Plankalkül

Bram Bruines (0213837)

January 8, 2010

$$
\begin{array}{c|l}
 & V \quad \Rightarrow Z \\
V & 0 \quad 0 \\
S & m\times(\sigma,\tau) \quad m\times(\sigma,\tau)
\end{array}
$$

$$
\begin{array}{c|l}
 & W1(m-1)\left[\begin{array}{l|l} Z \quad \Rightarrow Z & i \quad \Rightarrow \varepsilon \\ 0 \quad 1 & \\ i+1 & \\ (\sigma,\tau) \quad (\sigma,\tau) & 1.n \quad 1.n \end{array}\right. \\
V & \\
K & \\
S & \\
 & W\left[\varepsilon\geq 0 \overset{\rightarrow}{\cdot}\left[\begin{array}{l} Z{<}Z \ \vee(Z{=}Z \ \wedge Z{<}Z)\Rightarrow Z \\ 1\ 0\ 1\ 0\ 1\ 0\ 2 \\ 0\ \ \varepsilon.0\ \ 0\ \ \varepsilon.0\ \ 1\ \ \varepsilon.1 \\ \sigma\ \ \sigma\ \ \sigma\ \ \sigma \end{array}\right.\right. \\
V & \left.\left.\begin{array}{l} Z\overset{\rightarrow}{\cdot}\left[\begin{array}{l|l} Z \quad \Rightarrow Z & \varepsilon-1\Rightarrow\varepsilon \\ 0 \quad 0 & \\ \varepsilon \quad \varepsilon+1 & \\ (\sigma,\tau) \quad (\sigma,\tau) & \end{array}\right] \\ 2 \\ \overline{Z}\overset{\rightarrow}{\cdot}\left[\begin{array}{l|l} Z\Rightarrow Z & \mathrm{Fin}^3 \\ 1\ \ 0 & \\ \varepsilon\ \ \varepsilon+1 & \\ \sigma\ \ \sigma & \end{array}\right] \\ 2 \end{array}\right]\right] \\
K & \\
S & \\
 & \varepsilon \quad {=}{-}1\overset{\rightarrow}{\cdot}Z\Rightarrow Z \\
V & 1\ \ 0 \\
K & 0 \\
S & 1.n\ \ 1.n\ \ \sigma\ \ \sigma
\end{array}
$$

$$
\begin{array}{c|l}
 & Z \quad \Rightarrow R \\
V & 0 \quad 0 \\
S & m\times\sigma \quad m\times\sigma
\end{array}
$$

Contents

1 Introduction

The first modern programmable computers were the British Colossus, the American ENIAC and the German Z3. Both the ENIAC and the Colossus were built by teams of well-funded engineers, while the Z3 was built by Konrad Zuse, alone, based on an earlier version he had built in his parent's apartment. In 1998, four years after Zuse's death, a paper from Raúl Rojas demonstrated how the Z3 could be made Turing-complete. This article generated a renewed interest into many of Zuse's projects, most of which had not gotten much public attention. One of these was an obscure draft on a programming language from 1945.

This programming language called Plankalkül was what Raúl Rojas turned his attention to. In 2000 the Berlin University proudly announced the development of the first implementation of Plankalkül.[1] This has spawned a number of articles on Plankalkül containing introductions or biographies on Zuse, but little more formal. In some, Plankalkül is compared to Fortran or Algol 60 but only on a very general level. In this paper Plankalkül is examined in a more formal way in order to determine whether the draft of Plankalkül would have worked and the idea was simply ahead of its time.

The first draft of Plankalkül from 1945 contains some ambiguous features, but is on the whole well documented. Based on the draft its features will be explained and compared to modern programming languages. Then a semantics of Plankalkül will be built up in the style of Nielson & Nielson (1999). The basic features that make up the language will be described in a formal way in order to demonstrate that Plankalkül does not differ from modern programming languages apart from its remarkably early conception.

2 Plankalkül

The ENIAC, Colossus and Z3 were controlled by machine code, which means the program contains every calculation and operation the computer carries out. The ENIAC had to be rewired for each new computation, while the Colossus and Z3 could read punched tape containing the program to be carried out. What makes Plankalkül different, is that it is of a higher level. The program does not describe every step of every calculation. Instead it allows the programmer to use simple logical constructs. These are translated by the underling soft- or hardware to step-by-step machine code. A higher-level language offers a programmer simple building blocks to compose a program.

2.1 Plan

In Plankalkül, a program is composed of one or several blocks, called plans. A plan receives a number of input values, performs certain operations on them and returns a number of result values.

[1] Rojas (2000) , Rojas (2000a)

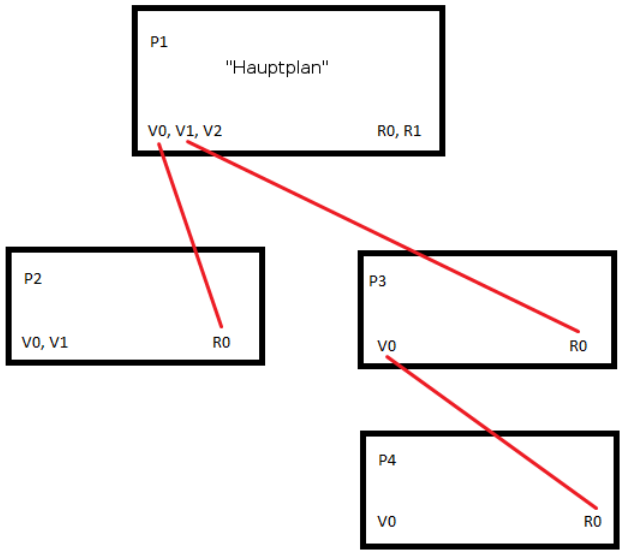


Figure 1: A chain of plans.

As is the modern method, a calculation is broken down into different tasks and for each a plan is written. The main plan or 'Hauptplan' is initally executed using the results of different plans, which can use the results of other plans, and so on. In this way all plans are chained together and a complete program is made. A plan is free from side-effects and there is no recursion.

2.2 Variables

Variables can not be given an arbitary name, but are referred to as V0, R0 or Z0, depending on their function. The V variables are used as the input variables for a plan, and are readonly. The Z variables or 'Zwischenwerte' are used for intermediary storage within a plan. The results of a plan are referred to as R values. Every plan is preceded by a 'Randauszug', which defines the number of parameters and results as well as their corresponding type.

$\mathrm{R}(V_0, V_1) \Rightarrow (R_0, R_1)$

The above plan has two input values V_0 and V_1 and returns two result values R_0 and R_1. In order to use the result of plan 1.2 as an parameter for plan 1, the following Randauszug can be employed.

$R_0 1.2(V_0) \Rightarrow R_0$

This Randauszug ensures that the result R_0 of plan 1.2 is used as a parameter for this plan, which returns a single variable. In this way plans can be chained together. Naturally the types of all the variables must be equivalent, otherwise this will not work[2].

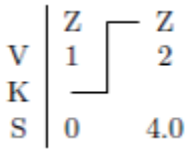
[2]Zuse (1945), p. 13-14.

2.3 Notation

The strangest aspects of Plankalkül for a modern programmer is likely to be Zuse's twodimensional notation. The index of an array is not written after the variable name, but is written immediatly below the variable name. The type of the variable, which must always be given explicitly, is written below that. Variables do not have to be declared.

	Z	Z	
V	4	2	*Part of the variable name, so here we have Z_4 and Z_2.*
K	2.3		*Variable index, we refer to $Z_4[2.3]$ and Z_2.*
S	0	0	*The type of $Z_4[2.3]$ and Z_2, in both cases a single bit.*

There is also a way to get the index for a variable from another variable.



The value of the variable Z_2 is used as the index for the variable Z_1, thus we get $Z_1[Z_2]$. While implementing Plankalkül 2000, a more conventional notation was chosen and the index and type are written after the variable name.

V0[1:5.0]	Variable V_0, component 1, of type 5.0
Z1[5.3 :9.0]	Variable Z_0, component 3 of component 5, of type 9.0 [3]
Z2[:12.3.0]	Variable Z_2, of type 12.3.0

In this paper a more conventional notation will be used for convenience. I will omit the type in examples whenever this is not relevant and shall use subscript to indicate the number and brackets to indicate the index of a variable, except in blocks of literal code. The variables above for instance will be written $V_0[1]$, $Z_1[5.3]$ and Z_2.

2.4 Datatypes

The available datatypes in Plankalkül are certainly very low-level. The only primitive datatype is the bit, from which more advanced datatypes can be constructed. The simplest datatype is 0, a single bit, called a 'Ja-Nein-Werte'. An array of bits is represented as n x 0 and an array of arrays of bits can be represented as m x n x 0. We can also create tuples. For instance, (0,0) is a tuple of two bits and (0,0,0) would be a tuple of three bits.

These same processes can be applied to any already existing datatype. Supposing we have a datatypes σ and τ we can make a tuple (σ,τ) and an array like m x σ. Ofcourse we can make a tuple (σ, m x τ) and so on.[4]

[3] Rojas (2000), p.12.

[4] I am ignoring quite a lot of alternative notation concerning types and declarations. In some cases there are two or three different ways of denoting the same datatype which can in some cases be handy but is mostly confusing. I shall therefore employ only one clear notation. Syntactical alternatives can be found in Zuse (1945), p.3-4.

	Z	V	Z
V	0	1	1
K		1	i
S	0	(σ,σ)	4 x 0

Z_0 is a bit. V_1 is a tuple of type σ. Z_1 is an array of four bits. Zuse devoted much time to describing the process of finding the right datatypes to represent problems and composing suitable datatypes. He also gives a few more advanced datatypes.[5]

- A8 A natural number.
- A9 A whole positive number.
- A10 A whole number (either positive or negative).
- A11 A positive fraction.
- A12 A fraction
- A12 A complex number.

Although the examples Zuse gives are more like scetches than complete definitions, these numbers can be used like any other datatype.

	Z	+	V	⇒	R
V	0		0		0
K	1				
S	12		12		12

For instance we could add two complex numbers and return a complex result as above. So even though the datatypes are all composed of bits, fairly complex datatypes can be constructed and used without too much trouble. The lack of characters, and input / output facilities is notable as those features tend to be added to any implementation fairly quickly.

2.5 Operators

There are a number of very familiar operators in Plankalkül, and some that may appear quite foreign at first glance. **Assignment** is perhaps the most basic operation.

$10 \Rightarrow Z_0$

This statement assigns the value 10 to variable Z_0. Needless to say, the type must be given and must be sufficient to contain the value 10. The next operator is the **conditional execution** or 'if'.

$Z_0 \underset{\cdot}{\rightarrow}$Statement

[5] Zuse (1945), p. 120-121.

If Z_0 evaluates as true the Statement is executed, otherwise it is ignored. An 'if then else' construction can obviously be constructed easily:

$Z_0 \dot{\rightarrow}$ Statement1
$\overline{Z_0} \dot{\rightarrow}$ Statement2

In Plankalkül, $\overline{Z_0}$ is true if Z_0 is not and thus this is equal to the fragment of 'if (Z_0) then Statement1; else Statement2;'. Next, there are the **looping constructs**. The first and most general loop is simply called 'W' for 'Wiederhohlung' or repetition.

$$\text{W} \left[\begin{array}{c} Z_0 \dot{\rightarrow} Statement1 \\ Z_1 \dot{\rightarrow} Statement2 \\ ... \end{array} \right]$$

The function of the 'W' operator is to repeatedly execute any of the statements for which the precondition evaluates as true, until all preconditions are false, or until the special symbol 'Fin' is encountered, which acts as a break. There are six other looping constructs W0 ... W5, which offer variations on the same operation. They are given one (in the case of W0 to W3) or two (in the case of W4 and W5) integer variables which are constant. Every loop employs a local variable such as e or i, which is checked against the constant value.

$$\text{W0 } (V_0) \left[\begin{array}{c} Statement1 \\ ... \end{array} \right]$$

The W0 operator simply executes all statements in the following block for a number of times , specified in the variable V_0. This variable can usually be accessed and accessed from within the block[6], and is automatically decreased or increased by one every cycle. When the initial value is zero, the loop will not run.[7] The W1 and W2 operators are used to loop through the components of an array.

$$\text{W1}(n) \left[\begin{array}{ccc} V_0[i] & \Rightarrow & R_0[i] \\ & ... & \end{array} \right]$$

In the example above, the components 0 - n of variable V_0 are copied to R_0. The difference between W1 and W2 is that in the former case the variable i counts up from 0 to n and in the latter case the variable i counts down from n to 0. The constructs W3 and W4 take two integers (n,m) and respectively count while $m \geq n$ and while $m \leq n$. W5 counts up or down while $m=n$ depending on whether $m<n$ or $m>n$.

2.6 Predicates

Zuse also describes the possiblity to define plans which act like logical predicates. He mentions the predicate 'for all', denoted by (x) R(x) (a more modern notation would be $\forall$x R(x)). This predicate could be declared in Plankalkül, but will need to get a domain, or the type of the elements for which this predicate will hold.

[6] There is one exception to this rule. That is in fact W0. More details on this choice are given below.
[7] Zuse (1945), p.27-28.

	(x)	R(x)
V		
K		
S	8	8

The predicate above would be denoted in a more modern notation as $\forall_{x:\mathbb{Z}}\mathrm{R(x)}$. A predicate is presumably, simply a plan which accepts one or more elements of the right type as an argument and returns a single boolean, representing either true or false. He then combines these predicates with the concept of a set to define several basic operators of set theory.

$\mathrm{x} \in V_1$	*Test whether x is an element of the set* V_1
$(\mathrm{x})(\mathrm{x} \in V_1 \Rightarrow \mathrm{R(x)})$	*Test whether the property R(x) holds for every element of the set* V_1.
$(\mathrm{Ex})(\mathrm{x} \in V_1 \Rightarrow \mathrm{R(x)})$	*Test whether there is an element of the set* V_1*, for which the property R(x) holds.*

Zuse goes on to define many different operations which are based on predicates and sets or sequences. The listoperators of the next section are an example.

2.7 Listoperators

Zuse realized that an array can hold a set or sequence of different values and he designed a number of operators specificaly for sets or sequences. These operators are common to functional programming languages and are not usually found in imperative languages.

$\acute{\mathrm{x}}(\mathrm{x} \in V_1 \wedge \mathrm{R(x)})$	*Returns the single x from the set* V_1 *for which R(x) holds.*
$\hat{\mathrm{x}}(\mathrm{x} \in V_1 \wedge \mathrm{R(x)})$	*Returns a subset of the set* V_1 *composed of every element for which R(x) holds.*
$\hat{}\hat{\mathrm{x}}(\mathrm{x} \in V_1 \wedge \mathrm{R(x)})$	*Returns a sequence composed of every element from* V_1 *for which R(x) holds.*

The precondition for $\acute{\mathrm{x}}(\mathrm{x} \in V_1 \wedge \mathrm{R(x)})$ is that there exists such a unique x. The functions above can easily be translated to a modern functional programming languages such as Haskell:

```
-- Return the single value from a link for which a property holds
xacute :: Num a => [a] -> (a -> Bool) -> a
xacute (x:xs) f
      | length (filter f (x:xs)) == 1  = (head (filter f (x:xs)))

-- Return a subset of a set, composed of every element for which a property holds
xcircum :: Num a => [a] -> (a -> Bool) -> [a]
xcircum [] f     = []
xcircum l  f     = nub (filter f l)

-- Return a sequence composed of every element from a set for which a property holds
xdblcircum :: Num a => [a] -> (a -> Bool) -> [a]
xdblcircum [] f     = []
xdblcircum l  f     = filter f l
```

There are also two operators which were probably only intended for use in a loop.

$\mu\mathrm{x}(\mathrm{x} \in l \wedge \mathrm{R(x)})$
$\lambda\mathrm{x}(\mathrm{x} \in l \wedge \mathrm{R(x)})$

The first, when used in a loop, returns a new element from an array each cycle of the loop, until every element has been processed or the loop has terminated in some other way. The latter is used in a similar way but moves from the end to the front.

```
-- These functions return either a value, or the symbol Fin
-- we can represent this in Haskell with the following datatype;
data Zuse a = Val a | Fin

-- Mu
mu :: Num a => Int -> [a] -> (a -> Bool) -> Zuse a
mu _ l f
  | filter f l == [] = Fin
mu 0 l f = Val (head (filter f l))
mu 0 l f = Val (head (filter f l))
mu n l f
  | length(filter f l) > n = Val (head (drop n (filter f l)))
  | otherwise              = Fin

-- Lambda
lambda :: Num a => Int -> [a] -> (a -> Bool) -> Zuse a
lambda _ l f
  | filter f l == [] = Fin
lambda 0 l f = Val (head (reverse(filter f l)))
lambda 0 l f = Val (head (reverse(filter f l)))
lambda n l f
  | length(filter f l) > n = Val (head (drop n (reverse(filter f l))))
  | otherwise              = Fin
```

Loops are of course not a feature of functional programming languages like Haskell, where recursion is used instead. The operators μx(x $\in l \wedge$ R(x)) and λx(x $\in l \wedge$ R(x)) were intended to be used in loops, where the loop has a local iteration variable. Using this variable as demonstrated, we can give Haskell functions which are equivalent.

2.8 Examples

I will now give a number of short programs and descriptions of exactly what they are supposed to do. I hope this will help in familiarizing the reader with the sometimes daunting task of reading Zuse's syntax. The second and third examples are taken from Zuse (1945).

2.8.1 Example 1

The first plan, called P1.1, calculates the factorial of a natural number.

P1.1

	R(V) ⇒	R
V	0	0
K		
S	10	10

	0 ⇒	R
V		0
K		
S		10

	W1(V)	R × i ⇒	R
V		0	0
K		10	10
S			

The first line contains the Randauszug and defines that P1.1 takes one argument, an integer called V_0 and returns an integer called R_0. The program then assigns the value '0' to R_0 and proceeds to loop as many times as the argument dictates, and adds the value to R_0. When the loop finishes, value R_0 contains the $n!$.

2.8.2 Example 2

The following plan is taken from Zuse[8], and is an operation an a list.

P3.16

	$R(V)$	$\Rightarrow R$
V	0	0
S	$m \times \sigma$	$m \times \sigma$

	$\vert W1(m)$	$V \Rightarrow R$	
V		0	0
K		i	$m-1-i$
S		σ	σ

As the Randauszug shows, this plan takes an array of elements of type σ and returns a list of the same type and dimension. The plan then enters into a loop, once for every element in the list. The elements of the list V_0 are then copied into R_0, back to front. The result is returned, a reversed copy of the received list.

2.8.3 Example 3

The last example is a larger plan, also given by Zuse[9], but slightly changed in order to conform to the syntax used in this paper. As the Randauszug shows the plan takes an array of values and returns an array of the same type and dimension.

[8] Zuse (1945), p.70
[9] Zuse (1945), p.73

P3.27

$$\begin{array}{l} \text{Ord } 1(V) \quad \Rightarrow R \\ \qquad\quad 0 \qquad\quad 0 \\ \qquad\quad m\times\sigma \quad m\times\sigma \end{array}$$

$$\left|\begin{array}{l|ll} & V & \Rightarrow Z \\ V & 0 & 0 \\ S & m\times\sigma & m\times\sigma \end{array}\right.$$

$$\left|\begin{array}{l|l} & W1(m-1)\left[\begin{array}{l|ll} & Z\Rightarrow Z & i \quad \Rightarrow\varepsilon \\ & 0 \quad 1 & \\ & \sigma \quad \sigma & 1.n \quad 1.n \end{array}\right. \\ V & \qquad W\left[\varepsilon\geq 0\overset{\rightarrow}{\cdot}\left[\begin{array}{ll|ll} Z<Z\overset{\rightarrow}{\cdot} & & Z\Rightarrow Z & \varepsilon-1\Rightarrow\varepsilon \\ 1 \quad 0 & & 0 \quad 0 & \\ \quad\ \varepsilon & & \varepsilon \quad \varepsilon+1 & \\ \sigma \quad \sigma & & \sigma \quad \sigma & \end{array}\right]\right] \\ K & \qquad\quad \left[\begin{array}{ll|ll} \overline{Z<Z}\overset{\rightarrow}{\cdot} & & Z\Rightarrow Z & \text{Fin}^3 \\ 1 \quad 0 & & 0 \quad 0 & \\ \quad\ \varepsilon & & \quad\ \varepsilon+1 & \\ \sigma \quad \sigma & & \sigma \quad \sigma & \end{array}\right] \\ S & \varepsilon \quad =-1\overset{\rightarrow}{\cdot}Z\Rightarrow Z \\ & \qquad\quad 1 \quad 0 \\ V & \qquad\quad\ \ 0 \\ K & 1.n \quad 1.n \quad \sigma \quad \sigma \\ S & \end{array}\right.$$

$$\left|\begin{array}{l|ll} & Z & \Rightarrow R \\ V & 0 & 0 \\ S & m\times\sigma & m\times\sigma \end{array}\right.$$

The first operation is to copy V_0 into a temporary variable, Z_0 because V_0 is readonly and cannot be altered. Then a complicated block is looped one short of once for every element in the array.

Within the loop, the first value is copied into a new temporary variable Z_1 and the value of the variable is copied into a variable e. A new loop is for the remaining elements, wherin elements in the array switch positions if the latter is greater than the former. Then when all elements have been dealt with, the very last element is put infront of the array. Finally the sorted list from Z_0 is copied into an output variable R_0.

2.9 Omissions

There are a number of features and functions that Zuse describe or merely mentions which fall beyond the score of this paper. A short list will be given below.

- It is possible to create function templates. Zuse describes the possibility to give a basic function as an argument to a function. It is thus possible to create a plan which operates on functions.
- There are many syntactic alternatives which can be very confusing. In each case I have tried to choose the most intuitive notation and to ignore the other choices.
- Zuse gives more datatypes then the ones I have listed in the section above. Zuse has a tendency to suggest and hint at many different possibilites. None of these are significantly different from the ones I have listed however.

- There is a list of builtin functions (Min, Max, etc.)[10] which I leave out of the syntax. All of these functions can be composed with the blocks which are available. In most cases, Zuse gives these compositions himself.
- There is another looping construct W6, which operates on a list, taking values until the list is empty. But I did not gain a sufficient understanding from the description[11] to formalize it.

3 Formalization

The syntax of Plankalkül will be given in an extended Backus Naur Form, and employs the following categories.

Variables	x_0, x_1, x_2
Boolean Expressions	b_0, b_1, b_2
Arithmatic Expressions	a_0, a_1, a_2
List Expressions	l_0, l_1, l_2
Statements	S_0, S_1, S_2
Numerical Expressions	n_0, n_1, n_2

The syntax in Plankalkül will be represented by the following syntax.

Program	::=	Plan $\mid$ Program
Plan	::=	Randauszug Stm
Randauszug	::=	R(Vars) $\Rightarrow$ (Vars)
Vars	::=	$x \mid$ Vars, x
BitExp	::=	$+ \mid$ - $\mid \overline{b} \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \mid b_0 \rightarrow b_1 \mid b_0 = b_1 \mid b_0 \sim b_1 \mid b_0 \nsim b_1 \mid x \in l \mid$ $(x)(\mathrm{x} \in l \Rightarrow \mathrm{R(x)}) \mid (\mathrm{Ex})(\mathrm{x} \in l \Rightarrow \mathrm{R(x)})$
ArithExp	::=	$n \mid x \mid a_0 + a_1 \mid a_0$ - $a_1 \mid a_0 \times a_1 \mid a_0 \div a_1 \mid \mathrm{N}(l) \mid \acute{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R(x)})$
ListExp	::=	$\varnothing \mid \hat{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R(x)}) \mid \hat{}\hat{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R(x)})$

[10] Zuse (1945), p.116
[11] Zuse (1945), p.30

LoopStm	::=	μx(x $\in l \wedge$ R(x)) | λx(x $\in l \wedge$ R(x)) | Stm | *LoopStm*
CondStm	::=	$\begin{matrix} b \dot{\rightarrow} S \\ CondStm \end{matrix}$ | $b \dot{\rightarrow} S$ | Fin
Stm	::=	AExp $\Rightarrow x$ | BExp $\Rightarrow x$ | $S_0 \mid S_1$ | $\begin{matrix} S_0 \\ S_1 \end{matrix}$ | $b \dot{\rightarrow} S$ | $W\,[CondStm]$ |
		$W0(n)\,[LoopStm]$ | $W1(n)\,[LoopStm]$ | $W2(n)\,[LoopStm]$ | $W3(n,m)\,[LoopStm]$ |
		$W4(n,m)\,[LoopStm]$ | $W5(n,m)\,[LoopStm]$

[12]

This syntax shows how statements in Plankalkül can be constructed. Every rule will be given a semantics to describe the effect of the statement on states.

A state is a representation of computer memory, formaly defined as a function from a variable name to a value. The notation from Nielson & Nielson (1999) denoted states like s, s′, s″. Supposing we have a variable x which represents the value 5 in state s, the function s x results in the value 5. Substitutions are also possible, s[x $\mapsto$ 5] means change the value of the variable x in state s to the value 5.

3.1 Boolean Expressions

The function $\mathcal{B}$ maps a boolean expression and a state to a truth value: tt for true and ff for false.

$\mathcal{B}$: BitExp $\rightarrow$ (State $\rightarrow$ {tt, ff})

The semantics of bits are given as follows:

[12]The BitExp expresssions + and - are not mathematical symbols. Instead Zuse uses + to denote the boolean value true and - to denote the boolean value false. At Stm there are two, semantically equivalent composition statements. The exclusion of one of them significantly decreases the readability of the code. Finally, Zuse uses the same unfortunate symbol he uses to represent assignment in (x)(x $\in l \Rightarrow$ R(x)) and (Ex)(x $\in l \Rightarrow$ R(x)). This symbol should not be interpreted as an assignment neither should it be interpreted as a logical implication.

$$
\begin{array}{rcl}
\mathcal{B}[\![+]\!]\mathrm{s} & = & \mathrm{tt} \\
\mathcal{B}[\![\text{-}]\!]\mathrm{s} & = & \mathrm{ff} \\
\mathcal{B}[\![\bar{b}]\!]\mathrm{s} & = & \begin{cases} \mathrm{tt} \text{ if } \mathcal{B}[\![b]\!]\mathrm{s} = \mathrm{ff} \\ \mathrm{ff} \text{ if } \mathcal{B}[\![b]\!]\mathrm{s} = \mathrm{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]\mathrm{s} & = & \begin{cases} \mathrm{tt} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \\ \mathrm{ff} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ or } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \end{cases} \\
\mathcal{B}[\![b_1 \vee b_2]\!]\mathrm{s} & = & \begin{cases} \mathrm{ff} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{ff} \text{ or } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{ff} \\ \mathrm{tt} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \end{cases} \\
\mathcal{B}[\![b_1 \rightarrow b_2]\!]\mathrm{s} & = & \begin{cases} \mathrm{ff} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ or } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{ff} \\ \mathrm{tt} \text{ otherwise} \end{cases} \\
\mathcal{B}[\![b_1 \sim b_2]\!]\mathrm{s} & = & \begin{cases} \mathrm{tt} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \\ \mathrm{tt} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{ff} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{ff} \\ \mathrm{ff} \text{ otherwise} \end{cases} \\
\mathcal{B}[\![b_1 \nsim b_2]\!]\mathrm{s} & = & \begin{cases} \mathrm{ff} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \\ \mathrm{ff} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{ff} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{ff} \\ \mathrm{tt} \text{ otherwise} \end{cases} \\
\mathcal{B}[\![x \in l]\!]\mathrm{s} & = & \begin{array}{l} \text{Let l} = \mathcal{L}[\![l]\!]\mathrm{s} \\ \begin{cases} \mathrm{tt} \text{ if } \mathcal{A}[\![x]\!]\mathrm{s} \in \mathrm{l} \\ \mathrm{ff} \text{ otherwise} \end{cases} \end{array} \\
\mathcal{B}[\![\mathrm{R}(x)]\!]\mathrm{s} & = & \begin{array}{l} \text{Let s}' \text{ be s}' \text{ such that } \langle R(x), s \rangle \rightarrow \mathrm{s}' \\ \begin{cases} \mathrm{tt} \text{ if } s'\ \mathrm{x} = \mathrm{tt} \\ \mathrm{ff} \text{ otherwise} \end{cases} \end{array} \\
\mathcal{B}[\![(\mathrm{x})(x \in l \Rightarrow \mathrm{R}(x))]\!]\mathrm{s} & = & \begin{array}{l} \text{Let l} = \mathcal{L}[\![l]\!]\mathrm{s} \\ \begin{cases} \mathrm{tt} \text{ if } \forall \mathrm{a{:}l}\ \mathcal{B}[\![\mathrm{R(x)}]\!]\mathrm{s[x/a]} = \mathrm{tt} \\ \mathrm{ff} \text{ otherwise} \end{cases} \end{array} \\
\mathcal{B}[\![(\mathrm{Ex})(x \in l \Rightarrow \mathrm{R}(x))]\!]\mathrm{s} & = & \begin{array}{l} \text{Let l} = \mathcal{L}[\![l]\!]\mathrm{s} \\ \begin{cases} \mathrm{tt} \text{ if } \exists \mathrm{a{:}l} \text{ such that } \mathcal{B}[\![\mathrm{R(x)}]\!]\mathrm{s[x/a]} = \mathrm{tt} \\ \mathrm{ff} \text{ otherwise} \end{cases} \end{array}
\end{array}
$$

3.2 Arithmatic Expressions

$\mathcal{A}$: ArithExp $\rightarrow$ (State $\hookrightarrow \mathbb{Z}$)

The function $\mathcal{A}$ maps an Arithmatic Expression to a real number. In contrast to the other interpretation functions, this one is partial since the construct $\acute{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R(x)})$ is partially defined.

$$
\begin{array}{rcl}
\mathcal{A}[\![n]\!]s & = & n \\
\mathcal{A}[\![x]\!]s & = & \mathrm{s}\ x \\
\mathcal{A}[\![a_0 + a_1]\!]\mathrm{s} & = & \mathcal{A}[\![a_0]\!]\mathrm{s} + \mathcal{A}[\![a_1]\!]\mathrm{s} \\
\mathcal{A}[\![a_0 \times a_1]\!]\mathrm{s} & = & \mathcal{A}[\![a_0]\!]\mathrm{s} \times \mathcal{A}[\![a_1]\!]\mathrm{s} \\
\mathcal{A}[\![a_0 \text{ - } a_1\ ]\!]\mathrm{s} & = & \mathcal{A}[\![a_0]\!]\mathrm{s} \text{ - } \mathcal{A}[\![a_1]\!]\mathrm{s} \\
\mathcal{A}[\![a_0 \div a_1]\!]s & = & \left\lfloor \frac{\mathcal{A}[\![a_0]\!]s}{\mathcal{A}[\![a_1]\!]s} \right\rfloor \\
\mathcal{A}[\![\mathrm{N}(l)]\!]s & = & \#(\mathcal{L}[\![l]\!]s) \\
 & & \text{a such that} \\
\mathcal{A}[\![\ \acute{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R}(\mathrm{x}))\ ]\!]\mathrm{s} & = & \text{if l be } \mathcal{L}[\![l]\!]\mathrm{s}, \\
 & & \text{then } a \in l \text{ and } \mathcal{L}[\![\mathrm{R}(\mathrm{x})]\!]\mathrm{s}[\mathrm{x/a}] = \mathrm{tt}
\end{array}
$$

3.3 List Expressions

Zuse has introduced two sorts of operators based on array datatypes. The first operates on sets, the second operates on sequences. The practical difference in Plankalkül is that sequences can contain more than one copy of the same element. A set cannot contain duplicates. For the semantics these will require two separate interpretation functions.

$\mathcal{L}$: ListExp $\rightarrow$ (State $\rightarrow$ $\{\mathbb{Z}\}$)
$\mathcal{L}'$: ListExp $\rightarrow$ (State $\rightarrow$ $(\mathbb{Z})$)

The first function maps a list expression and a state to a mathematical set, the second function maps a list expression and a state to a sequence.

$$
\begin{array}{rcl}
\mathcal{L}[\![\ \varnothing\ ]\!]\mathrm{s} & = & \varnothing \\
\mathcal{L}[\![\ \hat{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R}(\mathrm{x}))\ ]\!]\mathrm{s} & = & \{\ \mathrm{x} \mid \mathrm{x} \in \mathcal{L}[\![\mathrm{l}]\!]\mathrm{s} \wedge \mathcal{B}[\![\mathrm{R(a)}]\!]\mathrm{s}[\mathrm{a/x}] = \mathrm{tt}\ \} \\
\mathcal{L}'[\![\ \hat{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R}(\mathrm{x}))\ ]\!]\mathrm{s} & = & (\ \mathrm{x} \mid \mathrm{x} \in \mathcal{L}[\![\mathrm{l}]\!]\mathrm{s} \wedge \mathcal{B}[\![\mathrm{R(a)}]\!]\mathrm{s}[\mathrm{a/x}] = \mathrm{tt}\ )
\end{array}
$$

To give an example of the principle difference between these operators. If we suppose l is an array of integers, and denotes the following array in modern notation: [1, 2, 3, 3, 4, 5, 6] and the function R(x) returns true if the value of x is odd and false if the value of x is even, we get the following results:

$\mathcal{L}[\![\ \hat{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R}(\mathrm{x}))\ ]\!]\mathrm{s} = \{1,3,5\}$
$\mathcal{L}'[\![\ \hat{\mathrm{x}}(\mathrm{x} \in l \wedge \mathrm{R}(\mathrm{x}))\ ]\!]\mathrm{s} = (1,3,3,5)$

There are also two statements which are designed for use specifically in a loop, $\mu\mathrm{x}(\mathrm{x} \in l \wedge \mathrm{R}(\mathrm{x}))$ and $\lambda\mathrm{x}(\mathrm{x} \in l \wedge \mathrm{R}(\mathrm{x}))$. Both are meant to yield every element of a list which conforms to specific conditions. When there is no more element, the special symbol Fin is yielded.

$\mathcal{A}^{loop}$: ListExp $\rightarrow$ (State $\rightarrow$ $\mathbb{Z} \cup$ Fin)

$$\mu\text{x}(\text{x} \in l \wedge \text{R}(\text{x})) \quad = \quad \begin{cases} \text{Fin if } \mathcal{A}^{loop}[\![\text{N}(l)]\!]\text{s} > \text{s } i \\ \text{The } n^{th} \text{ value of } \mathcal{L}[\![\text{l}]\!]\text{s otherwise} \end{cases}$$

$$\lambda\text{x}(\text{x} \in l \wedge \text{R}(\text{x})) \quad = \quad \begin{cases} \text{Fin if } \mathcal{A}^{loop}[\![\text{N}(l)]\!]\text{s} > \text{s } i \\ \text{The } (\mathcal{A}^{loop}[\![\text{N}(l)]\!]\text{s - } n)^{th} \text{ value of } \mathcal{L}[\![\text{l}]\!]\text{s otherwise} \end{cases}$$

In accordance to the syntax, the above commands can only be used within a loop, otherwise the looping variable i is not defined.

3.4 Statements

Statements are handled by a function from state to state. A statement S_0 and a state s are mapped to a new state s′, $\langle \text{S, s} \rangle \rightarrow \text{s}'$, in accordance to a big step formantics in the style of Nielsen & Nielsen (1999). The purpose of each transition is to map a statement from an initial state to the state at the end of the execution of the statement.

$$[\text{ass}^1] \quad \langle a \Rightarrow x, \text{s} \rangle \rightarrow \text{s}[x \mapsto \mathcal{A}[\![a]\!]\text{s}]$$

$$[\text{ass}^2] \quad \langle b \Rightarrow x, \text{s} \rangle \rightarrow \text{s}[x \mapsto \mathcal{B}[\![b]\!]\text{s}]$$

$$[\dot{\rightarrow}_{tt}] \quad \frac{\langle S, \text{s} \rangle \rightarrow \text{s}'}{\langle b \mathbin{\dot{\rightarrow}} S, \text{s} \rangle \rightarrow \text{s}'} \qquad \text{if } \mathcal{B}[\![b]\!]\text{s} = \text{tt}$$

$$[\dot{\rightarrow}_{ff}] \quad \langle b \mathbin{\dot{\rightarrow}} S, \text{s} \rangle \rightarrow \text{s} \qquad \text{if } \mathcal{B}[\![b]\!]\text{s} = \text{ff}$$

$$[\text{comp}_1] \quad \frac{\langle S_1, \text{s} \rangle \rightarrow \text{s}', \langle S_2, \text{s}' \rangle \rightarrow \text{s}''}{\langle S_1 \mid S_2, \text{s} \rangle \rightarrow \text{s}''}$$

$$[\text{comp}_2] \quad \frac{\langle S_1, \text{s} \rangle \rightarrow \text{s}', \langle S_2, \text{s}' \rangle \rightarrow \text{s}''}{\left\langle \begin{matrix} S_0 \\ S_1 \end{matrix}, s \right\rangle \rightarrow \text{s}''}$$

As mentioned above, there are two equivalent composition rules.

3.5 Iteration statements

Since iteration in Plankalkül can be a bit complicated, and there are so many different variation on the theme, this section will exclusively deal with iteration. The first construction, 'W' is the most difficult to formalize because it is the most adaptable. It runs until either all conditional statements are false, or when the Fin symbol is encountered.

$$[W_{ff}] \qquad \langle \text{W} \begin{bmatrix} b_0 \mathrel{\dot{\to}} S_0 \\ \dots \\ b_n \mathrel{\dot{\to}} S_n \end{bmatrix}, \text{s}\rangle \to \text{s} \qquad \text{if } \mathcal{B}[\![b_0]\!]\text{s} = \text{ff} \dots \mathcal{B}[\![b_n]\!]\text{s} = \text{ff}$$

$$[W_{fin}] \qquad \langle \text{W} \begin{bmatrix} b_0 \mathrel{\dot{\to}} S_0 \\ \dots \\ b_n \mathrel{\dot{\to}} S_n \\ \text{Fin} \\ \dots \end{bmatrix}, \text{s}\rangle \to \text{s} \qquad \text{if } \mathcal{B}[\![b_0]\!]\text{s} = \text{ff} \dots \mathcal{B}[\![b_n]\!]\text{s} = \text{ff}$$

$$[W_{tt}] \qquad \frac{\langle S_{i^*}, \text{s}\rangle \to \text{s}', \langle \text{W} \begin{bmatrix} b_0 \mathrel{\dot{\to}} S_0 \\ \dots \\ b_n \mathrel{\dot{\to}} S_n \end{bmatrix}, \text{s}'\rangle \to \text{s}''}{\langle \text{W} \begin{bmatrix} b_0 \mathrel{\dot{\to}} S_0 \\ \dots \\ b_n \mathrel{\dot{\to}} S_n \end{bmatrix}, \text{s}\rangle \to \text{s}''} \qquad \begin{array}{l}\text{if } \mathcal{B}[\![b_{i^*}]\!]\text{s} = \text{tt}, \text{ where } i^* \text{ is} \\ \text{the lowest number such that } \mathcal{B}[\![b_a]\!]\text{s} = \text{tt}\end{array}$$

Zuse shows how W0 to W5 can be implemented using the basic W construct. This greatly helps to clarify precisely how he wanted them to function. As opposed to the simple W, W1 to W5 have an iteration variable which is used to keep track of the numbers of cycles of iteration. These variables, like the variables n and m which are arguments to the loop, are strictly integers.

$$\begin{array}{llll}
W0(n) & [P] & 0 \Rightarrow \varepsilon \mid & W \quad [\, \varepsilon < n \mathrel{\dot{\to}} [P \quad \mid \varepsilon + 1 \Rightarrow \varepsilon \,]] \\
W1(n) & [P(i)] & 0 \Rightarrow i \mid & W \quad [\, i < n \mathrel{\dot{\to}} [P(i) \mid i + 1 \Rightarrow i \,]] \\
W2(n) & [P(i)] & n \vdash 1 \Rightarrow i \mid & W \quad [\, i > 0 \mathrel{\dot{\to}} [P(I) \mid i - 1 \Rightarrow i \,]] \\
W3(n,m) & [P(i)] & n \Rightarrow i \mid & W \quad [\, i < m \mathrel{\dot{\to}} [P(i) \mid i + 1 \Rightarrow i \,]] \\
\underline{(n \le m)} & & & \\
W4(n,m) & [P(i)] & n \Rightarrow i \mid & W \quad [\, i > m \mathrel{\dot{\to}} [P(i) \mid i - 1 \Rightarrow i \,]] \\
\underline{(n \ge m)} & & & \\
W5(n,m) & [P(i)] & n \Rightarrow i \mid & W \quad \begin{bmatrix} i \neq m \mathrel{\dot{\to}} \begin{bmatrix} P(i) \\ m > n \mathrel{\dot{\to}} (i + 1 \Rightarrow i) \\ m < n \mathrel{\dot{\to}} (i - 1 \Rightarrow i) \end{bmatrix} \end{bmatrix}
\end{array}$$

The iteration variable is sometimes denoted by e and sometimes i. When nesting loops, every loop naturally must its own unique loop variable, but Zuse does not make clear how this would be accomplished. In an implementation this problem will need to be adressed. In Plankalkül 2000, nested loops are numbered $W0_0$, $W0_1$, etc.[13] The iteration variables belonging to each is numbered similarly i_0, i_1. Another possiblity would be to allow a programmer to choose a variable himself.

Iterate while $i < n$

[13] Rojas (2000), p.12

$[W_0^I]$	$\langle W_0(n)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}[i{\mapsto}0]$	if s i = undefined
$[W_0^{II}]$	$\langle W_0\ (n)\ [S_0],\ s\rangle \to s$	if s $i \geq \mathcal{A}[\![n]\!]$s
$[W_0^{III}]$	$\dfrac{\langle \mathrm{S},\mathrm{s}\rangle \to s',\ \langle W_0\ (n)\ [S_0],\ s'[i \mapsto \mathrm{s}\ i+1]\rangle \to s''}{\langle W_0\ (n)\ [S_0],\ \mathrm{s}\rangle \to s''}$	if s $i < \mathcal{A}[\![n]\!]$s

Iterate while $i < n$

$[W_1^I]$	$\langle W_1(n)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}[i{\mapsto}0]$	if s i = undefined
$[W_1^{II}]$	$\langle W_1(n)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}$	if s $i \geq \mathcal{A}[\![n]\!]$s
$[W_1^{III}]$	$\dfrac{\langle S_0,\mathrm{s}\rangle \to s',\ \langle W_1(n)[S_0],s'[i{\mapsto}\mathrm{s}\ i{+}1]\rangle \to s''}{\langle W_1(n)\ [S_0],\ \mathrm{s}\rangle \to s''}$	if s $i < \mathcal{A}[\![n]\!]$s

The constructs W0 and W1 are equivalent according to this semantics. Presumably Zuse meant the iteration variable for W0 not to be accessible within the loop, in contrast to all other statements. This is contrary to his descriptions on how variables are declares (always local) and he does not describe any way in which to accomplish this. As a result it is very difficult to realise in the semantics and in the end both have been left semantically equivalent.

Iterate while $i > 0$

$[W_2^I]$	$\langle W_2(n)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}[i{\mapsto}\mathcal{A}[\![n]\!]\mathrm{s}]$	if s i = undefined
$[W_2^{II}]$	$\langle W_2(n)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}$	if s $i \leq 0$
$[W_2^{III}]$	$\dfrac{\langle S_0,\mathrm{s}\rangle \to \mathrm{s}',\ \langle W_2(n)[S_0],\mathrm{s}'[i{\mapsto}(\mathrm{s}\ i){-}1]\rangle \to \mathrm{s}''}{\langle W_2(n)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}''}$	if s $i > 0$

Loops with two arguments:

Iterate while $n < m$

$[W_3^I]$	$\langle W_3(n,m)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}[i{\mapsto}\mathcal{A}[\![n]\!]\mathrm{s}]$	if s i = undefined
$[W_3^{II}]$	$\langle W_3(n,m)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}$	if s $i \geq \mathcal{A}[\![n]\!]$s
$[W_3^{III}]$	$\dfrac{\langle S_0,\mathrm{s}\rangle \to s',\ \langle W_3(n',m)\ [S_0],\ s'[i{\mapsto}\mathrm{s}\ i{+}1]\rangle \to \mathrm{s}''}{\langle W_3(n,m)\ [S_0],\ \mathrm{s}\rangle \to s''}$	if s $i < \mathcal{A}[\![m]\!]$s

Iterate while $m < n$

$[W_4^I]$	$\langle W_4(n,m)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}[i{\mapsto}\mathcal{A}[\![n]\!]\mathrm{s}]$	if s i = undefined
$[W_4^I]$	$\langle W_4(n,m)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}$	if $si \leq \mathcal{A}[\![n]\!]$s
$[W_4^{III}]$	$\dfrac{\langle S_0,\mathrm{s}\rangle \to s',\ \langle W_4(n',\mathrm{m})\ [S_0],\ s'[i{\mapsto}\mathrm{s}\ i{-}1]\rangle \to s''}{\langle W_4(n,m)\ [S_0],\ \mathrm{s}\rangle \to s''}$	if s $i > \mathcal{A}[\![m]\!]$s

Iterate while $i \neq m$

$[W_5^I]$	$\langle W_5(n,m)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}[i \mapsto \mathcal{A}[\![n]\!]\mathrm{s}]$	if s i = undefined
$[W_5^I]$	$\langle W_5(n,m)\ [S_0],\ \mathrm{s}\rangle \to \mathrm{s}$	if $\mathcal{A}[\![m]\!]\mathrm{s} = \mathcal{A}[\![n]\!]\mathrm{s}$
$[W_5^{III}]$	$\dfrac{\langle S_0,\mathrm{s}\rangle \to s',\ \langle W_5(n',\mathrm{m})\ [S_0],\ s'[i \mapsto \mathrm{s}\ i{+}1]\rangle \to s''}{\langle W_5(n,m)\ [S_0],\ \mathrm{s}\rangle \to s''}$	if $\mathcal{A}[\![m]\!]\mathrm{s} > \mathcal{A}[\![n]\!]\mathrm{s}$
$[W_5^{IV}]$	$\dfrac{\langle S_0,\mathrm{s}\rangle \to s',\ \langle W_5(n',\mathrm{m})\ [S_0],\ s'[i \mapsto \mathrm{s}\ i{-}1]\rangle \to s''}{\langle W_5(n,m)\ [S_0],\ \mathrm{s}\rangle \to s''}$	if $\mathcal{A}[\![m]\!]\mathrm{s} < \mathcal{A}[\![n]\!]\mathrm{s}$

This concludes the semantics of Plankalkül.

4 Application

A formal semantics has several uses. It can be used to prove or disprove the equivalence of different statements of Plankalkül, or it can be used to test an implementation, like Plankalkül 2000. It can be used towards generating automatic proofs. In itself it shows a few important properties of Plankalkül. The semantic of Plankalkül is equivalent to other minimal languages, such as the While from Nielson & Nielson (1999) and Edsger Dijkstra's Guarded Command Language introduced in Dijkstra (1975). Plankalkül may have an unfamiliar notation and a collection of features found in no other programming language since, but it is equivalent to every major and modern language. Everything that can be accomplished in a modern programming language, can theoretically be done in a minimal formal language. Everything that can theoretically be done in a modern programming language can be done in Plankalkül.

Another good way to use a formalized semantics is to test an implementation. The first implementation of Plankalkül was made at the University of Berlin in 2000.[14] This implementation written in Java has a linearised version of the syntax and a simulation of memory state after the execution of a plan.[15]. Unfortunately the implementation is not perfect, and the looping constructs do not seem to function. Even the simple examples from the implementation document containing a loop do so.[16] So unfortunately the implementation is not as complete as the document suggests, or some new errors have been introduced later. The Berlin implementation uses a subset of Plankalkül, and an adapted syntax. There is no way to chain different plans together, instead every plan is completely independent of all others. The mathematical symbols have been changed to symbols which are more suited to a modern keyboard and the notation for variables has been changed to be one-dimensional.

V0[1:5.0]	Variable V_0, component 1, of type 5.0
Z1[5.3 :9.0]	Variable Z_0, component 3 of component 5, of type 9.0 [17]
Z2[:12.3.0]	Variable Z_2, of type 12.3.0

Unfortunatly due to the unavailability of looping construct, no programs except very trivial ones can be compiled. A verification using the semantics would not be very interesting. Fortunatly the semantics rules I have developed in this paper can also be translated into a computer language.

[14] Rojas (2000), Rojas(2000a)
[15] It can be found on the internet at `http://www.zib.de/zuse/home/Programs/PlancalculCompiler`
[16] Rojas (2000a), for instance example P5 on p. 16.
[17] Rojas (2000), p.12.

With a simple parser, a compiler or interpreter of Plankalkül can be constructed, the latter of which I have done. Translating these rules to a functional programming language like Haskell can be done in a fairly intuitive way a few examples of which are given below.

4.1 Boolean Expressions

$$
\begin{array}{rcl}
\mathcal{B}[\![\bar{b}]\!]\mathrm{s} & = & \begin{cases} \mathrm{tt} \text{ if } \mathcal{B}[\![b]\!]\mathrm{s} = \mathrm{ff} \\ \mathrm{ff} \text{ if } \mathcal{B}[\![b]\!]\mathrm{s} = \mathrm{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]\mathrm{s} & = & \begin{cases} \mathrm{tt} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \\ \mathrm{ff} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ or } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \end{cases} \\
\mathcal{B}[\![b_1 \vee b_2]\!]\mathrm{s} & = & \begin{cases} \mathrm{ff} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{ff} \text{ or } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{ff} \\ \mathrm{tt} \text{ if } \mathcal{B}[\![b_1]\!]\mathrm{s} = \mathrm{tt} \text{ and } \mathcal{B}[\![b_2]\!]\mathrm{s} = \mathrm{tt} \end{cases}
\end{array}
$$

In the semantics of this paper, a boolean expression is given a value by function $\mathcal{B}$. Likewise in the haskell code, the function bexp determines which rule is appropriate for a given boolean expression. Each rule is a translation of the semantics rules above.

```
-- bexp: determines which rule must be used for a boolean expression
bexp :: ParseTree -> State -> Value
...
bexp (Branch "!"   (a:[])  ) state = zuseNot (aexp a state) (aexp b state)
bexp (Branch "/\\" (a:b:[])) state = zuseAnd (bexp a state) (bexp b state)
bexp (Branch "\\/" (a:b:[])) state = zuseOr  (bexp a state) (bexp b state)
...

-- zuseNot: flip a boolean value
zuseNot :: Value -> Value
zuseNot v =
  if v == tt then ff
  else if v == ff then tt
       else error "zuseNot: not a boolean value"

-- zuseAnd: implements a boolean and
zuseAnd :: Value -> Value -> Value
zuseAnd v1 v2 =
  if (v1 == tt) then
    if (v2 == tt) then tt
else if (v2 == ff) then ff
     else error "zuseAnd: not a boolean value"
  else if (v1 == ff) then ff
       else error "zuseAnd: not a boolean value"

--zuseOr: implements a boolean or (inclusive)
zuseOr :: Value -> Value -> Value
zuseOr v1 v2 =
```

```
  if (v1 == ff) then
    if (v2 == ff) then ff
else if (v2 == tt) then tt
     else error "zuseAnd: not a boolean value"
  else if (v1 == tt) then tt
       else error "zuseAnd: not a boolean value"
```

4.2 Arithmetic Expressions

$$\begin{array}{lcl} \mathcal{A}[\![a_0 + a_1]\!]\mathrm{s} & = & \mathcal{A}[\![a_0]\!]\mathrm{s} + \mathcal{A}[\![a_1]\!]\mathrm{s} \\ \mathcal{A}[\![a_0 \times a_1]\!]\mathrm{s} & = & \mathcal{A}[\![a_0]\!]\mathrm{s} \times \mathcal{A}[\![a_1]\!]\mathrm{s} \\ \mathcal{A}[\![a_0 \text{ - } a_1]\!]\mathrm{s} & = & \mathcal{A}[\![a_0]\!]\mathrm{s} \text{ - } \mathcal{A}[\![a_1]\!]\mathrm{s} \end{array}$$

Arithmatic expression are mapped to a value with a function $\mathcal{A}$. In the haskell code the function aexp determines which rule is appropriate and each of these rules is a close translation of the semantics above to haskell code.

```
--aexp: determines which rule is appropriate for an arithmatic expression.
aexp :: ParseTree -> State -> Value
...
aexp (Branch "+" (a:b:[])) state  = zuseAdd (aexp a state) (aexp b state) state
aexp (Branch "-" (a:b:[])) state  = zuseSub (aexp a state) (aexp b state) state
aexp (Branch "*" (a:b:[])) state  = zuseMult (aexp a state) (aexp b state) state
...

-- zuseAdd: Add two values together.
zuseAdd :: Value -> Value -> State -> Value
zuseAdd x y state = (A (devalueInt(x) + devalueInt(y)))

--zuseMult: Multiply two values.
zuseMult :: Value -> Value -> State -> Value
zuseMult x y state = (A (devalueInt(x) * devalueInt(y)))

--zuseSub: Subtract one value from another.
zuseSub :: Value -> Value -> State -> Value
zuseSub x y state = (A (devalueInt(x) - devalueInt(y)))
```

4.3 Iteration

Iterate while $i < n$

$$\begin{array}{lll} [W_1^I] & \langle W_1(n)\ [S_0],\ \mathrm{s}\rangle \rightarrow \mathrm{s}[i{\mapsto}0] & \text{if s } i = \text{undefined} \\ [W_1^{II}] & \langle W_1(n)\ [S_0],\ \mathrm{s}\rangle \rightarrow \mathrm{s} & \text{if s } i \geq \mathcal{A}[\![n]\!]\mathrm{s} \\ [W_1^{III}] & \dfrac{\langle S_0,\mathrm{s}\rangle \rightarrow s',\ \langle W_1(n)[S_0],s'[i{\mapsto}\mathrm{s}\ i{+}1]\rangle \rightarrow s''}{\langle W_1(n)\ [S_0],\ \mathrm{s}\rangle \rightarrow s''} & \text{if s } i < \mathcal{A}[\![n]\!]\mathrm{s} \end{array}$$

This looping construct uses a local iteration variable, i, which is accessible from within the loop. The rule 'zuseWiederOne' matches the three semantics rules closely and should be easy to understand even without the exact definition of the used Haskell datastructures.

```
-- plan: apply an expression to a state and return a new state
plan :: ParseTree -> State -> State
...
plan ( Branch "W1" (a:b:[]) ) state
  | (strip a) == "()"                    = zuseWiederOne (aexp a state) b state
  | otherwise                            = state
...

-- Simple one argument loop with a local variable
zuseWiederOne :: Value -> ParseTree -> State -> State
zuseWiederOne n s0 state
  | (s "i" state) == Undef  = zuseWiederOne n s0 (replace "i" n state)
  | (s "i" state) >= n      = state
  | otherwise               = zuseWiederOne n s0 (plan s0 newstate)
  where newstate = (replace "i" (A ((devalueInt (s "i" state)) + 1)) state)
```

4.4 Statements

$[\dot{\rightarrow}_{tt}]$	$\dfrac{\langle S,\mathrm{s}\rangle \rightarrow \mathrm{s}'}{\langle b \dot{\rightarrow} S,\ \mathrm{s}\rangle \rightarrow \mathrm{s}'}$	if $\mathcal{B}[\![b]\!]\mathrm{s} = \mathrm{tt}$
$[\dot{\rightarrow}_{ff}]$	$\langle b \dot{\rightarrow} S,\ \mathrm{s}\rangle \rightarrow \mathrm{s}$	if $\mathcal{B}[\![b]\!]\mathrm{s} = \mathrm{ff}$
$[\mathrm{comp}_1]$	$\dfrac{\langle S_1,\ \mathrm{s}\rangle \rightarrow \mathrm{s}',\ \langle S_2,\mathrm{s}'\rangle \rightarrow \mathrm{s}''}{\langle S_1 \mid S_2,\ \mathrm{s}\rangle \rightarrow \mathrm{s}''}$	

Statements are functions from one state to another. The function *plan* applies a statement to a state and returns a new state. The entire program is applied step-by-step to an initial empty state. The resulting state represents the memory state of the computer after running the program.

```
-- plan: apply an expression to a state and return a new state
plan :: ParseTree -> State -> State
...
plan (Branch "|"  (a:b:[]) ) state       = plan b (plan a state)
plan (Branch "->" (a:b:[]) ) state
  |  (strip a) == "tt"                   = plan b state
  |  (strip a) == "ff"                   = state
  |  (bexp a state) == tt                = plan b state
  |  otherwise                           = state
...
```

The syntax I have chosen to implement is based on Plankalkül 2000, but in some cases a syntax more closely resembling Zuse's original syntax is supported as well. The the resulting interpreter is an ongoing project. All the translated rules, and code can be found at `http://zuse.wanzin.nl`.

5 Conclusion

Plankalkül may have been conceived twenty-five years before the first high-level programming language was implemented but it was hardly a theoretical exercise. It is made up of many of the familiar constructs of modern imperative languages, and we find some constructs more akin to functional languages. None of its features are uncommon and a formal semantics can be given without much difficulty. We can conclude that Plankalkül is in no fundamental way different from modern languages.

Zuse was foremost an engineer, not a theoretical researcher, and there is little doubt that Plankalkül could have been implemented and used. Unfortunately it was published in the wrong circumstances, years before the need for higher-level languages was widely spread. In 1945 all programs were low-level, sometimes requiring a complete rewirering of the computer. Despite Zuse's efforts to gain public awareness for his language, most if not all the attention has gone to his computers, his language usually ending up as no more than a footnote.

Although many of Konrad Zuse's projects and accomplishments have remained far from public attention, this is changing. Zuse is gaining increasing recognition as the inventor of the Z3, the first Turing complete computer. It is likely that his contribution to the software side of computer science will slowly gain more public attention in the years to come.

6 Bibliography

Bauer & Woessner (1972) F.L, Bauer and H. Woessner, *The "Plankalkül" of Konrad Zuse: A Forerunner of Today's Programming Languages*, Communications of the ACM, Volume 15, Number 7, July 1972.

Dijkstra (1975) Edsger Dijkstra, *Guarded commands, Nondeterminacy and Formal derivation of Programs,* Association for Computing Machinery, Inc., 1975.

Giloi (1997) Wolfgang K. Giloi, *Konrad Zuse's Plankalkül: The First High-Level, "non von Neumann" programming language*, IEEE Annals of the History of Computing, Vol. 19, No. 2, 1997.

Hoare (1969) C.A.R. Hoare, *An Axiomatic Basis for Computer Languages*, Communications of the ACM, volume 12, number 10, October 1969.

Nielson & Nielson (1999) Hanne Riis Nielson and Flemming Nielson, *Semantics with Applications, A Formal Introduction*, `http://www.daimi.au.dk/~hrn`, 1999.

Rojas (1998) Raúl Rojas, *How to make Zuse's Z3 a universal computer*, IEEE Annals of the History of Computing 20 (3), 1998.

Rojas (2000) Raúl Rojas, Cüneyt Göktekin, Gerald Friedland, Mike Kruger, Ludmila Scharf, *Konrad Zuses Plankalkül - Seine Genese und einde moderne Implementierung*, Freie Universität Berlin, 2000.

Rojas (2000a) Raúl Rojas, Cüneyt Göktekin, Gerald Friedland, Mike Kruger, *Plankalkül: The First High-Level Programming Language and its Implementation*, Freie Universität Berlin, 2000.

Strachey (2000) Christopher Strachey, *Fundamental Concept in Programming Languages*, Higher-Order and Symbolic Computation, 13, 11-49, Kluwer Academic Publishers, 2000.

Zuse (1945) K. Zuse, *Der Plankalkül (In der Fassung vond 1945)*, Konrad Zuse Internet Archiven, `http://www.zib.de/zuse`, 1945.

Zuse (1959) K. Zuse, *Über den Plankalkül*, Elektron. Rechenal. 1 (1959) Bad Hersfeld, 1959.