we use *predicates in the description of algorithms.* In these applications, predicates do not describe general statements, but consist of tests which are *dynamically executed*, to select an appropriate branch of the algorithm, next to be executed.

The predicates being partial, can have different causes:

- the physical limitation of the compiler (machine);

  $x : \texttt{integer} \longrightarrow x < \texttt{maxint}$

- the use of mappings (arrays) as data types

  $a[i] > a[i+1]$

- the evaluation of functions may lead to non-terminating computations, or stack overflow

  $\texttt{fac}(x) > x^2$

  where $\texttt{fac}(x) \equiv$ `if` $x = 0$ `then` 1 `else` $x * \texttt{fac}(x-1)$ `endif`

Partial predicates cannot be avoided in the description of algorithms. Furthermore it is clear that any attempt to evaluate a predicate out of its definition domain leads to an error situation. In order to avoid this situations, we must be able to analyze and describe them. Hence we need a formalism to deal with partial predicates.

This can be done in several different ways

1. In the classical two-valued logic.

   It suffices to introduce the notion of *domain of a predicate* for which the predicate is defined, and consider a test on the arguments of the predicate to see if it will be defined.

   `if` $x \in$ `def`(P) `then` $P(x)$

2. In a three-valued logic.

   All predicates are **total** again, but take values within **EXTBOOLEAN**$= \{T, F, U\}$.

It is clear that we prefer the latter method, certainly when we have to deal with algorithms and implementations of algorithms. The most important argument for this choice is the fact that in practice it is not always simple to *determine the domain of a predicate.* Moreover, in a lot of cases, the domain of a predicate depends on the implementation (the programming language) or the machine (memory available).

2 Three-valued logic in TURBO PASCAL

For our examples, we rely on **BORLAND TURBO PASCAL Version 5.0**. In this version there is a compiler option {$B+} resp {$B–} for the so-called *Boolean Evaluation.* Using this options, Boolean Evaluation switches between the two different models of code generation for the `AND` and `OR` Boolean operators.

In the {$B+} state, the compiler generates code for complete Boolean expression evaluations. This means that every operant of the Boolean expression, built from the `AND` and `OR` operators, is guaranteed to be evaluated, even when the result of the entire expression is already known.

In the {$B–} state, the compiler generates code for the so-called short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident.

The Default setting for Turbo 5 is {$B–}, whereas for the older versions up to Version 3.0, there was no such an option and the Boolean Evaluation was done corresponding to the {$B+}-setting.

We will explain what is going on in each of those cases, with the help of some simple examples.

```
{$B-}
program deftest1;
var a: integer;
begin
 for a:=5 downto -5 do
     if true or (1/a>0) then writeln(a);
end.
```

```
With {$B-} we get:           With {$B+} we obtain
    5                          5
    4                          4
    3                          3
    2                          2
    1                          1
    0                         Runtime error 200 at ..address..
   -1                         Division by zero
   -2
   -3
   -4
   -5
```

```
{$B-}
program deftest2;
var a: integer;
begin
 for a:=5 downto -5 do
     if (1/a>0) or true then writeln(a);
end.
```

```
With {$B-} we get:           With {$B+} we obtain
    5                          5
    4                          4
    3                          3
    2                          2
    1                          1
Runtime error 200 at ...      Runtime error 200 at ...
Division by zero              Division by zero
```

```
{$B-}
program deftest3;
var
 a: integer;
begin
 for a:=5 downto -5 do
   if (a<>0) and (abs(1/a)>0.3 ) then writeln(a:4,' : ',1/a:8:5)
                                 else writeln(a:4,' : ');
end.
```

```
With {$B-} we get:       With {$B+} we get
   5 :                      5 :
   4 :                      4 :
   3 :  0.33333             3 :  0.33333
   2 :  0.50000             2 :  0.50000
   1 :  1.00000             1 :  1.00000
   0 :                     Runtime error 200 at ...
  -1 : -1.00000            Division by zero
  -2 : -5.00000
  -3 : -0.33333
  -4 :
  -5 :
```

The evaluation of the *guards* corresponds to the following definition tables for `OR` and `AND`.

{B+}

OR	T	F	U
T	T	T	U
F	T	F	U
U	U	U	U

AND	T	F	U
T	T	F	U
F	F	F	U
U	U	U	U

{B–}

OR	T	F	U
T	T	T	T
F	T	F	U
U	U	U	U

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	U	U

in both cases we have

a	NOTa
T	F
F	T
U	U

In the {$B–}-case we note that OR and AND are not commutative:

$$a \ \texttt{OR}\ b \ \neq \ b \ \texttt{OR}\ a$$

$$a \ \texttt{AND}\ b \ \neq \ b \ \texttt{AND}\ a$$

The simplest way to describe those operators, is the use of the **McCarthy** operator:

$$[\ ,\ ,\ ] : \texttt{EXBOOLEAN} \times \texttt{EXBOOLEAN} \times \texttt{EXBOOLEAN} \longmapsto \texttt{EXBOOLEAN}$$

where

$$[a, b, c] \equiv \texttt{if}\ \ a = \texttt{T}\ \texttt{then}\ b\ \texttt{else}$$
$$\texttt{if}\ \ a = \texttt{F}\ \texttt{then}\ c\ \texttt{else}\ \ \texttt{U}$$

Then we may define

$$\begin{array}{lcl} \texttt{NOT}\ a & \equiv & [a, \texttt{F}, \texttt{T}] \\ a\ \texttt{OR}\ b & \equiv & [a, \texttt{T}, b] \\ a\ \texttt{AND}\ b & \equiv & [a, b, \texttt{F}] \\ a \Longrightarrow b & \equiv & [a, b, \texttt{T}] \end{array}$$

From those definitions, we can deduce a number of properties:

associativity:

$$\begin{array}{lcl} a\ \texttt{OR}\ (b\ \texttt{OR}\ c) & = & (a\ \texttt{OR}\ b)\ \texttt{OR}\ c \\ a\ \texttt{AND}\ (b\ \texttt{AND}\ c) & = & (a\ \texttt{AND}\ b)\ \texttt{AND}\ c \end{array}$$

distributivity:

$$\begin{array}{lcl} a\ \texttt{AND}\ (b\ \texttt{OR}\ c) & = & (a\ \texttt{AND}\ b)\ \texttt{OR}\ (a\ \texttt{AND}\ c) \\ a\ \texttt{OR}\ (b\ \texttt{AND}\ c) & = & (a\ \texttt{OR}\ b)\ \texttt{AND}\ (b\ \texttt{OR}\ c) \end{array}$$

de MORGAN

$$\begin{array}{lcl} \texttt{NOT}(a\ \texttt{AND}\ b) & = & \texttt{NOT}a\ \texttt{OR}\ \texttt{NOT}b \\ \texttt{NOT}(a\ \texttt{OR}\ b) & = & \texttt{NOT}a\ \texttt{AND}\ \texttt{NOT}b \end{array}$$

But!

$$
\begin{array}{lcl}
a \ \texttt{OR} \ \ \texttt{NOT}a & = & \texttt{T} \Longleftrightarrow \texttt{def}a \\
a \ \texttt{AND} \ \ \texttt{NOT}a & = & \texttt{F} \Longleftrightarrow \texttt{def}a \\
a \ \texttt{OR} \ \texttt{T} & = & \texttt{T} \Longleftrightarrow \texttt{def}a
\end{array}
$$

From **deftest3** we can see that this calculus has some advantages in expressing *guards* of conditional expressions. The way of evaluating those guards in this way, is sometimes called **Lazy evaluation**.

However, the fact that `AND` and `OR` are not commutative makes this calculus inconvenient as a basis for the non-definedness notion in software specification.

3 The notion of *deduction calculus*

In order to avoid frightening off my audience I prefer to explain the notion of "deduction calculus" for the classical two-valued case, instead of the three-valued one.

For a thorough treatment of such a calculus, we refer to Hermes, *Introduction to Mathematical Logic* (Springer Verlag 1972).

3.1 Proposition calculus

Everybody knows the so-called truth tables for the classical connectives:

$\neg$	
T	F
F	T

$\wedge$	T	F
T	T	F
F	F	F

$\vee$	T	F
T	T	T
F	T	F

$\rightarrow$	T	F
T	T	F
F	T	T

$\leftrightarrow$	T	F
T	T	F
F	F	T

$\uparrow$	T	F
T	F	T
F	T	T

"$\uparrow$" is the so-called **sheffer stroke** or `NAND` operator:

$a \uparrow b = \neg(a \wedge b)$

Definitions:

$n-$ ary propositions: $P_n = \{p : \texttt{BOOLEAN}^n \mapsto \texttt{BOOLEAN}\}$

propositions: $P = \bigcup_{n=1}^{\infty} P_n$

composition: let $p \in P_k$, for $i := 1...k, q_{n_i} \in P_{n_i}$, then $p(q_{n_1}...q_{n_k}) \in P_{\sum n_i}$

Problem:

Find a minimal set of functions in P which generates P completely, using composition.

Answer: $\{\neg, \wedge\}$, $\{\uparrow\}$

As a computer scientist, we are not interested in building all possible propositions. Instead, we want to construct logical expressions from basic expressions, using the logical connectives and we also want some rules to derive valid expressions from other valid expressions.

A basis for this, is the

Formula-calculus for propositional logic

`atomic` := `identifier`

`formula` := `atomic` | $\neg$`formula` | (`formula` $\wedge$ `formula`)

This is an example of the description of a language, in the so-called Backus Naur form (BN-form). It allows us to generate **PROPPARSER** which is able to verify if a given expression is a *formula* from our proposition calculus.

PROP $= \{a : a \textit{ is a formula}\}$

To express relations between formulas, we introduce the following notations en definitions:

Let L denote a *finite list of formulas*,
Let a, b, c denote *arbitrary formulas*,
$L \models a$ means a is a **consequence** of L
$L \vdash a$ means a is **deducible** from L

$\models$ is a model-theoretical relation, this means that we need the notion of **interpretation of a formula**.

An **interpretation** I is a mapping **PROP** $\mapsto$ **BOOLEAN**, defined from $I_{\texttt{iden}}$ i.e. the restriction of I to the set of identifiers, and the tables for $\neg$ and $\wedge$.

We say that an interpretation I is a **model** for a list L of formulas iff

$I(p) = T$ for each proposition p of L.

Then we denote

$L \models a$ `iff` $I(a) = \texttt{T}$ *for each model* I *for* L.

Property

Since L is a finite list of formulas, each of them built from a finite number of identifiers, it is easy to show that the relation $L \models a$ is decidable.

We note that since $\{\neg, \wedge\}$ is functionally complete, it is not a restriction to consider only $\neg$ and $\wedge$ in the construction of our formulas. The other connectives will be defined in function of $\neg$ and $\wedge$.

With respect to theorems involving the structure of our calculus it is an advantage to have only two building connectives $\neg$ and $\wedge$.

We noticed already that in most cases we are not interested in the truth value of an expression, but in the relation $L \models a$. So we want to formalize this relation, in order to be able to deal with it, without having to refer to the interpretation.

So we introduce a new symbol $\vdash$ and read $L \vdash a$ as "*a is deducible from L*". This relation can be axiomatically defined be means of the following rules:

(A) $\quad a \vdash a \qquad$ assumption rule

(C) $$\frac{\begin{array}{lll} L_1 & \vdash & a \\ L_2 & \vdash & b \end{array}}{\begin{array}{lll} L_{12} & \vdash & a \wedge b \end{array}}$$ introduction of the conjunction

(C_i) $$\frac{\begin{array}{lll} L & \vdash & a_1 \wedge a_2 \end{array}}{\begin{array}{lll} L & \vdash & a_i \end{array}}$$ elimination of conjunction

$$\text{(R)} \quad \begin{array}{lrcl} L_1 & a & \vdash & b \\ L_2 & \neg a & \vdash & b \\ \hline L_{12} & & \vdash & b \end{array} \qquad \text{removal rule}$$

$$\text{(X)} \quad \begin{array}{lcl} L_1 & \vdash & a \\ L_2 & \vdash & \neg a \\ \hline L_{12} & \vdash & b \end{array} \qquad \text{contradiction rule}$$

note that L_{12} is the notation for $L_1 \cup L_2$

A **deduction** is a sequence of expressions of the form

$L_1 \vdash a_1, ..., L_n \vdash a_n$

where each of the $L_i \vdash a_i$ is either an application of the rule (A) or follows from a preceding $L_j \vdash a_j$ applying (C_i) or follows from preceding $L_j \vdash a_j$ and $L_k \vdash a_k$ applying one of the rules (C), (R), (X). We say that $L_n \vdash a_n$ is **derivable**. We also say the a_n is deducible from L_n.

As an example we consider :

$$\text{(SeAt)} \quad \begin{array}{lcl} L\ \neg a & \vdash & a \\ \hline L & \vdash & a \end{array} \text{(Self Assertion)} \qquad \begin{array}{lrcll} (1) & L, & \neg a & \vdash & a & \\ (2) & & a & \vdash & a & (A) \\ (3) & L & & \vdash & a & (R)(2,1) \end{array}$$

$$\text{(SeDe)} \quad \begin{array}{lcl} L,\ a & \vdash & \neg a \\ \hline L & \vdash & \neg a \end{array} \text{(Self Denial)} \qquad \begin{array}{lrcll} (1) & L & a & \vdash & \neg a & \\ (2) & & \neg a & \vdash & \neg a & (A) \\ (3) & L & & \vdash & \neg a & (R)(1,2) \end{array}$$

Note that $\vdash$ is defined algorithmically without talking about truth values.

Important Theorem

The described calculus is sound and complete with respect to $\models$ i.e.

sound: *if* $L \vdash a$ *then* $L \models a$
complete: *if* $L \models a$ *then* $L \vdash a$

Corollary

$L \vdash a$ *is decidable.*

Remarks

* *The relations $\vdash$ and $\models$ should not be confused with the logical connective $\rightarrow$ (implication).*

  $a \models b$ means that b is true whenever a is true (model theoretic)

  $a \vdash b$ just means that $a \vdash b$ is derivable, or that b is deducible from a.

  $a \rightarrow b$ is an abbreviation for $\neg(a \wedge \neg b) = \neg a \vee b$, and hence just a formula of `PROP`.

* We can show:

  $a \vdash b$ `iff` $\vdash a \rightarrow b$

In order to become more familiar with the concept of a *deduction* in a *deduction calculus*, we consider some derivations of simple rules.

(CCo) $(a \wedge b) \vdash (b \wedge a)$

(1)	$(a \wedge b)$	$\vdash$	$(a \wedge b)$	(A)
(2)	$(a \wedge b)$	$\vdash$	a	$(C_1)(1)$
(3)	$(a \wedge b)$	$\vdash$	b	$(C_2)(1)$
(4)	$(a \wedge b)$	$\vdash$	$(b \wedge a)$	$(C)(3,2)$

(CAs) $((a \wedge b) \wedge c) \vdash (a \wedge (b \wedge c))$ *similar*

(CAs′) $(a \wedge (b \wedge c)) \vdash ((a \wedge b) \wedge c)$ *similar*

It follows that

$$\frac{a, b, c \;\vdash\; d}{c, b, a \;\vdash\; d}$$

In order to be able to show this, we need more derived rules:

Cut Rule:

(CuRu)
$$\frac{\begin{array}{lcl} L_1 & \vdash & a \\ L_2,\ a & \vdash & b \end{array}}{\begin{array}{lcl} L_{12} & \vdash & b \end{array}}$$

(1)	L_1		$\vdash$	a	
(2)	L_2	a	$\vdash$	b	
(3)		$\neg a$	$\vdash$	$\neg a$	(A)
(4)	L_1	$\neg a$	$\vdash$	b	$(X)(1,2)$
(5)	L_{12}		$\vdash$	b	$(R)(2,4)$

Extension of the antecedent:

(ExtAn)
$$\frac{\begin{array}{lcl} L & \vdash & a \end{array}}{\begin{array}{lcl} L,\ b & \vdash & a \end{array}}$$

(1)	L		$\vdash$	a	
(2)		b	$\vdash$	b	(A)
(3)	L	b	$\vdash$	$a \wedge b$	$(C)(1,2)$
(4)	L	b	$\vdash$	a	$(C_1)(3)$

Decomposition of the antecedent:

(AnDc) $$\frac{L,\ (a \wedge b) \ \vdash\ c}{L,\ a,\ b \ \vdash\ c}$$

(1)	L	$(a \wedge b)$	$\vdash$	c	
(2)		a	$\vdash$	a	(A)
(3)		b	$\vdash$	b	(A)
(4)		a, b	$\vdash$	$a \wedge b$	$(C)(2,3)$
(5)	L	a, b	$\vdash$	c	$(CuRu)(4,1)$

Unification of the antecedent

(AnU) $$\frac{L,\ a,\ b \ \vdash\ c}{L,\ (a \wedge b) \ \vdash\ c}$$ Similar

Finally we consider the derivation of the **deductionrules**:

(DdRu$_1$) $$\frac{a \ \vdash\ b}{\vdash\ \neg(a \wedge \neg b)}$$

(1)	a	$\vdash$	b		
(2)	$a, \neg b$	$\vdash$	b	$(ExtAn)$	(1)
(3)	$a \wedge \neg b$	$\vdash$	b	(AnU)	(1)
(4)	$a \wedge \neg b$	$\vdash$	$a \wedge \neg b$	(A)	
(5)	$a \wedge \neg b$	$\vdash$	$\neg b$	(C_2)	(4)
(6)	$a \wedge \neg b$	$\vdash$	$\neg(a \wedge \neg b)$	(X)	$(3,5)$
(7)		$\vdash$	$\neg(a \wedge \neg b)$	$(SeDe)$	(6)

(DdRu$_2$) $$\frac{\vdash\ \neg(a \wedge \neg b)}{a \ \vdash\ b}$$

(1)		$\vdash$	$\neg(a \wedge \neg b)$		
(2)	$a \wedge \neg b$	$\vdash$	$a \wedge \neg b$	(A)	
(3)	$a \wedge \neg b$	$\vdash$	b	(X)	$(2,1)$
(4)	$a, \neg b$	$\vdash$	b	$(AnDc)$	(3)
(5)	a	$\vdash$	b	$(SeAt)$	(4)

Note that (DdRu$_1$) and (DdRu$_2$) can be written as:

(DdRu$_1$) $$\frac{a \ \vdash\ b}{\vdash\ a \rightarrow b}$$ (DdRu$_2$) $$\frac{\vdash\ a \rightarrow b}{a \ \vdash\ b}$$

Corollary:

$a \vdash b$ `iff` $\vdash a \rightarrow b$

Note that this corollary, together with the sound- and completeness theorem, justifies the confusion between $a \models b$, $a \vdash b$ and $\vdash a \rightarrow b$

3.2 2-valued Predicate Calculus

3.2.1

More precisely we want to deal with **many-sorted first order predicate calculus, with equality**.

Since we know already about "Propositional Calculus", it can be considered as an extension.

We note that we need to pay much more attention to the **language** of the predicate calculus. This languages will depend on the "theory" we want to formalize.

3.2.2 Basic notation

Given a set S, $|S|$ denote its cardinality. $S^+ = \bigcup_{n=1}^{\infty} S_n$ is the set of non-empty finite strings of elements of S. If $w \in S_n$, w is a string of length $|w| = n$. The empty string is denoted ϕ and $S^* = S^+ \cup \{\phi\}$. $|\phi| = 0$.

Given a **sort** set S, an S-**sorted set** is a family $A = \{A_s \mid s \in S\}$ of sets. Let A and B be S-sorted sets, $A \subseteq B$ means $A_s \subseteq B_s$ for all $s \in S$. An S-sorted arrow $f : A \mapsto B$ is an S-sorted set $\{f_s : A_s \mapsto B_s\}$ of arrows between the components; composition if such arrows is defined component-wise. For $s_1 ... s_n \in S^+$, $A^{s_1 \cdots s_n}$ denotes the cartesian product $A_{s_1} \times ... \times A_{s_n}$

3.2.3 Signature of a theory

Definition Given a sort set S, an S-sorted **operational signature** Ω is a S-sorted set $\{\Omega_{w,s} \mid \ < w, s > \in S^* \times S\}$. For the empty word ϕ, an element $c \in \Omega_{\phi,s}$ is called a *constant_identifier* of sort s. For each non-empty word $w \in S_n$, an element $f_{w,s}$ of $\Omega_{w,s}$ is called *operation_identifier* or a *function_identifier* of *domain type* w, *codomain type* s and *arity* n. $< w, s >$ is also called the *rank* of the operation.

Definition Given a sort set S, an S-sorted **relational signature** Π is a S^*-sorted set $\{\Pi_w \,|\, w \in S^*\}$. For the empty word ϕ, an element $P \in \Pi_\phi$ is called a *proposional_identifier*. For each non-empty word $w \in S_n$, an element P_w of Π_w is called a *predicate_identifier* or a *relational_identifier* of *domain type* w and *arity* n.

Definition An S-sorted *signature of a theory* is a tuple $\Sigma =< \Omega, \Pi >$ where Ω is an S-sorted operational signature and Π is an S-sorted relational signature.

We note that either Ω or Π may be empty, depending on the "theory" which has to be studied.

Examples A classical example considers the *theory of natural numbers*. We define a one-sorted signature as follows. Let S be the set of sorts is $S = \{nat\}$.

The S-sorted operational signature Ω consists of:

$$\begin{array}{ll} \Omega_{\phi,nat} & = \{0\} \\ \Omega_{nat,nat} & = \{succ\} \\ \Omega_{nat\,nat,nat} & = \{add, mult\} \end{array}$$

The S-sorted relational signature Π consists of:

$$\Pi_{natnat} \quad = \{lt\}$$

Then $\Sigma =< \Omega, \Pi >$ is the signature for the *theory of ordered natural numbers.*

In the context of theoretical computer science, one may extend this example to a two-sorted example. Consider the set of sorts is $S = \{nat,\ bool\}$, and extend Ω with

$$\Omega_{bool\,nat\,nat,nat} \quad = \{case\}$$

Note that *case* represents the following function:

$$case : \mathbf{B} \times \mathbf{N} \times \mathbf{N} \mapsto \mathbf{N} \qquad case(x, y, z) = \begin{cases} y, & \texttt{if } x = \texttt{tt}; \\ z, & \texttt{otherwise} \end{cases}$$

3.2.4 The language of Predicate calculus

The alphabet:

- consists in the first place of the *logical symbols*:

  "$\neg$", "$\wedge$", "(", ")", "$\forall$", "$=$"

  We recall that restricting the logical symbols to this set, has to do with the structural theorems of predicate logic.

- consists in the second place of the *signature* of the theory,

  $\Sigma =< \Omega, \Pi >$

  we will call Σ the *signature* of the language.

- consists in the third place of an S-sorted set $X = \{X_s \mid s \in S\}$ of *variable_identifiers* of *sort s*

Terms

$$\begin{array}{ll} term := & constant\_identifier \mid \\ & variable\_identifier \mid \\ & 1\_ary\_function\_identifier\ term \mid \\ & 2\_ary\_function\_identifier\ term\ term \mid \\ & \ldots \\ & n\_ary\_function\_identifier\ \underbrace{\text{term} \ldots \text{term}}_{n} \end{array}$$

Note that each *term* has a certain *sort*, which is the *codomain-type* of the corresponding function_identifier. It is assumed that sorts of the "argument"-terms correspond to the domain-type of the function_identifier.

Formulas

atomic := *propositional_identifier* |
1_ary_predicate_identifier term |
2_ary_predicate_identifier term term |
...
n_ary_predicate_identifier $\underbrace{\text{term ... term}}_{n}$ |
term = term (terms of the same sort)

formula := *atomic* |
"$\neg$"*formula* |
"("*formula* "$\wedge$" *formula*")" |
"$\forall$"*variable_identifier formula*

Remarks:

1. Note that the "(" and ")" arround a formula $(\alpha \wedge \beta)$ are needed to make a distinction in the construction of $((\alpha \wedge \beta) \wedge \gamma)$ and $(\alpha \wedge (\beta \wedge \gamma))$
2. Note that
   - *terms* are built out of *constant_identifiers, variable_identifiers* and *function_identifiers* only
   - *atomics* are built out of *predicate_identifiers* "acting" on those terms.
   - the "quantifier" $\forall$ only acts on *variable_identifiers* and not on *function_identifiers* or *predicate_identifiers*.

This means that we are dealing with **first order predicate calculus**.

Again it is possible to construct a parser **PREDPARSER**, which is able to verify if a given expression is a formula of the given language **PRED**. We refer to **OBJ** where a theory may be specified in the following way:

```
th NAT+Case is
   sort Nat Bool .
   op 0 : -> Nat .
   op succ_ : Nat -> Nat .
   op add__ : Nat Nat -> Nat .
   op mult__: Nat Nat -> Nat .
   op case___: Bool Nat Nat -> Nat
endth
```

We note that in **OBJ**, a *signature* only consists of an *S-sorted operational signature.* However there is no problem to represent the relation *lt*, since it can be considered in the folowing way:

```
op lt__ : Nat Nat -> Bool .
```

We introduced the *S-sorted relational signature* to cope with the **PROLOG**-style of handling logical formulas.

The description of the relations $L \models a$ and $L \vdash a$ for a list L of formulas and a formula a is more complex now.

3.2.5 The consequence relation: $\models$

Domain for the language

Before we are able to introduce the notion of *interpretation*, we need to fix a "domain" for the language.

If $\Sigma =< \Omega, \Pi >$ is the signature of the language **PRED**, a **domain** for **PRED**, is a so-called **Σ-algebra** $\mathcal{D}$. It consist of an S-sorted set D, plus an **interpretation** $\mathcal{I}$ of Σ in D, which is:

- a family of arrows

  $i_{s_1 \ldots s_n, s} : \Omega_{s_1 \ldots s_n, s} \mapsto [D^{s_1 \ldots s_n} \mapsto D_s]$ for each *rank* $< s_1 \ldots s_n, s >$ in Ω.

  which interpret the operation symbols $f_{s_1 \ldots s_n, s}$ in Σ as actual operations $F_{s_1 \ldots s_n, s}$ on D.

- a family of arrows

  $i_{s_1 \ldots s_n} : \Pi_{s_1 \ldots s_n} \mapsto [\mathcal{P}(D^{s_1 \ldots s_n})]$ for each *domain-type* $s_1 \ldots s_n$ in Π.

  which interpret the relation symbols $P_{s_1 \ldots s_n}$ in Σ as actual relations $R_{s_1 \ldots s_n}$ on $D^{s_1 \ldots s_n}$.

  Note that if $\Pi_\phi \neq \{\ \}$ for each $P \in \Pi_\phi$, $\mathcal{I}(P) \in \{\mathbf{T}, \mathbf{F}\}$

- a family of arrows

  i_s : *variable_identifier of sort* $s \mapsto D_s$

Interpretation of terms

The interpretation defined above, can be extended to an *interpretation of terms.* If we preserve the notation $\mathcal{I}$, we get

$\mathcal{I}(f_{s_1...s_n,s}t_{s_1}...t_{s_n}) = F_{s_1...s_n,s}(\mathcal{I}t_{s_1},...,\mathcal{I}t_{s_n})$

Valuation of a formula

We define the valuation of a formula $\mathcal{V}$: **PRED** $\mapsto$ **BOOLEAN**, corresponding to the considered interpretation $\mathcal{I}$ in the following way:

$$\begin{array}{ll}
\mathcal{V}P_\phi & := \mathcal{I}P_\phi \\
\mathcal{V}P_{s_1...s_n}t_1...t_n & := (\mathcal{I}t_1...\mathcal{I}t_n) \in R_{s_1...s_n} \\
\mathcal{V}t_1 = t_2 & := \mathcal{I}t_1 \equiv \mathcal{I}t_2 \equiv \text{ represents identity in } \mathcal{D} \\
\mathcal{V}\neg a & := \textbf{not } \mathcal{V}a \\
\mathcal{V}(a \wedge b) & := \mathcal{V}a \textbf{ and } \mathcal{V}b \\
\mathcal{V}\forall x_s\, a & := \textbf{ T } \texttt{iff} \textit{ for all } u_s \textit{ in } D_s\ \mathcal{V}^{u_s}_{x_s}a = \textbf{T}\ (^*)
\end{array}$$

(*) Notation: we denote $\mathcal{I}^{u_s}_{x_s}$ each interpretation, such that for a given interpretation $\mathcal{I}$

$$\begin{cases} \mathcal{I}^{u_s}_{x_s}x_s = u_s \in D_s \\ \mathcal{I}^{u_s}_{x_s}y_s = \mathcal{I}y_s \textit{ if } x_s \not\equiv y_s \end{cases}$$

$\mathcal{V}^{u_s}_{x_s}$ is the corresponding *evaluation.*

It follows that $\mathcal{V}\forall x_s a = \mathbf{F}$ iff $\mathcal{V}^{u_s}_{x_s}a = \mathbf{F}$ for some $u_s \in D_s$.

Note that **not** and **and** satisfy the following definition-tables:

not	
T	F
F	T

and	T	F
T	T	F
F	F	F

Definition $L \models a$

We say that $\mathcal{I}$ is *a model for a list L of formulas* iff $\mathcal{V}a \equiv \mathbf{T}$ for each $a \in L$.

a is a consequence of a list L of formulas ($L \models a$) iff for each model $\mathcal{I}$ for L, $\mathcal{V}a \equiv \mathbf{T}$, where $\mathcal{V}$ denotes the valuation corresponding to $\mathcal{I}$.

Remark

Note that the *decidability* of $L \models a$ is no longer trivial, and depends on the specific language.

3.2.6 The derivability relation: $\vdash$

Deduction rules:

It is sufficient to extend rules (A), (C), (C$_i$), (C) and (X) of the **proposition calculus**, with the following rules, to get **predicate calculus**

(G) $\dfrac{L \ \vdash \ \forall x\, a}{L \ \vdash \ a}$ removing the generalisor

$$(\mathrm{G}_x) \quad \frac{L \ \vdash \ a}{L \ \vdash \ \forall x\, a} \qquad \textit{if } x \textit{ not free in } L \ (^1) \text{ introduction of the generalisor}$$

$$(\mathrm{S}_x^t) \quad \frac{L \qquad \vdash \ a}{L(t/x)\ (^2) \ \vdash \ a(t/x)\ (^3)} \qquad \text{substitution rule}$$

$$(\mathrm{E}) \quad \vdash \ t = t \qquad \text{first equality rule}$$

$$(\mathrm{E}_x^t) \quad \frac{L \qquad \vdash \ a}{L, x = t \ \vdash \ a(t/x)} \qquad \text{second equality rule}$$

(1) It is possible to define a calculus to decide "x is free in L". We refer to *Hermes* for such a calculus.

(2) $L(t/x)$ is the list of formulas, obtained from L, by *substitution of t for x in each formula from L* (see (3))

(3) $a(t/x)$ is the formula obtained from a by *substitution of t for x in a*. Note that in the first place t and x much have the same sort. Moreover $a(t/x)$ only exists if "x is free for t in a"; roughly speaking this means that is is excluded that free variables from t become bound in a, after substitution.

E.g. Let $a \equiv \forall z(add(x, z) = z)$ and $t \equiv add(y, u)$ then $a(t/x) \equiv \forall z(add(add(y, u), z) = z)$, but for $t \equiv add(z, u)$, $a(t/x)$ is not defined.

Note that this restriction to the rules (S_x^t) and (E_x^t) prevent us deriving wrong conclusions. The following example illustrates this point.

Let *REALS* be the list of axioms for the theory of the "**reals**", then we have

$$REALS\, x * 1 = 1 \ \vdash \ \forall z(x * z = z)$$

From (S_x^t), with $t \equiv u * y$ we get

$$REALS\, u * y * 1 = 1 \ \vdash \ \forall z(u * y * z = z)$$

It is clear however, that if we apply that rule for $t \equiv u * z$, we get

$$REALS\, u * z * 1 = 1 \ \vdash \ \forall z(u * z * z = z)$$

which clearly does not hold.

Definition A **deduction** in the predicate calculus is a sequence

$$L_1 \vdash a_1, ..., L_n \vdash a_n$$

where each of the $L_i \vdash a_i$ is either an application of the rule (A) or (E) or follows from a preceding $L_j \vdash a_j$ applying (C_i), (G), (G_x), (S_x^t) or (E_x^t) or follows from preceding $L_j \vdash a_j$ and $L_k \vdash a_k$ applying one of the rules (C), (R), (X). We say that $L_n \vdash a_n$ is **derivable**. We also say the a_n is deducible from L_n.

Theorem

The considered predicate calculus is

sound *:* *if* $L \vdash a$ *then* $L \models a$*;*
complete*:* *if* $L \models a$ *then* $L \vdash a$*.*

Theorem

$\vdash$ *is undecidable*

An important practical problem is to find an algorithm for establishing the derivability of $L \vdash a$, for any L and a.

Assume that the alphabet of our predicate language is *finite*, i.e. we have a finite set S of sorts, and a finite signature Σ. Then the number of formulae with a fixed total number of symbols is *finite*. Hence the number of proofs with a given length is *finite*. Therefore the set of all proofs is countable, by taking them in order according to the number of symbols they contain and according to the ordering of the alphabet. If $L \vdash a$ is derivable, we will find its proof in the list after a finite number of trials. However, if $L \vdash a$ is not derivable, this algorithm continues for ever, looking unsuccessfully for a proof.

According to *Church's thesis* we may conclude that $L \vdash a$ is undecidable.

4 Partial predicates in a 3-valued calculus

4.1 The Language

In the first place we may copy the language of the 2-valued Predicate Predicate calculus (**2.2.3** and **2.2.4**)), i.e we have

signature of the language

alphabet

term

atomic

formula

The fact that the set of basic logical connectives

$\{\neg, \wedge, \forall\}$

must be extended with a new symbol

Δ

can be discussed.

In order to show some fundamental results concerning *undefinedness in computer science*, the symbol Δ is indispensable, and hence we add a new construction rule to the syntax of a **formula**

formula := Δ *formula*

The basic differences between the 2-valued and 3-valued logic, follow from the definition of the relations $L \models a$ and $L \vdash a$.

4.2 Truth-tables in PPC

PPC stands for **P**artial **P**redicate **C**alculus, and has been developed to deal with the *undefinedness notion* in a formal way. For the definition of $L \models a$, we consider the *truth-tables* for the connectives $\{\neg, \wedge, \Delta\}$.

We note that in the introduction we already considered the 3-valued connectives

`OR, AND, NOT`

from **Turbo5**.

In view of the fact that the `AND` was not commutative, we prefer to consider a new table for $\wedge$. We get:

$\wedge$	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

	$\neg$	Δ	J_T	J_F	J_U
T	F	T	T	F	F
F	T	T	F	T	F
U	U	F	F	F	T

Remarks

- Note that we have

$$\begin{aligned} J_T a &= a \wedge \Delta a \\ J_F a &= \neg a \wedge \Delta a \\ J_U a &= \neg \Delta a \end{aligned}$$

- Also note that this system of connectives is not *complete*, since there is no way to define the constant function

  $$\textit{undef}\ \{\mathrm{T,F,U}\} \mapsto \{\mathrm{U}\}$$

  using the connectives $\{\neg,\ \wedge,\ \Delta\}$.

  However, this not our objective, and moreover is such a function of no interest for our applications of logic in computer science.

Before we proceed, we want to show that the **Turbo5** "`AND`" and "`OR`" can be defined in the system. Hence we consider the following function table:

a	b	$a\,\texttt{AND}\,b$	$a \wedge b$	$\neg a$	$\neg a \wedge a$	$(\neg a \wedge a) \vee (a \wedge b)$
T	T	T	T	F	F	T
T	F	F	F	F	F	F
T	U	U	U	F	F	U
F	T	F	F	T	F	F
F	F	F	F	T	F	F
F	U	F	F	T	F	F
U	T	U	U	U	U	U
U	F	U	F	U	U	U
U	U	U	U	U	U	U

where $a \vee b$ is an abreviation for $\neg(\neg a \wedge \neg b)$, and hence satisfies the following table

$\vee$	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

So we conclude that

$$a\,\texttt{AND}\,b \equiv (\neg a \wedge a) \vee (a \wedge b)$$

and hence that `AND` is not symmetric.

From the fact that $\texttt{NOT}\,a \equiv \neg a$ and $a\,\texttt{OR}\,b \equiv \texttt{NOT}\,(\texttt{NOT}\,a\ \texttt{AND}\ \texttt{NOT}\,b)$, we get

$$
\begin{aligned}
a\,\mathtt{OR}\,b &\equiv \neg((\neg\neg a \wedge \neg a) \vee (\neg a \wedge \neg b)) \\
&\equiv \neg(a \wedge \neg a) \wedge \neg(\neg a \wedge \neg b) \\
&\equiv (\neg a \vee a) \wedge (a \vee b)
\end{aligned}
$$

4.3 The consequence relation: $\models$

4.3.1 Signature of a theory

Definition An S-sorted *signature of a theory* is a tuple $\Sigma =< \Omega, \Pi >$ where Ω is an S-sorted operational signature and Π is an S-sorted relational signature as in the 2-valued case.

4.3.2 Domain for the language

As in the 2-valued case we must fix a "*domain*" for the language, before we can introduce an *interpretation*.

If $\Sigma =< \Omega, \Pi >$ is the signature of the language **PPC**, a **domain** for **PPC**, is a so-called **partial Σ-algebra** $\mathcal{D}$. It consist of an S-sorted set D, plus an **interpretation** $\mathcal{I}$ of Σ in D, which is:

- a family of arrows

  $i_{s_1 \ldots s_n, s} : \Omega_{s_1 \ldots s_n, s} \mapsto [D^{s_1 \ldots s_n} \mapsto D_s, \mathcal{P}(D^{s_1 \ldots s_n})]$ for each *rank* $< s_1 \ldots s_n, s >$ in Ω.

  which interpret the operation symbols $f_{s_1 \ldots s_n, s}$ in Σ as actual operations $F_{s_1 \ldots s_n, s}$ on $D^{s_1 \ldots s_n}$ with given **definition-domains** $domF_{s_1 \ldots s_n, s} \subseteq D^{s_1 \ldots s_n}$.

  notation: $\mathcal{I} f_{s_1 \ldots s_n, s} = (F_{s_1 \ldots s_n, s}, domF_{s_1 \ldots s_n, s})$

  Note that if $\Omega_{\phi, s} \neq \{\ \}$ for each $c_s \in \Omega_{\phi, s}$, $\mathcal{I}(c_s)$ will be a *constant* in D_s and the **definition-domain** is not considered.

- a family of arrows

  $i_{s_1 \ldots s_n} : \Pi_{s_1 \ldots s_n} \mapsto [\mathcal{P}(D^{s_1 \ldots s_n}), \mathcal{P}(D^{s_1 \ldots s_n})]$ for each *domain-type* $s_1 \ldots s_n$ in Π.

  which interpret the relation symbols $P_{s_1 \ldots s_n}$ in Σ as actual relations $R_{s_1 \ldots s_n}$ on $D^{s_1 \ldots s_n}$ with given **definition-domains** $domR_{s_1 \ldots s_n} \subseteq D^{s_1 \ldots s_n}$ such that
  $R_{s_1 \ldots s_n} \subseteq domR_{s_1 \ldots s_n} \subseteq D^{s_1 \ldots s_n}$.

  Note that if $\Pi_\phi \neq \{\ \}$ for each $P \in \Pi_\phi$, $\mathcal{I}(P) \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}$

- a family of arrows

  i_s : *variable_identifier of sort* $s \mapsto D_s$

4.3.3 Interpretation of terms

The interpretation defined above, can be extended to an *interpretation of terms.* If we preserve the notation $\mathcal{I}$, we get

$$\mathcal{I}(f_{s_1...s_n,s}t_{s_1}...t_{s_n}) = F_{s_1...s_n,s}(\mathcal{I}t_{s_1}, ..., \mathcal{I}t_{s_n})$$

4.3.4 The meta-predicate $def_{\mathcal{I}}$

To be able to turn this *interpretation of terms* into a *valuation of formulas* we introduce a **meta predicate** $def_{\mathcal{I}}$

Definition

- we say that $def_{\mathcal{I}}x$ "holds" for any *variable-identifier* x
- we say that $def_{\mathcal{I}}c$ "holds" for any *constant-identifier* c
- If $t_{s_1}, ..., t_{s_n}, t_s$ are terms, corresponding to the *rank* $< s_1...s_n, s >$ of $f_{s_1...s_n,s} \in \Sigma$ such that $t_s = f_{s_1...s_n,s}t_{s_1}...t_{s_n}$, then

  $def_{\mathcal{I}}t$ iff $def_{\mathcal{I}}t_{s_1}$,...$def_{\mathcal{I}}t_{s_n}$ and $(\mathcal{I}t_{s_1},...,\mathcal{I}t_{s_n}) \in domF_{s_1...s_n,s}$, where
  $\mathcal{I}f_{s_1...s_n,s} = (F_{s_1...s_n,s}, domF_{s_1...s_n,s})$

4.3.5 Valuation of a formula

We define the valuation of a formula $\mathcal{V} : \mathbf{PPC} \mapsto \{\mathbf{T,F,U}\}$, corresponding to the considered interpretation $\mathcal{I}$ in the following way:

$$\mathcal{V}P_\phi \;:= \mathcal{I}P_\phi \in \{\mathbf{T,F,U}\}$$

$$\begin{array}{lll}
\mathcal{V}P_{s_1...s_n}t_1...t_n & := \mathbf{T} & \texttt{iff} \;\; (\texttt{fall}\, i\; def_{\mathcal{I}}t_{s_i}) \;\texttt{and}\; (\mathcal{I}t_1...\mathcal{I}t_n) \in R_{s_1...s_n} \\
 & := \mathbf{F} & \texttt{iff} \;\; (\texttt{fall}\, i\; def_{\mathcal{I}}t_{s_i}) \;\texttt{and}\; (\mathcal{I}t_1...\mathcal{I}t_n) \in domR_{s_1...s_n} - R_{s_1...s_n} \\
 & := \mathbf{U} & \texttt{iff} \;\; [(\texttt{fall}\, i\; def_{\mathcal{I}}t_{s_i}) \;\texttt{and}\; (\mathcal{I}t_1...\mathcal{I}t_n) \in D^{s_1...s_n} - domR_{s_1...s_n}] \\
 & & \qquad\quad \texttt{or}\; [\texttt{exists}\, i\; \texttt{not}\; def_{\mathcal{I}}t_{s_i}]
\end{array}$$

$$\begin{array}{lll}
\mathcal{V}(t_1 = t_2) & := \mathbf{T} & \texttt{iff} \;\; def_{\mathcal{I}}t_1 \;\texttt{and}\; def_{\mathcal{I}}t_2 \;\texttt{and}\; \mathcal{I}t_1 \equiv \mathcal{I}t_2 \\
 & := \mathbf{F} & \texttt{iff} \;\; def_{\mathcal{I}}t_1 \;\texttt{and}\; def_{\mathcal{I}}t_2 \;\texttt{and}\; \mathcal{I}t_1 \not\equiv \mathcal{I}t_2 \\
 & := \mathbf{U} & \texttt{iff} \;\; \texttt{not}\, def_{\mathcal{I}}t_1 \;\texttt{or}\; \texttt{not}\, def_{\mathcal{I}}t_2
\end{array}$$

$$\begin{array}{ll}
\mathcal{V}\neg a & := \textbf{NOT}\; \mathcal{V}a \\
\mathcal{V}(a \wedge b) & := \mathcal{V}a\; \textbf{AND}\; \mathcal{V}b
\end{array}$$

$$\begin{array}{llll}
\mathcal{V}\forall x_s\, a & := \mathbf{T} & \texttt{iff} \;\; \texttt{fall}\, u_s \in D_s & \mathcal{V}^{u_s}_{x_s}a = \mathbf{T} \;\; (*) \\
 & := \mathbf{F} & \texttt{iff} \;\; \texttt{exists}\; u_s \in D_s & \mathcal{V}^{u_s}_{x_s}a = \mathbf{F} \\
 & := \mathbf{U} & \texttt{iff} \;\; \texttt{fall}\, u_s \in D_s & \mathcal{V}^{u_s}_{x_s}a \in \{\mathbf{T,U}\} \\
 & & \quad\;\; \texttt{and exists}\, u_s \in D_s & \mathcal{V}^{u_s}_{x_s}a = \mathbf{U}
\end{array}$$

(*) Notation: we denote $\mathcal{I}_{x_s}^{u_s}$ each interpretation, such that for a given interpretation $\mathcal{I}$

$$\begin{cases} \mathcal{I}_{x_s}^{u_s} x_s = u_s \in D_s \\ \mathcal{I}_{x_s}^{u_s} y_s = \mathcal{I} y_s \ \textit{if}\ x_s \not\equiv y_s \end{cases}$$

$\mathcal{V}_{x_s}^{u_s}$ is the corresponding *evaluation.*

Note that **NOT** and **AND** satisfy the following definition-tables:

NOT	
T	F
F	T
U	U

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

4.3.6 The consequence relation $L \models a$

Definition

We say that $\mathcal{I}$ is *a model for a list L of formulas* iff $\mathcal{V}a \equiv \mathbf{T}$ for each $a \in L$.

Definition

For the definition of "*a formula a is* **valid**", we have two possibilities

1. *A formula a is* **valid** iff $\mathcal{V}a \in \{\mathbf{T},\mathbf{U}\}$
2. *A formula a is* **valid** iff $\mathcal{V}a \equiv \mathbf{T}$

Both definitions have their applications.

We have always promoted the first concept, and we were able to show that one can pass from the first definition to the latter. As a consequence we could derive some important remarks concerning the use of a *Partial Predicate Calculus* for software specifications.

Definition A formula *a is a consequence of a list L of formulas* ($L \models a$) iff a is *valid* for each model $\mathcal{I}$ of L.

Example

To illustrate the expressive power of the first definition, we consider the following example:

Consider a 1-sorted signature $\sigma =< \{\Omega_{ss,s}, \Omega_{s,s}, \Omega_{\phi,s}\}, >$, where

$\Omega_{ss,s} = \{f_{ss,s}\}$, $\Omega_{s,s} = \{f_{s,s}\}$, $\Omega_{\phi,s} = \{c_s\}$.

$\{u, v, w\}$: *variable_identifier* of sort s.

Let L be

$[\forall x \forall y \forall z\ f_{ss,s} x f_{ss,s} y z = f_{ss,s} x y f_{ss,s} z,$
$\forall x f_{ss,s} x c_s = x,$
$\forall x f_{ss,s} c_s x = x,$
$f_{ss,s} f_{s,s} u u = c_s]$

Let α be $(f_{ss,s}uv = f_{ss,s}uw \rightarrow v = w)$ where $(a \rightarrow b) \equiv (\neg a \vee b)$

then we have

$L \models \alpha$

In words this means:

In a semigroup with unity, each element that has a left inverse, is left cancelable

Now the formulas of L are *valid* for $(\mathcal{I}, \mathbf{Z}/6)$, where

$\mathcal{I} f_{ss,s}$:= "$*$"

$\mathcal{I} f_{s,s}$:=<*taking the invers if it exists*>

$\mathcal{I} c_s := 1$, $\mathcal{I} u := 3$,$\mathcal{I} v := 1$, $\mathcal{I} w := 5$

For this interpretation however we get

$3 * 1 = 3 * 5 \,(\mathrm{mod} 6)$

but

$1 \not\equiv 5$

and hence $\mathcal{V}\alpha \equiv \mathbf{F}$.

Note however that this fact does not contradicts $L \models \alpha$, since $f_{ss,s} f_{s,s} uu = c_s$ is not defined for this interpretation, since $\mathcal{I} u$ has no inverse in $\mathbf{Z}/6$.

Hence $\mathcal{I}$ is not a model for L

4.4 The deduction calculus PPC

4.4.1 Rules for PPC

Classical Rules

(A) $\quad a \vdash a$;

(C) $\quad \dfrac{L_1 \vdash a \quad L_2 \vdash b}{L_{12} \vdash a \wedge b}$; $\qquad$ (C_1) $\quad \dfrac{L \vdash a_1 \wedge a_2}{L \vdash a_1}$; $\qquad$ (C_2) $\quad \dfrac{L \vdash a_1 \wedge a_2}{L \vdash a_2}$;

(G) $\quad \dfrac{L \vdash \forall x\, a}{L \vdash a}$; $\qquad$ (G_x) $\quad \dfrac{L \vdash a}{L \vdash \forall x\, a}$ *if x not free in L*

(E) $\quad \vdash t = t$; $\qquad$ (E_x^t) $\quad \dfrac{L \vdash a}{L, x = t \vdash a(t/x)}$;

Modified Classical Rules

$$(\mathrm{R}') \quad \begin{array}{lrcl} L_1 & a & \vdash & b \\ L_2 & \neg a & \vdash & b \\ L_3 & \neg\Delta a & \vdash & b \\ \hline L_{123} & & \vdash & b \end{array} \qquad \text{removal rule}$$

$$(\mathrm{X}') \quad \begin{array}{lcl} L_1 & \vdash & a \\ L_2 & \vdash & \neg a \\ L_3 & \vdash & \Delta a \\ \hline L_{123} & \vdash & b \end{array} \qquad \text{contradiction rule}$$

$$(\mathrm{S}'^{t}_{x}) \quad \begin{array}{lcl} L_1 & \vdash & a \\ L_2 & \vdash & \Delta t\ (^1) \\ \hline L_{23}\ (^2) & \vdash & a(t/x) \end{array} \qquad \text{substitution rule}$$

(1) $\Delta t := \Delta(t = t)$, to be considered as an abreviation.

(2) $L_3 := L_1(t/x)$ and $L_{23} := L_2 \cup L_3$.

Rules concerning Δ

(Ad_1) $a \vdash \Delta a$; (Ad_2) $\neg a \vdash \Delta a$;

(D_0) $\vdash \Delta\Delta a$;

(D_1) $\neg\Delta a \vdash a$; (D_2) $\neg\Delta a \vdash \neg a$;

(Dn_1) $\Delta a \vdash \Delta\neg a$; (D_2) $\Delta\neg a \vdash \Delta a$;

(C'_1) $a_1 \wedge a_2 \vdash \Delta a_1$; (C'_2) $a_1 \wedge a_2 \vdash \Delta a_2$;

$$(\mathrm{Cd}_0) \quad \begin{array}{lcl} L_1 & \vdash & \Delta a_1 \\ L_2 & \vdash & \Delta a_2 \\ \hline L_{12} & \vdash & \Delta(a_1 \wedge a_2) \end{array};$$

$$(\mathrm{Cd}_1) \quad \begin{array}{lcl} L_1 & \vdash & \neg a_1 \\ L_2 & \vdash & \Delta a_1 \\ \hline L_{12} & \vdash & \Delta(a_1 \wedge a_2) \end{array}; \qquad (\mathrm{Cd}_2) \quad \begin{array}{lcl} L_1 & \vdash & \neg a_2 \\ L_2 & \vdash & \Delta a_2 \\ \hline L_{12} & \vdash & \Delta(a_1 \wedge a_2) \end{array};$$

(Gd_1) $\forall x\, a \vdash \Delta a$; (Gd_2) $\dfrac{L \vdash \forall x\, \Delta a}{L \vdash \Delta\forall x\, a}$;

(Gd$_3$) $\neg a \vdash \Delta\forall x\, a$;

(Ed$_1$) $\vdash \Delta x$ (Ed$_2$) $\vdash \Delta c$

(De$_0$)
$$\begin{array}{lll} L_1\ \neg\Delta t_1 & \vdash & a \\ L_2\ \neg\Delta t_2 & \vdash & a \\ \hline L_{12}\ \neg\Delta(t_1 = t_2) & \vdash & a \end{array}$$

(De$_1$) $\Delta(t_1 = t_2) \vdash \Delta t_1$; (De$_2$) $\Delta(t_1 = t_2) \vdash \Delta t_2$;

4.4.2 The notion: "deduction"

A **deduction** is a sequence of expressions of the form

$L_1 \vdash a_1, ..., L_n \vdash a_n$

where each of the $L_i \vdash a_i$ is either an application of a "*one line rule*" or follows from a preceding $L_j \vdash a_j$ applying a "*two lines rule*" or follows from preceding $L_j \vdash a_j$ and $L_k \vdash a_k$ applying a "*three lines rule*", or follows from preceding $L_j \vdash a_j$, $L_k \vdash a_k$ $L_l \vdash a_l$ applying a "*four lines rule*".

4.4.3 The soundness and completness theorem

The calculus **PPC** is sound and complete with respect to the definition of "$\models$" that uses the first definition of "*valid formulas*".

4.4.4 Derived deduction rules

In order to make the system more useful in practice, we need a number of derived rules.

Classical rules

Some of the classical rules remain valid, but need new proofs.

(SeAt) $\dfrac{L\ \neg a \ \vdash\ a}{L \ \vdash\ a}$ (Self Assertion)

$$\begin{array}{llrcll} (1) & L & \neg a & \vdash & a & \\ (2) & & a & \vdash & a & (A) \\ (3) & & \neg\Delta a & \vdash & a & (D_1) \\ (4) & L & & \vdash & a & (R')(2,1,3) \end{array}$$

(SeDe) $\dfrac{L,\ a \ \vdash\ \neg a}{L \ \vdash\ \neg a}$ (Self Denial) *similar proof*

$$(\mathrm{DdRu}_1)\quad \frac{a \vdash b}{\vdash \neg(a \wedge \neg b)} \qquad \text{cfr.} \qquad \frac{a \vdash b}{\vdash a \to b}$$

$$(\mathrm{DdRu}_2)\quad \frac{\vdash \neg(a \wedge \neg b)}{a \vdash b} \qquad \text{cfr.} \qquad \frac{\vdash a \to b}{a \vdash b}$$

(NN_1) $\quad a \vdash \neg\neg a$

(NN_2) $\quad \neg\neg a \vdash a$

Modified classical rules

Other classical rules no longer hold, and need some modification:

Cut Rule:

$$(\mathrm{CuRu'})\quad \frac{\begin{array}{lcl} L_1 & \vdash & a \\ L_2,\ a & \vdash & b \\ L_3 & \vdash & \Delta a \end{array}}{\begin{array}{lcl} L_{123} & \vdash & b \end{array}}$$

(1)	L_1		$\vdash$	a	
(2)	L_2	a	$\vdash$	b	
(3)	L_3		$\vdash$	Δa	
(4)		$\neg a$	$\vdash$	$\neg a$	(A)
(5)	L_{13}	$\neg a$	$\vdash$	b	$(X')(1,4,3)$
(6)		$\neg\Delta a$	$\vdash$	$\neg\Delta a$	(A)
(7)			$\vdash$	$\Delta\Delta a$	(D_0)
(8)	L_3	$\neg\Delta a$	$\vdash$	b	$(X')(3,6,7)$
(9)	L_{123}		$\vdash$	b	$(R')(2,5,8)$

Note that we can show the so-called

Short Cut Rule:

$$(\mathrm{ShCu'})\quad \frac{\begin{array}{lcl} L_1 & \vdash & \Delta a \\ L_2,\ \Delta a & \vdash & b \end{array}}{\begin{array}{lcl} L_{12} & \vdash & b \end{array}}$$

(1)	L_1		$\vdash$	Δa	
(2)	L_2	Δa	$\vdash$	b	
(3)				$\Delta\Delta a$	(D_0)
(4)	L_{12}		$\vdash$	b	$(CuRu')(1,2,3)$

4.5 The deduction calculus BCJ

We will denote $L \models_b a$ for the second interpretation of "$\models$":

$L \models_b a$ `iff` "*a is valid for each model of L*", where "*a is valid*" means $\mathcal{V}a = \mathbf{T}$.

In the literature we find a formalization of this interpretation in *Barringer, Cheng and Jones: "A logic Covering Undefinedness in Program Proofs"* Acta Inf 21, pp251-269 (1984).

We first note that they use the so-called *natural deduction* style, to formalize their system, instead of the *assumption calculus.*

4.5.1 Natural deduction for 2-valued Logic

To see the difference, we reconsider the classical 2-valued predicate calculus in this formalization:

$$\frac{a, b}{a \wedge b}\,; \qquad \frac{a_1 \wedge a_2}{a_1}; \qquad \frac{a_1 \wedge a_2}{a_2};$$

$$\frac{a, \neg a}{b}; \qquad \frac{a \vdash b\,;\ \neg a \vdash b}{b};$$

$$\frac{\forall x\, a}{a}; \qquad \frac{a\,;\ x = t}{a(t/x)}; \qquad \frac{}{t = t};$$

- Note that that the rules (S^t_x) and (G_x) give some problems within this formalism, so we need to consider:

  $$\frac{L \quad \vdash \quad a}{L(t/x) \quad \vdash \quad a(t/x)}; \qquad \frac{L \quad \vdash \quad a}{L \quad \vdash \quad \forall x\, a} \quad \textit{if } x \textit{ is not free in } L$$

  Example:

  $$\begin{array}{lrcl} & (x-2) * f_1(x) & = & (x-2) * f_2(x) \\ x \neq 2 & f_1(x) & = & f_2(x) \\ & & \cdots & \\ & f_1(x) * f_3(x) & = & f_4(x) \\ & f_1(2) * f_3(2) & = & f_4(2) \end{array}$$

For this reason one prefers to use the connectives "$\vee$" and "$\exists$" instead of "$\wedge$" and "$\forall$".

Then we get:

$$(\vee\text{–I}_1)\ \frac{a_1}{a_1 \vee a_2}; \qquad (\vee\text{–I}_2)\ \frac{a_2}{a_1 \vee a_2}; \qquad (\vee\text{-E})\ \frac{a_1 \vee a_2\,;\ a_1 \vdash b\,;\ a_2 \vdash b}{b};$$

$$(\mathrm{X})\ \frac{a\ ;\ \neg a}{b}; \qquad (\mathrm{R})\ \frac{a \vdash b\ ;\ \neg a \vdash b}{b};$$

$$(\exists\text{-I})\ \frac{a(t/x)}{\exists x\ a}; \qquad (\exists\text{-E})\ \frac{\exists x\ a(x)\ ;\ a(y/x) \vdash b}{b}\ \textit{y not free in } b;$$

$$(=\text{-subst})\ \frac{t_1 = t_2\ ,\ a}{a(t_1/t_2)}; \qquad (=\text{-term})\ \frac{}{t = t}$$

Example of some deductions:

a)
$$\dfrac{\dfrac{\dfrac{\neg a}{\neg\neg a}\ (\mathrm{X}) \quad \dfrac{\neg\neg a}{\neg\neg a}}{\neg\neg a}\ (\mathrm{R})}{}$$

hence: $\dfrac{a}{\neg\neg a}$

b)
$$\dfrac{\dfrac{a_1 \vee a_2 \quad \dfrac{\neg a_1 \quad a_1}{\neg(a_1 \vee a_2)}\ (\mathrm{X}) \quad \dfrac{\neg a_2 \quad a_2}{\neg(a_1 \vee a_2)}\ (\mathrm{X})}{\neg(a_1 \vee a_2)}\ (\vee\text{ - E}) \quad \dfrac{\neg(a_1 \vee a_2)}{\neg(a_1 \vee a_2)}}{\neg(a_1 \vee a_2)}\ (\mathrm{R})$$

hence: $\dfrac{\neg a_1, \neg a_2}{\neg(a_1 \vee a_2)}$

Note that the **(CuRu)** is a natural part of this system. This means that

$$\text{if } \frac{\frac{a}{b}}{c} \qquad \text{then} \qquad \frac{a}{c}.$$

Or if $a \vdash b$ and $b \vdash c$ then $a \vdash c$.

Natural Deduction Style proofs in a structured presentation

Those examples show that the way of writing down proofs, may be fast if executed with pen and paper, but they can hardly be reconstructed or verified. Hence a more structured way of presenting proofs is needed. In this way we may rewrite the presented proofs in the following way:

a) $a \vdash \neg\neg a$ *goal*

1. $\neg a \vdash \neg\neg a$ *subgoal*
   - 1.1 a *premise*
   - 1.2 $\neg a$ *premise 1.1*
   - 1.3 $\neg\neg a$ (X)(1.1,1.2)
2. $\neg\neg a \vdash \neg\neg a$
3. $\neg\neg a$ (R)(1.,2.)

b) $\neg a_1, \neg a_2 \vdash \neg(a_1 \vee a_2)$ *goal*

1. $a_1 \vee a_2 \vdash \neg(a_1 \vee a_2)$ *subgoal*
   - 1.1 $a_1 \vdash \neg(a_1 \vee a_2)$ *subsubgoal*
     - 1.1.1. $\neg a_1$ *premise*
     - 1.1.2. a_1 *premise 1.1*
     - 1.1.3. $\neg(a_1 \vee a_2)$ (X)$(-1, -2)$
   - 1.2 $a_2 \vdash \neg(a_1 \vee a_2)$ *subsubgoal*
     - 1.2.1. $\neg a_2$ *premise*
     - 1.2.2. a_2 *premise 1.2*
     - 1.2.3. $\neg(a_1 \vee a_2)$ (X)$(-1, -2)$
   - 1.3 $a_1 \vee a_2$ *premise 1.*
   - 1.4 $\neg(a_1 \vee a_2)$ ($\vee$-E)$(1.3, 1.1, 1.2)$
2. $\neg(a_1 \vee a_2) \vdash \neg(a_1 \vee a_2)$
3. $\neg(a_1 \vee a_2)$ (R)$(1., 2.)$

c) Let $a_1 \wedge a_2 \equiv \neg(\neg a_1 \vee \neg a_2)$. Show $a_1 \wedge a_2 \vdash a_1$

$\neg(\neg a_1 \vee \neg a_2) \vdash a_1$ *goal*

1. $\neg a_1 \vdash a_1$ *subgoal*
   - 1.1 $\neg a_1$ *premise 1.*
   - 1.2 $\neg a_1 \vee \neg a_2$ ($\vee$-I)(1.1)
   - 1.3 $\neg(\neg a_1 \vee \neg a_2)$ *premise*
2. $a_1 \vdash a_1$
3. a_1 (R)(1.,2.)

4.5.2 The deduction rules for BCJ

Classical rules

$$(\vee\text{–I}_1)\ \frac{a_1}{a_1 \vee a_2}; \qquad (\vee\text{–I}_2)\ \frac{a_2}{a_1 \vee a_2}; \qquad (\vee\text{-E})\ \frac{a_1 \vee a_2\ ;\ a_1 \vdash b\ ;\ a_2 \vdash b}{b};$$

$$(\neg\vee\text{-I})^{(*)}\ \frac{\neg a_1; \neg a_2}{\neg(a_1 \vee a_2)}; \qquad (\neg\vee\text{-E}_1)^{(*)}\ \frac{\neg(a_1 \vee a_2)}{\neg a_1}; \qquad (\neg\vee\text{-E}_2)^{(*)}\ \frac{\neg(a_1 \vee a_2)}{\neg a_2};$$

$$(\neg\neg\text{-I})^{(*)}\ \frac{a}{\neg\neg a}; \qquad (\neg\neg\text{-E})^{(*)}\ \frac{\neg\neg a}{a};$$

$$(\text{X})\ \frac{a\ ;\ \neg a}{b};$$

$$(\exists\text{-I})\ \frac{a(t/x)\ ;\ t = t}{\exists x\ a}; \qquad (\exists\text{-E})\ \frac{\exists x\ a(x)\ ;\ a(y/x) \vdash b}{b}\ \textit{with } y \textit{ not free in } b;$$

$$(\neg\exists\text{-I})^{(*)}\ \frac{\neg a(x)}{\neg\exists\ a(x)}; \qquad (\neg\exists\text{-E})^{(*)}\ \frac{\neg\exists x\ a(x)\ ;\ t = t}{\neg a(t/x)};$$

$$(=\text{-subst})\ \frac{t_1 = t_2\ ,\ a}{a(t_1/t_2)}; \qquad (=\text{-contr})\ \frac{\neg(t = t)}{a};$$

$$(=\text{-const})\ \frac{}{c = c}; \qquad (=\text{-var})\ \frac{}{x = x};$$

Rules concerning Δ

$$(\Delta\text{-I}_1)\ \frac{a}{\Delta a}; \qquad (\Delta\text{-I}_2)\ \frac{\neg a}{\Delta a}; \qquad (\Delta\text{-E})\ \frac{\Delta a\ ;\ a \vdash b\ ;\ \neg a \vdash b}{b};$$

$$(\neg\Delta\text{-I})\ \frac{\Delta a \vdash b\ ;\ \Delta a \vdash \neg b}{\neg\Delta a}; \qquad (\neg\Delta\text{-E})\ \frac{\neg\Delta a \vdash b\ ;\ \neg\Delta a \vdash \neg b}{\Delta a};$$

$$(\neg\Delta\text{-contr}_1)\ \frac{t_1 = t_2\ ;\ \neg\Delta(t_1 = t_2)}{a}; \qquad (\neg\Delta\text{-contr}_2)\ \frac{\neg(t_1 = t_2)\ ;\ \neg\Delta(t_1 = t_2)}{a};$$

(*) Note that those rules must be given explicitly, since there is no way to prove them by the *contradiction* or the *excluded middle* principle.

4.6 PPC versus BCJ

In the sequel we will use the following notations:

(1) $L \vdash_{PPC} a \iff L \models_{PPC}$ for **PPC**
(2) $L \vdash_{BCJ} a \iff L \models_{BCJ}$ for **BCJ**

4.6.1 The Δ-rule

A very import distinction between (1) and (2) is

$$(\Delta_{BCJ})\ \frac{L \ \vdash_{BCJ} \ a}{L \ \vdash_{BCJ} \ \Delta a}$$

We note that in **PPC** this Δ-rule is not deducible, and that

$L \vdash_{PPC} a$ and $a \vdash_{PPC} \Delta a$ (Ad_1)

do not imply

$L \vdash_{PPC} \Delta a$

Now we try to add

$$(\Delta_{PPC})\ \frac{L \ \vdash_{PPC} \ a}{L \ \vdash_{PPC} \ \Delta a}$$

to the rules of **PPC**.

4.6.2 Remarks

a) First we have to note that this rule is *consistent* with **PPC**.

   Indeed, if one takes each *function* and each *predicate* as *everywhere defined*, one gets a model for that theory.

b) In that case however, the notion of *partial function* disappears.

   From (E) $\vdash t = t$ and (Δ_{PPC}) we get $\vdash_{PPC} \Delta t$ for each term term t.

c) Moreover from (D_1) $\neg\Delta a \vdash a$ or (D_2) $\neg\Delta a \vdash \neg a$ and (Δ_{PPC}) we get

   $\neg\Delta a \vdash b$

   for each formula b.

   Hence every formula must be defined, since if not, the theory becomes trivial.

d) It follows from b) that we cannot keep the rule (E) $\vdash t = t$, if we want the formalization of $\models_b$.

4.6.3 Theorem

Within **PPC**, *we have for each term t:*

(E) $\vdash t = t$ *iff* *(E′)* $\neg(t = t) \vdash a$ *for any formula a.*

Proof.

$\Longrightarrow$) Assume (E) then (E′) follows.

$$\begin{array}{lcll} & \vdash & t = t & (E) \\ \neg(t = t) & \vdash & \neg(t = t) & (A) \\ \neg(t = t) & \vdash & \Delta t & (Ad_2) \\ \neg(t = t) & \vdash & a & (X') \end{array}$$

$\Longleftarrow$) From (E′) follows (E).

$$\begin{array}{lcll} \neg(t = t) & \vdash & t = t & (E') \\ t = t & \vdash & t = t & (A) \\ \neg\Delta(t = t) & \vdash & t = t & (D_1) \\ & \vdash & t = t & (R') \end{array}$$

QED

4.6.4 Corollaries

1. It follows that (=-contr) in **BCJ**, which is just an other way of writing (E′), is an alternative for the reflexivity of the equality relation.

2. If we replace in **PPC** (E) by (E′), and we remove (D_1) and (D_2), then

   PPC′= **PPC**−{D_1, D_2, E } + (E′)

   becomes a candidate for an equivalent formulation for **BCJ**.

We note however that **PPC′** is sound with $\models_b$, but not complete, since the Δ-rule is still missing.

Now we consider **PPC″**=**PPC′**+(Δ). Then we have

4.6.5 Theorem

In **PPC″** the rules

$$\text{(SeAt)}\ \frac{L, \neg a \ \vdash \ a}{L \ \vdash \ a} \qquad \text{and} \qquad \text{(SeDe)}\ \frac{L, a \ \vdash \ \neg a}{L \ \vdash \ \neg a}$$

cannot hold.

Proof. In order to get this conclusion, we show that

$\mathbf{PPC}' + (\text{SeAt}) \Longleftrightarrow \mathbf{PPC}' + (\text{D}_1)$

$\mathbf{PPC}' + (\text{SeDe}) \Longleftrightarrow \mathbf{PPC}' + (\text{D}_2)$

from remark b), it follows that in this theory all formulas must be defined.

(1)	$L\,,\,\neg a$	$\vdash$	a	
(2)	A	$\vdash$	a	(A)
(3)	$\neg\Delta a$	$\vdash$	a	(D_1)
(4)	L	$\vdash$	a	$(R')(2,1,3)$

hence (SeAt)

(1)	$\neg a$	$\vdash$	Δa	(Ad_2)
(2)	$\neg\Delta a$	$\vdash$	$\neg\Delta a$	(A)
(3)		$\vdash$	$\Delta\Delta a$	(D_0)
(4)	$\neg\Delta a\,,\,\neg a$	$\vdash$	a	$(X')(1,2,3)$
(5)	$\neg\Delta a$	$\vdash$	a	$(SeAt)(4)$

hence (D_1)

4.6.6 Remarks

1. Note that *contraposition* rules no longer hold. We have

$$\mathbf{PPC}' + (\text{SeAt}) + (\text{SeDe}) \Longrightarrow \begin{cases} (\text{CaPo}_1) \quad \dfrac{L, a \;\vdash\; b}{L, \neg b \;\vdash\; \neg a}; \\[2ex] (\text{CaPo}_2) \quad \dfrac{L, \neg a \;\vdash\; b}{L, \neg b \;\vdash\; a}; \end{cases}$$

$$\mathbf{PPC}' + (\text{CaPo}_1) \Longrightarrow (\text{D}_1)$$
$$\mathbf{PPC}' + (\text{CaPo}_2) \Longrightarrow (\text{D}_2)$$

2. Also note that the *deduction* rule no longer holds.

3. If for some reason we should apply (SeDe) in $\mathbf{PPC}''$, on a formula a that is not defined for a theory L, then the considered theory becomes inconsistent.

   Assume $L \vdash_{PPC''} \neg\Delta a$, and suppose we have shown $L, \neg a \vdash_{PPC''} a$.
   If we apply (SeDe), we get $L \vdash_{PPC''} a$ and hence by (Δ) $L \vdash_{PPC''} \Delta a$. From the contradiction rule we get $L \vdash_{PPC''} b$ for each formula b.

Let $\mathbf{PPC}^* = \mathbf{PPC}'' + (\text{CN}_0) + (\text{GN}_x)$, where

$$(\text{CN}_0) \quad \frac{\begin{array}{ll} L_1, \neg a_1 & \vdash\; b \\ L_2, \neg a_2 & \vdash\; b \end{array}}{L_{12}, \neg(a_1 \wedge a_2) \;\vdash\; b}; \qquad (\text{GN}_x) \quad \frac{L, \neg a \;\vdash\; b}{L, \neg\forall x\, a \;\vdash\; b};$$

then we have:

4.6.7 Theorem

PPC* and **BCJ** are equivalent.

Note that in **PPC*** we still have some modified versions of the rules (SeAt), (SeDe), (DeRu).

$$(\mathrm{SeAt}')\ \frac{\begin{array}{lll} L_1, \neg a & \vdash^* & a \\ L_2 & \vdash^* & \Delta a \end{array}}{\begin{array}{lll} L_{12} & \vdash^* & a \end{array}}; \qquad (\mathrm{SeDe}')\ \frac{\begin{array}{lll} L_1, a & \vdash^* & \neg a \\ L_2 & \vdash^* & \Delta a \end{array}}{\begin{array}{lll} L_{12} & \vdash^* & \neg a \end{array}};$$

$$(\mathrm{DeRu}')\frac{\begin{array}{lll} L & \vdash^* & a \to b \end{array}}{\begin{array}{lll} L, a & \vdash^* & b \end{array}};$$

$$(\mathrm{DeRu}'')\frac{\begin{array}{lll} L_1, a & \vdash^* & b \\ L_2 & \vdash^* & \Delta a \end{array}}{\begin{array}{lll} L_{12} & \vdash^* & a \to b \end{array}}; \qquad (\mathrm{DeR}''')\ \frac{\begin{array}{lll} L_1, a & \vdash^* & \Delta b \\ L_2 & \vdash^* & \Delta a \end{array}}{\begin{array}{lll} L_{12} & \vdash^* & \Delta(a \to b) \end{array}};$$

5 Applications of PPC or BCJ in computer science

5.1 Partially defined functions

There are a number of simple functions, for which it is not easy to describe the domain. We consider **TURBO5**-Pascal to illustrate this statement.

1. ```
   program undef1;
   uses crt;
   var x :integer;
       ch:char;
    function f(n:integer):integer;
   {============================
    The domain of this function consists of all odd numbers <= maxint
    The value of the function = maxint}
    var k : integer;
    begin{f}
      k:=0;
      while k<maxint do k:=k+n;
      f:=k;
    end{f};
   begin{undef1}
     clrscr;
     ch:='y';
     while (ch='y') or (ch='Y') do
       begin
        write('Give an integer: ');
        readln(x);
        writeln('f(',x,')=',f(x));
        write('Continue ? ');
        ch:=readkey;
        writeln;
       end
   end{undef1}.
   ```

2. ```
   program undef2;
   uses crt;
   var x :integer;
       ch:char;
    function f(n:integer):integer;
   {============================
    The domain of this function consists of the divisors of maxint
    For TURBO5 we get 1,7,31,217,1057,4681,32767
    The value of the function = maxint}
    var k : integer;
    begin{f}
      k:=0;
      while n*(k div n)<maxint do k:=k+n;
      f:=k;
    end{f};
   begin{undef2}
     clrscr;
     ch:='y';
     while (ch='y') or (ch='Y') do
       begin
        write('Give an integer: ');
        readln(x);
        writeln('f(',x,')=',f(x));
   ```

```
      write('Continue ? ');
      ch:=readkey;
      writeln;
     end
end{undef2}.
```

3.
```
program undef3;
uses crt;
var x :integer;
    ch:char;
 function f(n:integer):integer;
{=============================
 The domain of this function consists of all odd numbers <= maxint
 The function is NOT constant}
 var k : integer;
 begin{f}
   k:=0;
   while n*k<maxint do k:=k+n;
   f:=k;
 end{f};
begin{undef3}
  clrscr;
  ch:='y';
  while (ch='y') or (ch='Y') do
    begin
     write('Give an integer: ');
     readln(x);
     writeln('f(',x,')=',f(x));
     write('Continue ? ');
     ch:=readkey;
     writeln;
    end
end{undef3}.
```

4.
```
program undef4;
uses crt;
var x :integer;
    ch:char;
 function f(n:integer):integer;
{=============================}
 var k,c : integer;
 begin{f}
   k:=0;c:=1;
   while k<(maxint-c) do k:=k+n;
   f:=k;
 end{f};
begin{undef4}
  clrscr;
  ch:='y';
  while (ch='y') or (ch='Y') do
    begin
     write('Give an integer: ');
     readln(x);
     writeln('f(',x,')=',f(x));
     write('Continue ? ');
     ch:=readkey;
     writeln;
    end
end{undef4}.
```

5.
```
program undef5;
uses crt;
```

```
var x :integer;
    out:text;
 function f(n:integer):integer;
{=============================
 partial function, not defined for n:=8*t}
 var k,c : integer;
 begin{f}
   k:=0;c:=3;
   while k<(maxint-c) do k:=k+n;
   f:=k;
 end{f};
begin{undef5}
  clrscr;
  assign(out,'undef5.out');
  rewrite(out);
  x:=1;
  writeln(out,'undef5.out');
  writeln(out,'==========');
  writeln(out);
  while x<=480 do
    begin
      write(x:4);
      if x mod 8 <> 0 then write(out,x:4,f(x):6)
                      else writeln(out,x:4,' - ');
      x:=x+1;
    end{while};
  close(out);
end{undef5}.
```

undef5.out

```
   1 32764   2 32764   3 32766   4 32764   5 32765   6 32766   7 32767   8 -
   9 32764  10 32764  11 32764  12 32764  13 32767  14 32766  15 32767  16 -
  17 32767  18 32764  19 32765  20 32764  21 32765  22 32764  23 32766  24 -
  25 32764  26 32764  27 32764  28 32764  29 32765  30 32766  31 32767  32 -
  33 32764  34 32766  35 32766  36 32764  37 32764  38 32764  39 32767  40 -
  41 32764  42 32764  43 32766  44 32764  45 32767  46 32766  47 32767  48 -
  49 32764  50 32764  51 32767  52 32764  53 32767  54 32764  55 32766  56 -
  57 32767  58 32764  59 32764  60 32764  61 32764  62 32766  63 32767  64 -
  65 32764  66 32764  67 32767  68 32764  69 32764  70 32766  71 32766  72 -
  73 32767  74 32764  75 32765  76 32764  77 32766  78 32766  79 32764  80 -
  81 32767  82 32764  83 32767  84 32764  85 32767  86 32766  87 32767  88 -
  89 32767  90 32764  91 32766  92 32764  93 32765  94 32764  95 32764  96 -
  97 32766  98 32764  99 32764 100 32764 101 32766 102 32766 103 32764 104 -
 105 32764 106 32764 107 32766 108 32764 109 32764 110 32766 111 32764 112 -
 113 32765 114 32766 115 32766 116 32764 117 32765 118 32764 119 32764 120 -
 121 32767 122 32764 123 32764 124 32764 125 32767 126 32766 127 32766 128 -
 129 32766 130 32764 131 32767 132 32764 133 32767 134 32764 135 32765 136 -
 137 32767 138 32764 139 32767 140 32764 141 32765 142 32766 143 32764 144 -
 145 32765 146 32766 147 32764 148 32764 149 32765 150 32764 151 32767 152 -
 153 32767 154 32766 155 32767 156 32764 157 32764 158 32764 159 32764 160 -
 161 32767 162 32764 163 32766 164 32764 165 32767 166 32764 167 32766 168 -
 169 32766 170 32766 171 32767 172 32764 173 32767 174 32766 175 32764 176 -
 177 32767 178 32764 179 32766 180 32764 181 32767 182 32766 183 32764 184 -
 185 32764 186 32764 187 32767 188 32764 189 32767 190 32764 191 32767 192 -
 193 32764 194 32766 195 32766 196 32764 197 32767 198 32764 199 32765 200 -
 201 32764 202 32766 203 32767 204 32764 205 32767 206 32764 207 32764 208 -
 209 32764 210 32764 211 32764 212 32764 213 32766 214 32766 215 32767 216 -
 217 32767 218 32764 219 32767 220 32764 221 32764 222 32764 223 32766 224 -
 225 32767 226 32764 227 32767 228 32764 229 32766 230 32766 231 32764 232 -
 233 32764 234 32764 235 32766 236 32764 237 32764 238 32764 239 32766 240 -
 241 32767 242 32766 243 32764 244 32764 245 32766 246 32764 247 32765 248 -
 249 32765 250 32766 251 32764 252 32764 253 32764 254 32766 255 32767 256 -
 257 32767 258 32766 259 32767 260 32764 261 32766 262 32764 263 32764 264 -
 265 32764 266 32766 267 32765 268 32764 269 32766 270 32764 271 32765 272 -
 273 32764 274 32766 275 32767 276 32764 277 32767 278 32766 279 32767 280 -
 281 32765 282 32764 283 32767 284 32764 285 32765 286 32764 287 32766 288 -
 289 32765 290 32764 291 32766 292 32764 293 32766 294 32764 295 32764 296 -
 297 32766 298 32764 299 32766 300 32764 301 32767 302 32766 303 32767 304 -
 305 32766 306 32764 307 32767 308 32764 309 32764 310 32766 311 32764 312 -
 313 32764 314 32764 315 32766 316 32764 317 32764 318 32764 319 32767 320 -
 321 32764 322 32766 323 32764 324 32764 325 32767 326 32766 327 32764 328 -
 329 32765 330 32766 331 32764 332 32764 333 32764 334 32766 335 32767 336 -
 337 32767 338 32766 339 32767 340 32764 341 32767 342 32766 343 32767 344 -
 345 32766 346 32766 347 32767 348 32764 349 32764 350 32764 351 32764 352 -
 353 32767 354 32764 355 32764 356 32764 357 32767 358 32766 359 32764 360 -
 361 32764 362 32766 363 32765 364 32764 365 32766 366 32764 367 32767 368 -
 369 32764 370 32764 371 32764 372 32764 373 32764 374 32764 375 32765 376 -
 377 32764 378 32766 379 32766 380 32764 381 32766 382 32766 383 32766 384 -
 385 32766 386 32764 387 32764 388 32764 389 32765 390 32766 391 32767 392 -
 393 32765 394 32766 395 32764 396 32764 397 32765 398 32764 399 32767 400 -
 401 32764 402 32764 403 32767 404 32764 405 32767 406 32766 407 32764 408 -
 409 32765 410 32766 411 32767 412 32764 413 32764 414 32764 415 32764 416 -
```

```
417 32765 418 32764 419 32766 420 32764 421 32767 422 32764 423 32764 424 -
425 32766 426 32766 427 32767 428 32764 429 32764 430 32766 431 32767 432 -
433 32766 434 32766 435 32764 436 32764 437 32766 438 32766 439 32766 440 -
441 32766 442 32764 443 32764 444 32764 445 32767 446 32766 447 32767 448 -
449 32764 450 32764 451 32767 452 32764 453 32765 454 32764 455 32767 456 -
457 32767 458 32766 459 32764 460 32764 461 32765 462 32764 463 32767 464 -
465 32767 466 32764 467 32767 468 32764 469 32766 470 32766 471 32764 472 -
473 32767 474 32764 475 32764 476 32764 477 32764 478 32766 479 32767 480 -
```

It is easy to show that $\{8 * t\} \cap \texttt{dom} f = \{\ \}$

Proof:

The program will stop for $8 * t$ if

$$\begin{array}{lll} (1) & 8 * t = & (2^{15} - 1) \bmod 2^{16} \\ (2) & 8 * t = & (2^{15} - 2) \bmod 2^{16} \\ (3) & 8 * t = & (2^{15} - 3) \bmod 2^{16} \\ (4) & 8 * t = & (2^{15} - 4) \bmod 2^{16} \end{array}$$

Since the right-hand sides of (1) and (2) represent odd numbers, those cases are excluded.

For (2) we must have:

$8 * t = 2^{15} - 2 + k(2^{16})$ or

$8 * t + 2 = 2^{15}(1 + 2k)$ or

$4 * t + 1 = 2^{14}(1 + 2k)$ which is impossible.

For (4) we must have:

$8 * t = 2^{15} - 4 + k(2^{16})$ or

$8 * t + 4 = 2^{15}(1 + 2k)$ or

$2 * t + 1 = 2^{13}(1 + 2k)$ which is impossible.

6.
```
program undef6;
uses crt;
var x,c :integer;
    out:text;
 function f(n,c:integer):integer;
{================================
 partial function;
 domain of f depends on n and c
 eg. not defined for (c=1 and n:=4*t) or (c=3 and n=8*t}
 var k : integer;
 begin{f}
   k:=0;
   while k<(maxint-c) do k:=k+n;
   f:=k;
 end{f};
begin{undef6}
  clrscr;
  assign(out,'undef6.out');
  rewrite(out);
  x:=1;c:=1;
  writeln(out,'undef6.out for c:=1');
  writeln(out,'===================');
  writeln(out);
```

```
    while x<=480 do
      begin
        write(x:4);
        if x mod 4 <> 0 then write(out,x:4,f(x,c):6)
                        else write(out,x:4,' - ');
        if x mod 8 = 0 then writeln(out);
        x:=x+1;
      end{while};
    close(out);
  end{undef6}.
```

Concerning the this function $f(n, c)$, we note that

$$\forall x(x : \texttt{integer} \Longrightarrow -1 - \texttt{maxint} < x < +\texttt{maxint}$$

From the following testprogram we see that

$$\begin{array}{lcrcl}
\texttt{maxint} & = & 32767 & = & 2^{15} - 1 \\
1 + \texttt{maxint} & = & -32768 & = & -2^{15} \\
-1 - \texttt{maxint} & = & -32768 & & \\
2 * (1 + \texttt{maxint}) & = & 0 & &
\end{array}$$

```
program test;
uses crt;
 var x,y:integer;
begin
 clrscr;
 y:=maxint;
 writeln('   maxint: ',y);
 x:=1+y;
 writeln(' 1+maxint: ',x);
 x:=-1-y;
 writeln('-1-maxint: ',x);
 x:=2*(1+y);
 writeln('2*(1+maxint): ',x);
end.
```

We ran "`undef6`" for $c := 1, 4, 9, 27$ and got the following:

```
undef6.out_for_c:=1

   1 32766   2 32766   3 32766   4 -   5 32767   6 32766   7 32767   8 -
   9 32766  10 32766  11 32766  12 -  13 32767  14 32766  15 32767  16 -
  17 32767  18 32766  19 32767  20 -  21 32767  22 32766  23 32766  24 -
  25 32767  26 32766  27 32767  28 -  29 32767  30 32766  31 32767  32 -
  33 32766  34 32766  35 32766  36 -  37 32766  38 32766  39 32767  40 -
  41 32766  42 32766  43 32766  44 -  45 32767  46 32766  47 32767  48 -
  49 32767  50 32766  51 32767  52 -  53 32767  54 32766  55 32766  56 -
  57 32767  58 32766  59 32767  60 -  61 32766  62 32766  63 32767  64 -
  65 32766  66 32766  67 32767  68 -  69 32766  70 32766  71 32766  72 -
  73 32767  74 32766  75 32767  76 -  77 32766  78 32766  79 32766  80 -
  81 32767  82 32766  83 32767  84 -  85 32767  86 32766  87 32767  88 -
  89 32767  90 32766  91 32766  92 -  93 32767  94 32766  95 32767  96 -
  97 32766  98 32766  99 32766 100 - 101 32766 102 32766 103 32766 104 -
 105 32766 106 32766 107 32766 108 - 109 32766 110 32766 111 32767 112 -
 113 32767 114 32766 115 32766 116 - 117 32767 118 32766 119 32767 120 -
 121 32767 122 32766 123 32767 124 - 125 32767 126 32766 127 32766 128 -
 129 32766 130 32766 131 32767 132 - 133 32767 134 32766 135 32767 136 -
 137 32767 138 32766 139 32767 140 - 141 32767 142 32766 143 32766 144 -
 145 32767 146 32766 147 32766 148 - 149 32767 150 32766 151 32767 152 -
 153 32767 154 32766 155 32767 156 - 157 32766 158 32766 159 32766 160 -
 161 32767 162 32766 163 32766 164 - 165 32767 166 32766 167 32766 168 -
 169 32766 170 32766 171 32767 172 - 173 32767 174 32766 175 32767 176 -
 177 32767 178 32766 179 32766 180 - 181 32767 182 32766 183 32767 184 -
 185 32766 186 32766 187 32767 188 - 189 32767 190 32766 191 32767 192 -
 193 32766 194 32766 195 32766 196 - 197 32767 198 32766 199 32767 200 -
 201 32766 202 32766 203 32767 204 - 205 32767 206 32766 207 32766 208 -
```

209	32767	210	32766	211	32767	212	-	213	32766	214	32766	215	32767	216	-
217	32767	218	32766	219	32767	220	-	221	32767	222	32766	223	32766	224	-
225	32767	226	32766	227	32767	228	-	229	32766	230	32766	231	32766	232	-
233	32767	234	32766	235	32766	236	-	237	32767	238	32766	239	32766	240	-
241	32767	242	32766	243	32766	244	-	245	32766	246	32766	247	32767	248	-
249	32767	250	32766	251	32766	252	-	253	32766	254	32766	255	32767	256	-
257	32767	258	32766	259	32767	260	-	261	32766	262	32766	263	32766	264	-
265	32767	266	32766	267	32767	268	-	269	32766	270	32766	271	32767	272	-
273	32766	274	32766	275	32767	276	-	277	32767	278	32766	279	32767	280	-
281	32767	282	32766	283	32767	284	-	285	32767	286	32766	287	32766	288	-
289	32767	290	32766	291	32766	292	-	293	32766	294	32766	295	32767	296	-
297	32766	298	32766	299	32766	300	-	301	32767	302	32766	303	32767	304	-
305	32766	306	32766	307	32767	308	-	309	32766	310	32766	311	32766	312	-
313	32766	314	32766	315	32766	316	-	317	32767	318	32766	319	32767	320	-
321	32766	322	32766	323	32766	324	-	325	32767	326	32766	327	32767	328	-
329	32767	330	32766	331	32766	332	-	333	32766	334	32766	335	32767	336	-
337	32767	338	32766	339	32767	340	-	341	32767	342	32766	343	32767	344	-
345	32766	346	32766	347	32767	348	-	349	32767	350	32766	351	32766	352	-
353	32767	354	32766	355	32766	356	-	357	32767	358	32766	359	32766	360	-
361	32766	362	32766	363	32766	364	-	365	32766	366	32766	367	32767	368	-
369	32766	370	32766	371	32766	372	-	373	32766	374	32766	375	32766	376	-
377	32766	378	32766	379	32766	380	-	381	32766	382	32766	383	32766	384	-
385	32766	386	32766	387	32766	388	-	389	32767	390	32766	391	32767	392	-
393	32767	394	32766	395	32766	396	-	397	32766	398	32766	399	32767	400	-
401	32767	402	32766	403	32767	404	-	405	32767	406	32766	407	32766	408	-
409	32767	410	32766	411	32767	412	-	413	32766	414	32766	415	32767	416	-
417	32766	418	32766	419	32766	420	-	421	32767	422	32766	423	32766	424	-
425	32766	426	32766	427	32767	428	-	429	32766	430	32766	431	32767	432	-
433	32766	434	32766	435	32766	436	-	437	32766	438	32766	439	32766	440	-
441	32766	442	32766	443	32766	444	-	445	32767	446	32766	447	32767	448	-
449	32766	450	32766	451	32767	452	-	453	32767	454	32766	455	32767	456	-
457	32767	458	32766	459	32766	460	-	461	32767	462	32766	463	32767	464	-
465	32767	466	32766	467	32767	468	-	469	32766	470	32766	471	32766	472	-
473	32767	474	32766	475	32766	476	-	477	32767	478	32766	479	32767	480	-

undef64.out for c:=4
==========

1	32763	2	32764	3	32763	4	32764	5	32765	6	32766	7	32767	8	-
9	32764	10	32764	11	32764	12	32764	13	32767	14	32766	15	32767	16	-
17	32767	18	32764	19	32765	20	32764	21	32765	22	32764	23	32766	24	-
25	32764	26	32764	27	32764	28	32764	29	32765	30	32766	31	32767	32	-
33	32764	34	32766	35	32763	36	32764	37	32764	38	32764	39	32767	40	-
41	32764	42	32764	43	32766	44	32764	45	32767	46	32766	47	32763	48	-
49	32764	50	32764	51	32767	52	32764	53	32767	54	32764	55	32766	56	-
57	32767	58	32764	59	32764	60	32764	61	32764	62	32766	63	32767	64	-
65	32763	66	32764	67	32763	68	32764	69	32763	70	32766	71	32766	72	-
73	32767	74	32764	75	32765	76	32764	77	32766	78	32766	79	32763	80	-
81	32763	82	32764	83	32763	84	32764	85	32767	86	32766	87	32767	88	-
89	32763	90	32764	91	32766	92	32764	93	32765	94	32764	95	32764	96	-
97	32766	98	32764	99	32764	100	32764	101	32763	102	32766	103	32764	104	-
105	32764	106	32764	107	32763	108	32764	109	32763	110	32766	111	32764	112	-
113	32765	114	32766	115	32766	116	32764	117	32765	118	32764	119	32764	120	-
121	32767	122	32764	123	32764	124	32764	125	32767	126	32766	127	32766	128	-
129	32766	130	32764	131	32767	132	32764	133	32767	134	32764	135	32765	136	-
137	32767	138	32764	139	32767	140	32764	141	32765	142	32766	143	32764	144	-
145	32765	146	32766	147	32764	148	32764	149	32765	150	32764	151	32767	152	-
153	32767	154	32766	155	32763	156	32764	157	32764	158	32764	159	32764	160	-
161	32763	162	32764	163	32763	164	32764	165	32767	166	32764	167	32766	168	-
169	32766	170	32766	171	32767	172	32764	173	32767	174	32766	175	32764	176	-
177	32763	178	32764	179	32763	180	32764	181	32767	182	32766	183	32764	184	-
185	32763	186	32764	187	32763	188	32764	189	32767	190	32764	191	32767	192	-
193	32764	194	32766	195	32763	196	32764	197	32767	198	32764	199	32765	200	-
201	32763	202	32766	203	32767	204	32764	205	32767	206	32764	207	32763	208	-
209	32764	210	32764	211	32764	212	32764	213	32766	214	32766	215	32767	216	-
217	32767	218	32764	219	32767	220	32764	221	32764	222	32764	223	32766	224	-
225	32763	226	32764	227	32763	228	32764	229	32766	230	32766	231	32763	232	-
233	32764	234	32764	235	32766	236	32764	237	32764	238	32764	239	32766	240	-
241	32767	242	32766	243	32763	244	32764	245	32763	246	32764	247	32765	248	-
249	32765	250	32766	251	32763	252	32764	253	32764	254	32766	255	32767	256	-
257	32767	258	32766	259	32767	260	32764	261	32766	262	32764	263	32764	264	-
265	32764	266	32766	267	32765	268	32764	269	32766	270	32764	271	32765	272	-
273	32763	274	32766	275	32767	276	32764	277	32763	278	32766	279	32763	280	-
281	32765	282	32764	283	32767	284	32764	285	32765	286	32764	287	32766	288	-
289	32765	290	32764	291	32766	292	32764	293	32766	294	32764	295	32764	296	-
297	32766	298	32764	299	32766	300	32764	301	32767	302	32766	303	32763	304	-
305	32766	306	32764	307	32763	308	32764	309	32763	310	32766	311	32764	312	-
313	32764	314	32764	315	32766	316	32764	317	32764	318	32764	319	32767	320	-
321	32764	322	32766	323	32764	324	32764	325	32767	326	32766	327	32764	328	-
329	32765	330	32766	331	32764	332	32764	333	32764	334	32766	335	32767	336	-
337	32767	338	32766	339	32767	340	32764	341	32767	342	32766	343	32767	344	-
345	32766	346	32766	347	32763	348	32764	349	32764	350	32764	351	32764	352	-
353	32767	354	32764	355	32764	356	32764	357	32763	358	32766	359	32764	360	-
361	32764	362	32766	363	32765	364	32764	365	32766	366	32764	367	32767	368	-
369	32764	370	32764	371	32763	372	32764	373	32763	374	32764	375	32765	376	-
377	32764	378	32766	379	32766	380	32764	381	32766	382	32766	383	32766	384	-
385	32766	386	32764	387	32763	388	32764	389	32765	390	32766	391	32767	392	-
393	32765	394	32766	395	32764	396	32764	397	32765	398	32764	399	32767	400	-
401	32764	402	32764	403	32763	404	32764	405	32763	406	32766	407	32764	408	-
409	32765	410	32766	411	32767	412	32764	413	32764	414	32764	415	32764	416	-
417	32765	418	32764	419	32766	420	32764	421	32767	422	32764	423	32764	424	-
425	32766	426	32766	427	32767	428	32764	429	32763	430	32766	431	32767	432	-
433	32766	434	32766	435	32764	436	32764	437	32766	438	32766	439	32763	440	-
441	32766	442	32764	443	32764	444	32764	445	32763	446	32766	447	32767	448	-
449	32763	450	32764	451	32767	452	32764	453	32765	454	32764	455	32767	456	-
457	32767	458	32766	459	32764	460	32764	461	32765	462	32764	463	32767	464	-
465	32767	466	32764	467	32767	468	32764	469	32766	470	32766	471	32764	472	-
473	32767	474	32764	475	32763	476	32764	477	32764	478	32766	479	32767	480	-

undef69.out for c:=9
==========

1	32758	2	32758	3	32760	4	32760	5	32760	6	32760	7	32760	8	32760
9	32760	10	32760	11	32758	12	32760	13	32760	14	32760	15	32760	16	-
17	32759	18	32760	19	32765	20	32760	21	32760	22	32758	23	32766	24	32760
25	32764	26	32760	27	32764	28	32760	29	32761	30	32760	31	32767	32	-

33	32758	34	32758	35	32760	36	32760	37	32764	38	32760	39	32760	40	32760
41	32759	42	32760	43	32766	44	32760	45	32760	46	32766	47	32759	48	-
49	32758	50	32764	51	32767	52	32760	53	32767	54	32764	55	32766	56	32760
57	32760	58	32760	59	32758	60	32760	61	32764	62	32766	63	32760	64	-
65	32760	66	32758	67	32763	68	32764	69	32762	70	32760	71	32766	72	32760
73	32758	74	32764	75	32765	76	32760	77	32766	78	32760	79	32763	80	-
81	32763	82	32764	83	32759	84	32760	85	32767	86	32766	87	32761	88	32760
89	32763	90	32760	91	32760	92	32764	93	32765	94	32760	95	32764	96	-
97	32761	98	32758	99	32758	100	32764	101	32763	102	32766	103	32764	104	32760
105	32760	106	32764	107	32760	108	32764	109	32758	110	32766	111	32764	112	-
113	32761	114	32760	115	32758	116	32760	117	32760	118	32758	119	32758	120	32760
121	32762	122	32764	123	32764	124	32764	125	32767	126	32760	127	32766	128	-
129	32766	130	32760	131	32760	132	32760	133	32767	134	32760	135	32761	136	32760
137	32767	138	32762	139	32762	140	32760	141	32761	142	32766	143	32764	144	-
145	32761	146	32758	147	32764	148	32764	149	32765	150	32760	151	32767	152	32760
153	32767	154	32766	155	32763	156	32760	157	32758	158	32764	159	32764	160	-
161	32763	162	32760	163	32763	164	32764	165	32762	166	32760	167	32766	168	32760
169	32761	170	32766	171	32767	172	32764	173	32759	174	32760	175	32758	176	-
177	32763	178	32760	179	32760	180	32760	181	32761	182	32760	183	32764	184	32760
185	32758	186	32764	187	32759	188	32760	189	32767	190	32764	191	32760	192	-
193	32760	194	32758	195	32760	196	32764	197	32767	198	32758	199	32761	200	32760
201	32763	202	32766	203	32767	204	32764	205	32761	206	32764	207	32758	208	-
209	32758	210	32760	211	32758	212	32764	213	32766	214	32760	215	32758	216	32760
217	32767	218	32758	219	32767	220	32764	221	32758	222	32764	223	32766	224	-
225	32759	226	32760	227	32759	228	32760	229	32766	230	32758	231	32762	232	32760
233	32758	234	32760	235	32766	236	32764	237	32764	238	32758	239	32760	240	-
241	32759	242	32762	243	32763	244	32764	245	32760	246	32764	247	32761	248	32760
249	32761	250	32766	251	32758	252	32760	253	32764	254	32766	255	32767	256	-
257	32767	258	32766	259	32767	260	32760	261	32761	262	32760	263	32764	264	32760
265	32764	266	32766	267	32761	268	32760	269	32766	270	32760	271	32760	272	-
273	32760	274	32766	275	32759	276	32764	277	32759	278	32762	279	32759	280	32760
281	32765	282	32760	283	32767	284	32760	285	32765	286	32764	287	32766	288	-
289	32761	290	32760	291	32761	292	32764	293	32766	294	32764	295	32764	296	32760
297	32760	298	32760	299	32766	300	32760	301	32767	302	32766	303	32763	304	-
305	32759	306	32764	307	32763	308	32764	309	32758	310	32762	311	32758	312	32760
313	32764	314	32758	315	32760	316	32764	317	32764	318	32764	319	32760	320	-
321	32760	322	32762	323	32762	324	32760	325	32762	326	32766	327	32758	328	32760
329	32761	330	32762	331	32758	332	32760	333	32764	334	32766	335	32767	336	-
337	32762	338	32758	339	32761	340	32764	341	32767	342	32766	343	32767	344	32760
345	32766	346	32762	347	32759	348	32760	349	32761	350	32758	351	32759	352	-
353	32759	354	32760	355	32758	356	32760	357	32763	358	32760	359	32758	360	32760
361	32764	362	32760	363	32765	364	32760	365	32766	366	32764	367	32767	368	-
369	32764	370	32758	371	32758	372	32764	373	32758	374	32760	375	32765	376	32760
377	32764	378	32766	379	32766	380	32764	381	32766	382	32760	383	32766	384	-
385	32766	386	32760	387	32762	388	32760	389	32760	390	32760	391	32767	392	32760
393	32761	394	32766	395	32764	396	32760	397	32765	398	32760	399	32767	400	-
401	32764	402	32764	403	32759	404	32764	405	32759	406	32762	407	32760	408	32760
409	32761	410	32760	411	32767	412	32764	413	32758	414	32758	415	32764	416	-
417	32765	418	32758	419	32766	420	32760	421	32767	422	32758	423	32764	424	32760
425	32766	426	32766	427	32759	428	32760	429	32758	430	32758	431	32767	432	-
433	32766	434	32766	435	32764	436	32760	437	32758	438	32766	439	32760	440	32760
441	32766	442	32758	443	32759	444	32764	445	32759	446	32766	447	32760	448	-
449	32760	450	32760	451	32767	452	32760	453	32765	454	32760	455	32760	456	32760
457	32762	458	32766	459	32764	460	32760	461	32765	462	32762	463	32767	464	-
465	32767	466	32758	467	32767	468	32760	469	32760	470	32766	471	32758	472	32760
473	32760	474	32764	475	32762	476	32764	477	32764	478	32760	479	32767	480	-

undef627.out_for_c:=27

1	32740	2	32740	3	32742	4	32740	5	32740	6	32742	7	32746	8	32744
9	32742	10	32740	11	32747	12	32748	13	32747	14	32746	15	32745	16	32752
17	32742	18	32742	19	32756	20	32740	21	32760	22	32758	23	32752	24	32760
25	32750	26	32760	27	32751	28	32760	29	32741	30	32760	31	32767	32	-
33	32740	34	32742	35	32760	36	32760	37	32745	38	32756	39	32760	40	32760
41	32759	42	32760	43	32766	44	32760	45	32760	46	32752	47	32759	48	32752
49	32758	50	32750	51	32742	52	32760	53	32754	54	32744	55	32749	56	32760
57	32746	58	32740	59	32745	60	32760	61	32757	62	32766	63	32760	64	-
65	32760	66	32740	67	32763	68	32740	69	32748	70	32760	71	32766	72	32760
73	32740	74	32764	75	32742	76	32756	77	32766	78	32760	79	32740	80	32752
81	32763	82	32764	83	32755	84	32760	85	32767	86	32766	87	32749	88	32760
89	32752	90	32760	91	32760	92	32752	93	32765	94	32752	95	32750	96	-
97	32761	98	32758	99	32740	100	32764	101	32750	102	32742	103	32754	104	32760
105	32760	106	32754	107	32742	108	32744	109	32755	110	32742	111	32745	112	32752
113	32741	114	32746	115	32744	116	32740	117	32760	118	32758	119	32758	120	32760
121	32762	122	32752	123	32741	124	32764	125	32750	126	32760	127	32766	128	-
129	32766	130	32760	131	32750	132	32740	133	32751	134	32760	135	32744	136	32752
137	32743	138	32748	139	32742	140	32760	141	32741	142	32766	143	32747	144	32752
145	32741	146	32740	147	32764	148	32764	149	32751	150	32742	151	32767	152	32760
153	32742	154	32766	155	32763	156	32760	157	32746	158	32740	159	32754	160	-
161	32763	162	32756	163	32763	164	32764	165	32742	166	32740	167	32755	168	32760
169	32761	170	32766	171	32746	172	32764	173	32759	174	32748	175	32740	176	32752
177	32745	178	32752	179	32757	180	32760	181	32761	182	32760	183	32757	184	32752
185	32745	186	32764	187	32740	188	32752	189	32744	190	32750	191	32760	192	-
193	32760	194	32758	195	32760	196	32764	197	32767	198	32740	199	32741	200	32760
201	32763	202	32750	203	32749	204	32740	205	32761	206	32754	207	32748	208	32752
209	32755	210	32760	211	32749	212	32748	213	32766	214	32742	215	32758	216	32744
217	32767	218	32756	219	32740	220	32752	221	32752	222	32764	223	32740	224	-
225	32751	226	32740	227	32755	228	32760	229	32747	230	32744	231	32748	232	32744
233	32752	234	32760	235	32752	236	32764	237	32764	238	32758	239	32743	240	32752
241	32743	242	32762	243	32763	244	32752	245	32742	246	32764	247	32741	248	32760
249	32741	250	32750	251	32756	252	32760	253	32764	254	32766	255	32767	256	-
257	32767	258	32766	259	32767	260	32760	261	32761	262	32750	263	32746	264	32744
265	32742	266	32750	267	32745	268	32760	269	32749	270	32744	271	32750	272	32752
273	32760	274	32754	275	32742	276	32748	277	32748	278	32742	279	32759	280	32760
281	32751	282	32740	283	32742	284	32760	285	32742	286	32764	287	32766	288	-
289	32749	290	32740	291	32761	292	32740	293	32766	294	32764	295	32745	296	32760
297	32742	298	32746	299	32749	300	32756	301	32754	302	32766	303	32763	304	32752
305	32752	306	32742	307	32755	308	32748	309	32754	310	32762	311	32740	312	32760
313	32746	314	32746	315	32760	316	32740	317	32749	318	32754	319	32753	320	-
321	32742	322	32754	323	32755	324	32756	325	32745	326	32766	327	32755	328	32760
329	32741	330	32742	331	32740	332	32740	333	32764	334	32752	335	32743	336	32752
337	32752	338	32758	339	32749	340	32764	341	32757	342	32746	343	32767	344	32760
345	32752	346	32754	347	32759	348	32748	349	32761	350	32740	351	32744	352	-
353	32746	354	32756	355	32754	356	32752	357	32763	358	32760	359	32756	360	32760

```
361 32764 362 32760 363 32750 364 32760 365 32755 366 32764 367 32767 368 32752
369 32764 370 32746 371 32758 372 32764 373 32750 374 32740 375 32765 376 32752
377 32764 378 32744 379 32749 380 32764 381 32766 382 32760 383 32766 384 -
385 32766 386 32760 387 32762 388 32744 389 32760 390 32760 391 32757 392 32744
393 32757 394 32766 395 32764 396 32740 397 32765 398 32740 399 32767 400 32752
401 32764 402 32764 403 32755 404 32740 405 32743 406 32744 407 32760 408 32760
409 32745 410 32748 411 32767 412 32764 413 32758 414 32748 415 32744 416 -
417 32742 418 32758 419 32757 420 32760 421 32767 422 32752 423 32745 424 32752
425 32766 426 32766 427 32742 428 32748 429 32756 430 32758 431 32756 432 32752
433 32755 434 32766 435 32745 436 32756 437 32744 438 32740 439 32742 440 32752
441 32756 442 32752 443 32759 444 32764 445 32743 446 32740 447 32746 448 -
449 32742 450 32740 451 32753 452 32740 453 32765 454 32744 455 32760 456 32760
457 32762 458 32766 459 32764 460 32744 461 32748 462 32748 463 32744 464 32752
465 32753 466 32752 467 32767 468 32760 469 32760 470 32752 471 32740 472 32744
473 32746 474 32764 475 32748 476 32764 477 32745 478 32760 479 32746 480 -
```

We get the following table:

c			x
0			$2 * t$
1		2	$4 * t$
3	...	6	$8 * t$
7	...	14	$16 * t$
14	...	30	$32 * t$
31	...	62	$64 * t$
	...		...
1023	...	2046	$2048 * t$
$(2^k - 1)$	...	$(2^{k+1} - 2)$	$2^{k+1} * t$

5.2 Guards in TURBO5 with {$–B}

In **3.2** we have seen that

$$a\,\texttt{OR}\,b \quad \equiv \quad (\neg a \vee a) \wedge (a \vee b)$$

Show:

$$\vdash \quad \forall x\ (:\ \textit{real}) \quad (x = 0\ \texttt{OR}\ x/x = 1)$$
$$\vdash \quad \forall x\ (:\ \textit{real}) \quad ((\neg(x = 0) \vee x = 0) \wedge (x = 0 \vee x/x = 1))$$

Proof:

$\vdash \forall x((\neg(x = 0) \vee x = 0) \wedge (x = 0 \vee x/x = 1))$	*goal*
1. $\vdash \neg(x = 0) \vee x = 0$	*(reals)*
2. $\vdash x = 0 \vee x/x = 1$	*subgoal*
2.1 $\vdash \neg(x = 0) \vee x = 0$	*(reals)*
2.2 $x = 0 \vdash x = 0 \vee x/x = 1$	($\vee$-I)
2.3 $\neg(x = 0) \vdash x = 0 \vee x/x = 1$	*subgoal*
2.3.1 $x/x = 1$	*pr (2.3)&(reals)*
2.3.2 $x = 0 \vee x/x = 1$	($\vee$-I)(2.3.1)
2.4 $x = 0 \vee x/x = 1$	($\vee$-E)(2.1, 2.3, 2.2)
3. $\vdash (\neg(x = 0) \vee x = 0) \wedge (x = 0 \vee x/x = 1)$	($\wedge$-I)(1., 2.)
4. $\vdash \forall x((x = 0 \vee x = 0) \wedge (\neg(x = 0) \vee x/x = 1))$	($\forall$-I)(3.)

5.3 Program Correctness

5.3.1 An example

Consider the following pascal code,

```
function subp(i,j:integer):integer;
begin
   if i=j then subp := 0
          else subp := subp(i+1,j)+1
end;
```

We want to show that

$$\forall i \forall j((i \geq 0 \wedge j \geq 0 \wedge j \geq i) \rightarrow \texttt{subp}(i, j) = j - i)$$

or that over the *natural numbers*

$$\forall i, j(j \geq i \rightarrow \texttt{subp}(i, j) = j - i)\ (*)$$

5.3.2 Axioms for the natural numbers

In the first place we need some specific properities about the *natural numbers*, i.e. some of the axioms that characterize the *natural numbers*, such as:

induction

$$\text{(induct)} \quad \frac{\begin{array}{rcl} & \vdash & a(0/x) \\ x \geq 0, a(x) & \vdash & a(x+1/x) \end{array}}{\forall x(x \geq 0 \rightarrow a(x))}$$

5.3.3 The implementation

Furthermore we need some preliminary properties concerning

$\texttt{subp}$

that follow from the implementation.

$$(\mathrm{d}_1) \quad \frac{}{\texttt{subp}(n,n) = 0}; \qquad (\mathrm{d}_2) \quad \frac{n_1 \neq n_2, \texttt{subp}(n_1+1, n_2) = n_3}{\texttt{subp}(n_1, n_2) = n_3 + 1}$$

In order to prove (d_1) and (d_2) we have to know that in view of the *recursion* used in the implementation, $\texttt{subp}$ must be considered as the **least fixed point** of a function F defined as follows:

$\texttt{subp} = \mu F$

$F = \lambda f.\lambda i, j \ \texttt{if}\ i = j\ \texttt{then}\ 0\ \texttt{else}\ f(i+1, j) + 1$

We simplify the definition of $\texttt{subp}$, using the following notation:

$\texttt{subp}(i, j) \stackrel{\text{def}}{=} \texttt{if}\ i = j\ \texttt{then}\ 0\ \texttt{else}\ f(i+1, j) + 1$

The operational interpretation of this definition results in the interpretation of $\stackrel{\text{def}}{=}$ as *strong equality*

$(==\text{def})\ s_1 == s_2 \dashv\vdash (s_1 = s_2 \wedge \Delta s_1 \wedge \Delta s_2) \vee (\neg\Delta s_1 \wedge \neg\Delta s_2)$

We say that two terms are "*strongly equal*" iff they are "*defined*" or "*not defined*" simultaneously and if they are "*defined*", then they are "*equal*"

From *strong equality* we get *equality*, but not vice versa.

We have:

$$(==\rightarrow=_1)\ \frac{s_1 == s_2; s_1 = s_1}{s_1 = s_2} \qquad (==\rightarrow=_2)\ \frac{s_1 == s_2; s_2 = s_2}{s_1 = s_2}$$

5.3.4 Hoare-Floyd Logic

Furthermore we need some rules concerning basic *program statements.*

Whenever we are sure that the program will never abort, "*Hoare-Floyd Logic*" may handle the program. This logic is based on 2-valued predicate calculus, to which expressions of the form

$\{p\}\, S\, \{q\}$

are added, where p and q are formulas from the predicate logic, and S is a sequence of programming statements, i.e. a program.

One of those rules is called "*composition rule*":

$$\frac{\{p\}\ S_1\{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1; S_2\ \{q\}}$$

This means that whenever p is **true** and S_1 is executed, then r is **true**, and whenever r is **true** and S_2 is executed, then q is **true**, if then p is **true** and the composed program $S_1; S_2$ is executed, q will be **true**.

However, if we consider two large programs S_1 and S_2 such that the composition $S_1; S_2$ hardly fits into the memory of the computer (e.g. a PC where there is no swapping space available), then it is clear that if some part of S_1 or S_2 uses recursion, *stack overflow* may occur and $S_1; S_2$ will abort.

Andrzej Blikle from Warsaw (Poland), has done some research on this topic and in *MetaSoft Primer* (LNCS 288), he describes a system based on **PPC***.

However the job is not yet finished. To illustrate some of the problems, we go on with the proof about `subp`, where we need to specify how that function will be evaluated. *Blikle* has no clear answer for this problem.

5.3.5 Extending Hoare-Floyd

We may extend Hoare-Floyd by allowing predicates over **BCJ**. For our proof we need the following rules concerning the *program statement* `if ... then ... else`:

$$(\texttt{if-then}\text{–E})\ \frac{p, (\texttt{if}\ p\ \texttt{then}\ s_1\ \texttt{else}\ s_2) == s}{s_1 == s};$$

$$(\texttt{if-else}\text{–E})\ \frac{\neg p, (\texttt{if}\ p\ \texttt{then}\ s_1\ \texttt{else}\ s_2) == s}{s_2 == s};$$

Then we are able to prove (d_1) and (d_2).

(d_1)	$\vdash \texttt{subp}(n,n) = 0$	*goal*
1.	$\texttt{subp}(n,n) == \texttt{if}\ n = n\ \texttt{then}\ 0\ \textit{else}\ \texttt{subp}(n+1,n)+1$	*def of* `subp`
2.	$n = n$	=-var
3.	$\texttt{subp}(n,n) == 0$	if-then–E(2., 1.)
4.	$0 = 0$	= -cons
5.	$\texttt{subp}(n,n) = 0$	$(==\rightarrow=_2)(3., 4.)$

(d_2)	$n_1 \neq n_2, \mathtt{subp}(n_1+1, n_2) = n_3 \vdash \mathtt{subp}(n_1, n_2) = n_3 + 1$	*goal*
1.	$\mathtt{subp}(n_1, n_2) == \mathtt{if}\ n_1 = n_2\ \mathtt{then}\ 0\ \mathit{else}\ \mathtt{subp}(n_1+1, n_2) + 1$	*def of* subp
2.	$n_1 \neq n_2$	*pr*
3.	$\neg(n_1 = n_2)$	*notation*
4.	$\mathtt{subp}(n_1, n_2) == \mathtt{subp}(n_1+1, n_2) + 1$	if-else–E(3., 1.)
5.	$\mathtt{subp}(n_1+1, n_2) = n_3$	*pr*
6.	$\mathtt{subp}(n_1, n_2) == n_3 + 1$	=-subst(5, 4)
7.	$n_3 = n_3$	=-var
8.	$1 = 1$	=-const
9.	$n_3 + 1 = n_3 + 1$	*(integers)*(7., 8.)
10.	$\mathtt{subp}(n_1, n_2) = n_3 + 1$	$(==\rightarrow=_2)$(9., 6.)

Now we are able to show the property $(*)$

	$\vdash \forall i, j\ (j \geq i \rightarrow \mathtt{subp}(i, j) = j - i)$	*goal*
1.	$\vdash \forall k\ (k \geq 0 \rightarrow \mathtt{subp}(j-k, j) = k)$	*subgoal*
1.1	$\mathtt{subp}(j-0, j) = 0$	*subsubgoal*
1.1.1	$\mathtt{subp}(j, j) = 0$	d_1
1.1.2	$j = j - 0$	*(integers)*
1.1.3	$\mathtt{subp}(j-0, j) = 0$	=-subst(−1, −2)
1.2	$k \geq 0, \mathtt{subp}(j-k, j) = k \vdash \mathtt{subp}(j-(k+1), j) = k+1$	*subsubgoal*
1.2.1	$k \geq 0$	*pr*(1.2)
1.2.2	$k + 1 > 0$	*(integers)*(−1)
1.2.3	$j - (k+1) \neq j$	*(integers)*(−1)
1.2.4	$\mathtt{subp}(j-k, j) = k$	*pr*(1.2)
1.2.5	$j - k = j - (k+1) + 1$	*(integers)*
1.2.6	$\mathtt{subp}(j-(k+1)+1, j) = k$	=-subst(−1, −2)
1.2.7	$\mathtt{subp}(j-(k+1), j) = k+1$	d_2(−1, −4)
1.3	$\vdash \forall k\ (k \geq 0 \rightarrow \mathtt{subp}(j-k, j) = k)$	induct(1.1, 1.2)
2.	$j - i \geq 0 \rightarrow \mathtt{subp}(j-(j-i), j) = j - i$	∀-E(1.)
3.	$j - i \geq 0 \rightarrow \mathtt{subp}(i, j) = j - i$	*(integers)*(2.)
4.	$j \geq i \rightarrow \mathtt{subp}(i, j) = j - i$	*(integers)*(3.)
5.	$\forall j\ (j \geq i \rightarrow \mathtt{subp}(i), j) = j - i)$	∀-I(3.)
6.	$\forall i \forall j\ (j \geq i \rightarrow \mathtt{subp}(i), j) = j - i)$	∀-I(4.)

So for this example everything seems to be ok. Let us take however the following TURBO5 implementation, and check the performance of subp.

```
program subptest;
uses crt;
var
  j:integer;
function subp(i,j:integer):integer;
  begin
    if i=j then subp:=0
```

```
          else subp:=subp(i+1,j)+1
  end;
begin
  clrscr;
  for j:=1550 to maxint do
      begin
        write('subp(1,',j,')=',subp(1,j),'  ');
        if ((j-1549) mod 3) = 0 then writeln
      end;
end.
```

We get the following result:

```
subp(1,1550)=1549  subp(1,1551)=1550  subp(1,1552)=1551
subp(1,1553)=1552  subp(1,1554)=1553  subp(1,1555)=1554
subp(1,1556)=1555  subp(1,1557)=1556  subp(1,1558)=1557
subp(1,1559)=1558  subp(1,1560)=1559  subp(1,1561)=1560
subp(1,1562)=1561  subp(1,1563)=1562  subp(1,1564)=1563
subp(1,1565)=1564  subp(1,1566)=1565  subp(1,1567)=1566
subp(1,1568)=1567  subp(1,1569)=1568  subp(1,1570)=1569
subp(1,1571)=1570  subp(1,1572)=1571  subp(1,1573)=1572
subp(1,1574)=1573  subp(1,1575)=1574  subp(1,1576)=1575
subp(1,1577)=1576  subp(1,1578)=1577  subp(1,1579)=1578
subp(1,1580)=1579  subp(1,1581)=1580  subp(1,1582)=1581
subp(1,1583)=1582  subp(1,1584)=1583  subp(1,1585)=1584
subp(1,1586)=1585  subp(1,1587)=Runtime error 202 at 1DF2:000B.
Error 202: Stack overflow error
```

That we didn't noticed this problem during our proof, follows from the fact that in the application of the *induction principle* for the proof of

$$\vdash \forall k\,(k \geq 0 \rightarrow \mathtt{subp}(j-k, j) = k)$$

we didn't took into account that on line 1.2.7, the use of (d_2) depends on the limitations of the stack. Since `subp` uses recursion in its implementation, each use of (d_2) adds one more element to the stack, which connot go on for ever.

5.4 Presentation of data structures

From the previous example it became clear that a lot of those proofs will involve the internal representation of *procedures* and *data structures*

The next example especially concerns *data structures*. Much more details can be found in *C.B. Jones* Systematic Program Development. Talk CWI Amsterdam 1983. This paper deals with proofs about algorithms which use binary trees to represent mappings from **Keys** to **Data**. An interesting feature of those proofs is the use of *structural induction*

Here a highly simplified problem is considered only to illustrate such induction over tree-like objects.

Supose that we have

$$\texttt{Tree} = \texttt{Node} \cup \texttt{Data}$$

Then consider the **VDM**[1]-function

$$\texttt{mk-Node}(l, r)$$

(say mark node (l, r), with the trees l and r)

$$\texttt{Node} = \{\texttt{mk-Node}(l, r) \mid l, r \in \texttt{Tree}\}$$

Two selector functions L, R are available, such that

$$\begin{array}{lll} L: & \texttt{Node} \mapsto \texttt{Tree} & L(\texttt{mk-Node}(l, r)) = l \\ R: & \texttt{Node} \mapsto \texttt{Tree} & R(\texttt{mk-Node}(l, r)) = r \end{array}$$

Then we may consider the following axiom:

structural induction

$$\text{(struct induct)} \quad \frac{\begin{array}{lll} t \in \texttt{Data} & \vdash & p(t) \\ t \in \texttt{Node}, p(L(t)), p(R(t)) & \vdash & p(t) \end{array}}{\begin{array}{lll} t \in \texttt{Tree} & \vdash & p(t) \end{array}}$$

Consider the function

$$\begin{array}{lll} \mathit{collect}(t) & \stackrel{\text{def}}{=} & \texttt{if } t \in \texttt{Data} \quad \texttt{then } \{t\} \\ & & \texttt{else } \mathit{collect}(L(t)) \cup \mathit{collect}(R(t)) \end{array}$$

which is defined over `Tree`, and takes values in <u>*set of*</u>`Data`. Note that this is a non-trivial application of the many-sorted version of **BCJ**.

Then we have the following axioms:

$$(\mathrm{d}_3) \quad \frac{t \in \texttt{Data}}{\mathit{collect}(t) = \{t\}}$$

$$(\mathrm{d}_4) \quad \frac{t \in \texttt{Node}, \mathit{collect}(L(t)) = s_1, \mathit{collect}(R(t)) = s_2}{\mathit{collect}(t) = s_1 \cup s_2}$$

[1] **VDM** = **V**ienna **D**evelopment **M**ethod

We can show the following propery about `Trees`:

$t \in \texttt{Tree} \vdash \mathit{collect}(t) \neq \{\}$

Proof:

	$t \in \texttt{Tree} \vdash \mathit{collect}(t) \neq \{\}$	*goal*
1.	$t \in \texttt{Data} \vdash \mathit{collect}(t) \neq \{\}$	*subgoal*
	1.1 $\mathit{collect}(t) = \{t\}$	$pr(1.)(\mathrm{d}_3)$
	1.2 $\mathit{collect}(t) \neq \{\}$	*(sets)*(-1)
2.	$t \in \texttt{Node}, \mathit{collect}(L(t)) \neq \{\}, \mathit{collect}(R(t)) \neq \{\}$ $\vdash \mathit{collect}(t) \neq \{\}$	*subgoal*
	2.1 $\mathit{collect}(t) = \mathit{collect}(L(t)) \cup \mathit{collect}(R(t))$	$pr(2.)(\mathrm{d}_4)$
	2.2 $\mathit{collect}(t) \neq \{\}$	*(sets)*$(-1), pr(2.)$
3.	$\mathit{collect}(t) \neq \{\}$	*(struct induct)*,$pr(1.)(2.)$

5.5 Operations on datastructures

5.5.1 Introduction

We note that this way of describing datatypes, as used in **VDM**, is called the "*model theoretic*" approach. It deviates at this point from an other widely accepted "*property oriented*" style, which is more *algebraic axiomatic*. So if a relational database system is to be specified in **VDM**, the state might be modelled using (among other things) a mapping from relation names to sets of tuples.

The problem of representing mappings from *keys* to *data* occurs over and over again in computer systems. The problem of *partial operations* and even *non-determinism* can not be avoided.

In the model oriented specification one uses so-called *pre-conditions* to indicate the partial operations, and in general non-determinism is handled by (relational) *post-conditions*. For more details we refer to *Barringer et all* and *Blikle*.

5.5.2 Specification

For this example, we use the **VDM**-notation. In most cases the have an obvious meaning.

The top-level specification of the mapping from `Keys` to `Data` is made trivial, by the use of a class of objects called

$\texttt{Mkd} = \underline{\mathit{map}}\ \texttt{Key}\ \underline{\mathit{to}}\ \texttt{Data}$

The initial object in `Mkd` is

$m_0 = []$

The following operations can be defined:

INSERT(K:`Key`,D:`Data`)

$\underline{ext}$	$\underline{wr}\ M : \texttt{Mkd}$	external write
$\underline{pre}$	$k \notin \underline{dom}\ m$	precondition, m value for M
$\underline{post}$	$m = \overleftarrow{m} \cup [k \mapsto d]$	postcondition
		k value for K, d value for D
		$\overleftarrow{m}$ value at the beginning

FIND(K:`Key`) D:`Data`

$\underline{ext}$	$\underline{rd}\ M : \texttt{Mkd}$	external read
$\underline{pre}$	$k \in \underline{dom}\ m$	
$\underline{post}$	$d = \overleftarrow{m}\ (k)$	

DELETE(K:`Key`)

$\underline{ext}$	$\underline{wr}\ M : \texttt{Mkd}$
$\underline{pre}$	$k \in \underline{dom}\ m$
$\underline{post}$	$m = \overleftarrow{m} - \{k\}$

Note that each of these operations have been specified to be partial, in order to illustrate the proof rules. Although the operations are deterministic, relational post-conditions have been given. The experience with **VDM** suggests that *non- determinism* often arises during design.

5.5.3 Representation

This section is concerned with showing that the behaviour of one data type, models the behaviour of another.

Each representation is chosen so that it is closer to the final implementation, or so that it can be manipulated efficiently. In this way mappings can be represented as binary trees:

$\texttt{Bintree} = [\texttt{Binnode}]$

$\texttt{Binnode} :: \texttt{Bintree Key Data Bintree}^{(*)}$

$$\mathit{invBinnode}(\mathit{mk\text{-}Binnode}(lt, mk, md, rt)) \stackrel{\text{def}}{=}$$

$$(\forall\, lk \in \mathit{collkeys}(lt)\ \ lk < mk) \wedge (\forall\, rk \in \mathit{collkeys}(rt)\ \ mk < rk)$$

represents an *invariant*

$\mathit{inital}\ (\texttt{Bintree})\ \mathit{object} :\ t_0 = \texttt{nil}$

The set of objects, satisfying the definition $^{(*)}$ is given as:

$$\texttt{Binnode} = \{\mathit{mk\text{-}Binnode}(lt, mk, md, rt) \mid lt, rt \in \texttt{Bintree} \ \wedge\ mk \in \texttt{Key} \ \wedge\ md \in \texttt{Data}\}$$

Note that the use of the constructor function *mk-Binnode* as a parameter of the data type invariant, provides a way of naming the values of the sub-fields of a constructed object.

The function which collects the keys is defined as

$$\mathit{collkeys} : \texttt{Bintree} \mapsto \underline{\mathit{set\ of}}\ \texttt{Key}$$

$$\mathit{collkeys}(t) \stackrel{\text{def}}{=}$$

$$\begin{array}{lll} \mathit{cases}\ t: & \texttt{nil} & \mapsto \{\} \\ & \mathit{mk\text{-}Binmode}(lt, mk, md, rt) & \mapsto \mathit{collkeys}(lt) \cup \{mk\} \cup \mathit{collkeys}(rt) \end{array}$$

From now on "`Binnode`" (and thus "`Bintree`") is taken to be the set of objects which satisfy the invariant; these are called the *valid* objects.

Lemma(*collkeys1*)

$$\forall t \in \texttt{Bintree}\ \mathit{collkeys}(t) \in \underline{\mathit{set\ of}}\ \texttt{Key}$$

This lemma says that the function *collkeys* is *total*.

Lemma(*collkeys2*)

$$\begin{array}{l} \forall nd \in \texttt{Binnode} \\ \quad [\underline{\mathit{let}}\ \mathit{mk\text{-}Binnode}(lt, mk, md, rt)\ \underline{\mathit{in}} \\ \qquad \mathit{is\text{-}disj}(\mathit{collkeys}(lt), \{mk\}) \wedge \\ \qquad \mathit{is\text{-}disj}(\mathit{collkeys}(lt), \mathit{collkeys}(rt)) \wedge \\ \qquad \mathit{is\text{-}disj}(\{mk\}, \mathit{collkeys}(rt)) \wedge \\ \qquad \forall k \in \mathit{collkeys}(nd)((k < mk \rightarrow k \in \mathit{collkeys}(lt)) \wedge (mk < k \rightarrow k \in \mathit{collkeys}(rt)))] \end{array}$$

This expresses some trivial facts in a formal way.

The next step in developing the `Bintree`-*theory*, is the definition of a function to locate a given `Key` in a `Tree`.

$$\mathit{findb}(K : \texttt{Key}, T : \texttt{Bintree})D : \texttt{Data}$$

$$\underline{\mathit{pre}}\ k \in \mathit{collkeys}(t)$$

$$\mathit{findb}(k, \mathit{mk\text{-}Binnode}(lt, mk, md, rt)) \stackrel{\text{def}}{=}$$

$$\begin{array}{llll} \underline{\mathit{if}} & k = mk & \underline{\mathit{then}} & \mathit{md} \\ \underline{\mathit{else\ if}} & k < mk & \underline{\mathit{then}} & \mathit{findb}(lt, k) \\ \underline{\mathit{else}} & & & \mathit{findb}(rt, k) \end{array}$$

This function is not total, but it becomes total if we consider its restriction to the appropriate cartesian product of `Key` and `Bintree`, imposed by the precondition. This is expressed in the following:

Lemma(*findb1*)

$\forall k \in \texttt{Key}, t \in \texttt{Bintree}\ (k \in \mathit{collkeys}(t) \rightarrow \mathit{findb}(k,t) \in \texttt{Data})$

We note that the proof uses structural induction which can be stated as:

$$\frac{\begin{array}{l} p(\texttt{nil}), \\ t = \mathit{mk\text{-}Binnode}(lt, mk.md, rt), \mathit{invBinnode}(t), p(lt), p(rt) \ \vdash \ p(t) \end{array}}{t \in \texttt{Bintree} \ \vdash \ p(t)}$$

for the proof of the lemma, one uses the abbreviation

$p(k,t) := k \notin \mathit{collkeys}(t) \vee \mathit{findb}(k,t) \in \texttt{Data}$

Proof:

$t \in \texttt{Bintree}, k \in \texttt{Key} \vdash p(k,t)$		*goal*
1.	$k \notin \mathit{collkeys}(\texttt{nil})$	*collkeys*, (*sets*)
2.	$p(k, \texttt{nil})$	$\vee$-I(1.), p
3.	$t = \mathit{mk\text{-}Binnode}(lt, mk, md, rt) \wedge$	
	$\mathit{invBinnode}(t) \wedge p(k, lt) \wedge p(k, rt) \vdash p(k,t)$	*subgoal*
3.1	$k \in \mathit{collkeys}(t) \vee k \notin \mathit{collkeys}(t)$	*collkeys1*, (*sets*)
3.2	$k \notin \mathit{collkeys}(t) \vdash p(k,t)$	$\vee$-I, p
3.3	$k \in \mathit{collkeys}(t) \vdash p(k,t)$	*subsubgoal*
3.3.1	$k < mk \vee k = mk \vee mk < k$	*collkeys2*,*pr*(3.3), Key
3.3.2	$k = mk \vdash \mathit{findb}(k,t) \in \texttt{Data}$	*subsubsubgoal*
3.3.2.1	$\mathit{findb}(k,t) = md$	*findb*,*pr*(3.3.2)
3.3.3	$k < mk \vdash \mathit{findb}(k,t) \in \texttt{Data}$	*subsubsubgoal*
3.3.3.1	$k \in \mathit{collkeys}(lt)$	*collkeys2*, *pr*(3.3), *pr*(3.3.3)
3.3.3.2	$\mathit{findb}(k, lt) \in \texttt{Data}$	*pr*(3), p, (-1)
3.3.3.3	$\mathit{findb}(k,t) = \mathit{findb}(k, lt)$	*findb*, *pr*(3.3.3)
3.3.4	$mk < k \vdash \mathit{findb}(k,t) \in \texttt{Data}$ *similar*	*subsubsubgoal*
3.3.5	$\mathit{findb}(k,t) \in \texttt{Data}$	$\vee$-E(3.3.1, 3.3.2,3.3.3,3.3.4)
3.3.6	$p(k,t)$	$\vee$-I(-1), p
3.4	$p(k,t)$	$\vee$-E(3.1,3.2,3.3)
4.	$p(k,t)$	*struct induct*(2, 3), p

Q.E.D

Similar considerations arise for the function that insert values into trees.

$insb(K : \texttt{Key}, D : \texttt{Data}, T : \texttt{Bintree})R : \texttt{Bintree}$

$\underline{pre}\ k \notin collkeys(t)$

$insb(k, d, t) \stackrel{\text{def}}{=}$

$cases\ t$:

$$\begin{array}{lll} \texttt{nil} & \mapsto & mk\text{-}Binnode(\texttt{nil}, k, d, \texttt{nil}) \\ mk\text{-}Binnode(lt, mk, md, rt) & \mapsto & \\ \quad \underline{if} \quad k < mk \quad \underline{then} & & mk\text{-}Binnode(insb(k, d, lt), mk, md, rt) \\ \quad \underline{else} & & mk\text{-}Binnode(lt, mk, md, insb(k, d, rt)) \end{array}$$

Again this function becomes total w.r.t. its pre-condition.

Lemma(*insb1*)

$\forall t \in \texttt{Bintree}, k \in \texttt{Key}, d \in \texttt{Data}$

$k \in collkeys(t) \vee insb(k, d, t) \in \texttt{Bintree}$

The proof goes by *structural induction* and uses the following abbreviation:

$p(k, t) := k \in collkeys(t) \vee$

$(insb(k, d, t) \in \texttt{Bintree} \wedge collkeys(insb(k, d, t)) = collkeys(t) \cup \{k\}$

Proof:

$t \in \texttt{Bintree}, k \in \texttt{Key}, d \in \texttt{Data} \vdash p(k,t)$		*goal*
1.	$insb(k,d,\texttt{nil}) = mk\text{-}Binnode(\texttt{nil},k,d,\texttt{nil})$	*insb*
2.	$insb(k,d,\texttt{nil}) \in \texttt{Bintree}$	$^{(*)}(1.)$
3.	$collkeys(insb(k,d,\texttt{nil})) = \{k\}$	$collkeys(1.)$
4.	$collkeys(insb(k,d,\texttt{nil})) = collkeys(\texttt{nil}) \cup \{k\}$	$collkeys, (3.), (sets)$
5.	$p(k,\texttt{nil})$	$\wedge$-I(2.,4.),$\vee$-I, p
6.	$t = mk\text{-}Binnode(lt,mk,md,rt) \wedge invBinnode(t)$ $\wedge p(k,lt) \wedge p(k,rt) \vdash p(k,t)$	*subgoal*
6.1	$k \in collkeys(t) \vee k \notin collkeys(t)$	$collkeys1, (sets)$
6.2	$k \in collkeys(t) \vdash p(k,t)$	$\vee$-I, p
6.3	$k \notin collkeys(t) \vdash p(k,t)$	*subsubgoal*
6.3.1	$collkeys(t) = collkeys(lt) \cup \{mk\} \cup collkeys(rt)$	$collkeys, pr(6)$
6.3.2	$k < mk \ \vee \ mk < k$	$(-1), pr(6.3), \texttt{Key}$
6.3.3	$k < mk \vdash p(k,t)$	*subsubsubgoal*
6.3.3.1	$k \notin collkeys(lt)$	$pr(6.3), (6.3.1)$
6.3.3.2	$invBinnode(insb(k,d,lt))$	$(-1), pr(6.), p$
6.3.3.3	$collkeys(insb(k,d,lt)) = collkeys(lt) \cup \{k\}$	$(-2), pr(6.), p$
6.3.3.4	$insb(k,d,t) =$ $mk\text{-}Binnode(insb(k,d,lt),mk,md,rt)$	$insb, pr(6.3.3)$
6.3.3.5	$p(k,t)$	$\wedge$-I$(-1,-2)$, $\vee$-I, p
6.3.4	$mk < m \vdash p(k,t)$ – similar –	*subsubsubgoal*
6.3.5	$p(k,t)$	$\vee$-E(6.3.2, 6.3.3, 6.3.4)
6.4	$p(k,t)$	$\vee$-E(6.1, 6.2, 6.3)
7.	$p(k,t)$	*struct induct* $(5,6)$

Q.E.D

Note that the function

$delb(K : \texttt{Key}, T : \texttt{Bintree})R : \texttt{Bintree}$

is harder to write. Maybe enthusiastic readers will tackle this problem.

Now remains the task showing that a series of operations on `Bintree` *model* those on the abstract mapping (`Mkd`) data type. The key to these proofs is to relate the underlying objects by a function which retrieves the *abstraction* from the *representation*.

$$
\begin{array}{lll}
retrm : \texttt{Bintree} \mapsto \texttt{Mkd} & & \\
\quad cases \ \ t : & \texttt{nil} & \mapsto [] \\
 & mk\text{-}Binnode(lt,mk,md,rt) & \mapsto \underline{merge}([k \mapsto d], retrm(lt), retrm(rt))
\end{array}
$$

Properties:

a) *retrm* is total.

b) The representation is *adequate*, i.e.

$$\forall\, m \in \texttt{Mkd}\ \exists\, t \in \texttt{Bintree}\ (\mathit{retrm}(t) = m)$$

c) $\forall\, t \in \texttt{Bintree}\ (\underline{\mathit{dom}}(\mathit{retrm}(t)) = \mathit{collkeys}(t))$

d) $\mathit{retrm}(t_0) = m_0 \quad \text{or} \quad \mathit{retrm}(\texttt{nil}) = []$

e) $\forall\, t \in \texttt{Bintree}, k \in \texttt{Key}, d \in \texttt{Data}$

$$k \notin \mathit{collkeys}(t) \rightarrow \mathit{retrm}(\mathit{insb}(k, d, t)) = \mathit{retrm}(t) \cup [k \mapsto d]$$

f) $\forall\, t \in \texttt{Bintree}, k \in \texttt{Key}$

$$k \in \mathit{collkeys}(t) \rightarrow \mathit{findb}(k, t) = (\mathit{retrm}(t))(k)$$

We then get for the *INSERT* operation on `Bintree`

INSERTB(K:`Key`,D:`Data`)

$\underline{\mathit{ext}}$ $\underline{\mathit{wr}}$ $T : \texttt{Bintree}$
$\underline{\mathit{pre}}$ $k \notin \mathit{collkeys}(t)$
$\underline{\mathit{post}}$ $t = \mathit{insb}(k, d, \overleftarrow{t})$

Notice that the post-condition of *INSERTB* could also be

$\underline{\mathit{post}}$ $\mathit{retrm}(t) = \mathit{retrm}(\overleftarrow{t}) \cup [k \mapsto d]$

For the model of the *FIND* operation one might use:

FINDB(K:`Key`) D:`Data`

$\underline{\mathit{ext}}$ $\underline{\mathit{rd}}$ $T : \texttt{Bintree}$
$\underline{\mathit{pre}}$ $k \in \mathit{collkeys}(t)$
$\underline{\mathit{post}}$ $d = \mathit{findb}(k, \overleftarrow{t})$

5.5.4 Implementation

If we want to implement the trees in Pascal, a second step of data refinement is needed, since the trees above cannot be constructed directly. Instead each node must be created as a record on the heap, and nested trees must be represented by pointers. Thus

`Heap` = <u>map</u> `Ptr` <u>to</u> `Binnoderep`

`Root` = [`Ptr`]

$$\begin{array}{lll} \texttt{Binnoderep} :: & LP & : [\texttt{Ptr}] \\ & MK & : \texttt{Key} \\ & MD & : \texttt{Data} \\ & RP & : [\texttt{Ptr}] \end{array}$$

Without formalizing the statement, it is clear that the mapping should be well-founded and that all used keys should be in the domain of the map.

The remainder of the refinement step can be seen from:

$$\mathit{collkeysh} : \texttt{Root} \times \texttt{Heap} \mapsto \underline{\mathit{set\ of}}\ \texttt{Key}$$

$$\mathit{collkeysh}(p, m) \stackrel{\text{def}}{=}$$

$$\begin{array}{lll} \texttt{if} & p = \texttt{nil} & \texttt{then}\ \{\} \\ \texttt{else} & & (\underline{\texttt{let}}\ \mathit{mk\text{-}Binnoderep}(lp, mk, md, rp) = m(p)\ \underline{\texttt{in}} \\ & & \mathit{collkeysh}(lp, m) \cup \{mk\} \cup \mathit{collkeysh}(rp, m)) \end{array}$$

$$\mathit{findbhn} : \texttt{Key} \times \texttt{Ptr} \times \texttt{Heap} \mapsto \texttt{Binnoderep}$$

$$\underline{\mathit{pre}}\ k \in \mathit{collkeysh}(p, m)$$

$$\mathit{findbhn}(k, p, m) \stackrel{\text{def}}{=}$$

$$\begin{array}{llll} \underline{\mathit{let}}\ \mathit{mk\text{-}Binnoderep}(lp, mk, md, rp) = m(p)\ \underline{\mathit{in} & & & \\ \texttt{if} & k = mk & \texttt{then} & m(p) \\ \texttt{else if} & k < mk & \texttt{then} & \mathit{findbhn}(k, lp, m) \\ \texttt{else} & & & \mathit{findbhn}(k, rp, m) \end{array}$$

FINDBH(K:Key) D:Data

$$\begin{array}{ll} \underline{\mathit{ext}} & \underline{\mathit{rd}}\ RT \quad : \texttt{Root} \\ & \underline{\mathit{rd}}\ HP \quad : \texttt{Heap} \\ \underline{\mathit{pre}} & k \in \mathit{collkeysh}(rt, hp) \\ \underline{\mathit{post}} & d = MD(\mathit{findbh}(k, \overleftarrow{rt}), \overleftarrow{hp}) \end{array}$$

Now we note that for the implementation of

$$\mathit{insbhn} : \texttt{Key} \times \texttt{Data} \times \texttt{Ptr} \times \texttt{Heap} \mapsto \texttt{Ptr} \times \texttt{Heap}$$

it is much be easier to consider *insbhn* as a procedure, acting on Ptr × Heap, with given Key and Data, instead of a function as expressed in the above notation.

Then we get:

$$\mathit{insbhn}(K : \texttt{Key}, D : \texttt{Data},\ \texttt{Var}P : \texttt{Ptr}, M : \texttt{Heap})$$

$$\underline{\mathit{pre}}\ k \notin \mathit{collkeysh}(p, m)$$

$$
\begin{array}{l}
\mathit{insbhn}(k, d, p, m) \stackrel{\text{def}}{=} \\
\texttt{if} \quad p = \texttt{nil} \ \ \texttt{then} \ \ \texttt{new}(r); \\
\qquad\qquad\qquad\qquad p := r; \\
\qquad\qquad\qquad\qquad m(r) := \mathit{mk\text{-}Binnode}(\texttt{nil}, k, d, \texttt{nil}) \\
\texttt{else} \ \ \underline{\mathit{let}} \ \ \mathit{mk\text{-}Binnode}(lp, mk, md, rp) = m(p) \ \ \underline{\mathit{in}} \\
\qquad\qquad\qquad \texttt{if} \ \ k < mk \ \ \texttt{then} \ \ \mathit{insbh}(k, d, lp, m) \\
\qquad\qquad\qquad \texttt{else} \ \ \mathit{insbh}(k, d, rp, m)
\end{array}
$$

INSBH(K:`Key`,D:`Data`)

$$
\begin{array}{ll}
\underline{\mathit{ext}} & \underline{\mathit{wr}}\ RT : \texttt{Root} \\
 & \underline{\mathit{wr}}\ HP : \texttt{Heap} \\
\underline{\mathit{pre}} & k \notin \mathit{collkeysh}(rt, hp) \\
\underline{\mathit{post}} & (rt, hp) \leftarrow \mathit{insbhn}(k, d, \overleftarrow{rt}, \overleftarrow{hp})
\end{array}
$$

6 Parallel Programs

6.1 Disjoint Parallel Programs

(`disjoint parallelism`)

$$\frac{\{p_i\}\ S_1\{q_i\}\ ,\ i = 1, \ldots n}{\{\bigwedge_{i=1}^{n} p_i\}\ [S_1 \parallel \ldots \parallel S_n]\ \{\bigwedge_{i=1}^{n} q_i\}}$$

6.2 Parallel Programs with Shared Variables

(`parallelism with shared variables`)

$$\frac{\{p_i\}\ S_1^*\{q_i\}\ ,\ i = 1, \ldots n,\ \texttt{interference free}}{\{\bigwedge_{i=1}^{n} p_i\}\ [S_1 \parallel \ldots \parallel S_n]\ \{\bigwedge_{i=1}^{n} q_i\}}$$