Honours Year Project Report

Semantics and Implementation of the Plankalkül

By

Jimmo Barrion Petisme

Department of Computer Science

School of Computing

National University of Singapore

2003/04

Honours Year Project Report

Semantics and Implementation of the Plankalkül

By

Jimmo Barrion Petisme

Department of Computer Science

School of Computing

National University of Singapore

2003/04

Project No: H41080

Advisor: Asst. Prof. Martin Henz

Deliverables:

Report: 1 Volume

Program: 1 Disk

Abstract

The semantics of the Plankalkül, the first high-level programming language, is investigated in this project. This is done through a linear representation of the language which is called *linPlank*. At the same time, the bit-centric data model of the Plankalkül is studied and a simple type system is formulated. The semantics and data model serve as a base for an implementation of *linPlank*. The implementation highlights a number of interesting aspects and shortcomings of the Plankalkül. The semantics and implementation developed here allow for a detailed assessment of the historical significance of the language.

Subject Descriptors:

D.3.3 Language Constructs and Features

F.3.2 Semantics of Programming Languages

F.3.3 Studies of Program Constructs

K.2 History of Computing

Keywords:

Semantics of the Plankalkül, Implementation of the Plankalkül

Acknowledgements

First of all, I would like to thank God for giving me an opportunity to prove myself through this project. He has given me a lot of courage and support, and I know that's He's always there watching.

I would also like to thank my mother, who is constantly praying for my health and safety, and my father, who's probably having fun Up There. Without them, I won't exist.

Next I would like to thank Dr Martin Henz, my very knowledgeable supervisor. Even though he is very busy with academic responsibilities as well as his role in FriarTuck Corporation, he still manages to help me with the best that he could.

Then I would like to thank Yuliana Wiyono for her encouragement and countless support througout the completion of this thesis.

Last but not the least, I would like to thank the following: The 6th Management Committee of the NUS Students' Computing Club, for understanding that I might not be able to fully perform my responsibilities due to this project; the Academic Liaison Committee for being such understanding and supportive people; the NUS Filipino Community – you're the best gang ever; and of course Keppel Offshore and Marine Ltd, for giving me an opportunity to study here in the wonderful world of NUS.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Konrad Zuse and the History of Computing

Konrad Zuse (1910-1995) is well-known as one of the pioneers of computing in Germany. He is most attributed to the development of the computers Z1 to Z4, sophisticated machines that can be said to be quite ahead of his time back then. Being a civil engineer, he started building computing machines in order to ease him from the burden of manually solving tons of tedious equations – a motivation that had shaped his whole career.

The Z1 to Z4 computers, which were each developed in sequence between 1936 and 1945, made use of binary digits, which is similar to the representation of data in modern computers. In fact, the architecture of these computers were very similar to the modular component architecture of modern processors, except for the fact that the latter are larger. Additionally, the Z1 and Z2 computers made use of mechanical contraptions to implement storage, while the Z3 and Z4 made use of a combination of mechanical storage and relays. However, during World War II, the machines, along with the design papers, were destroyed, except for the Z4, which was transfered to the small town of Hinterstein in 1945. After the war, the Z4 was used in the Institute of Applied Mathematics at the Eidgenössische Technische Hochschule (ETH) at Zürich, Switzerland, and it was Heinz Rutishauser, one of the pioneers of Algol, who programmed it (Zuse, 1980, n.d.; Bauer, 1980).

It is quite astonishing to note that Zuse made these developments without knowledge of the happenings in other countries, not even in Germany itself. He had not known of the earlier work of Charles

Babbage until some time later, nor had he heard of the research of John von Neumann and others during that time.

1.2 The Plankalkül

1.2.1 Background

During World War II, when Konrad Zuse took refuge in Hinterstein, along with his pregnant wife and the not-yet-finished Z4, he took the time to write some papers regarding his theory of computation. Two of these are the "Statements of a Theory of General Calculation" and "The Plankalkül". The former described the use of binary numbers to represent different kinds of data, as well as the Calculus of Propositions (also known as Propositional Logic) and its use to conduct more complicated operations like arithmetic and algebra. The latter, on the other hand, described a way to represent algorithms or, in Zuse's terms, "plans for calculation" (Bauer, 1980; Zuse, n.d., 1980, 1989).

1.2.2 Features

The Plankalkül is disputably the first high-level programming language ever designed. It had most of the features of a modern procedural language like C or Pascal, except that pointers, global variables and recursive functions are not supported. Programs written in the Plankalkül can be said to be black boxes that accept some input values and return some output values. Take for example the following program, taken from Zuse (1989):

P1.72	V	$\leq$	V	$\Rightarrow$	R
V	0		1		0
S	1.n		1.n		0

	W1(n)	V	$\Rightarrow$	Z	V	$\nsim$	V	$\Rightarrow$	Fin^2	$\overline{\text{Z}}$	$\Rightarrow$	R
V		0		0	0		1			0		0
K		i			i		i					
S		0		0	0		0			0		0

The program above describes the procedure for testing whether V0 $\leq$ V1 is true or false. At first glance, one may be overwhelmed by the matrix-like notation used. However, this two-dimensional

syntax is supposedly for ease of reading, and every Plankalkül program can be rewritten using a linear representation, as this project will demonstrate.

Aside from this peculiar feature, the Plankalkül is also able to perform Predicate Logic operations, such as checking whether all possible values for a variable that satisfies a particular condition, and list operations, such as collecting all the possible values for a variable that satisfies a particular condition in a list. In his book "The Plankalkül", Zuse demonstrated the power of the language by writing a chess-playing program. Also, because of these features, Giloi suggested in his paper (1997) that the Plankalkül had the expressibility the language APL, or even Lisp, had.

Despite all these powerful features, the Plankalkül was not embraced in its entirety; nor had the ideas revolving around it made its way into Algol, even though Rutishauser and some others involved in the project knew of the Plankalkül.

1.2.3 Status and Issues

Scarce Documentation

Most of the material about the Plankalkül is written in German. However, an English translation of "The Plankalkül" has been published by the Germany-based Gessellschaft für Mathematik und Datenverarbeitung (GMD) in 1989. It is to be noted that this translation uses a lot of ambiguous wording, and decyphering the statements in the book can cause headaches.

A few papers were written about the language. In Bauer and Wössner (1972), the various features of the Plankalkül are highlighted, and translations of some of the programs into Algol 68 are presented. Giloi (1997), already mentioned in the previous section, assesses the features of the Plankalkül and relates it to both von Neumann and non-von Neumann languages. In addition, Knuth and Trabb Pardo (1980) explores the language along with other programming languages conceptualized between 1945 and 1958. They compared each of the languages by writing their chosen algorithm – which they aptly named the TPK algorithm – in each of the languages.

Unusual Two-Dimensional Syntax

One of the issues hindering the development of an implementation of the Plankalkül is the unusual syntax used, which used multiple lines to express the different elements of the language. Until now, the standard input and output of a computer is still based on line feed, meaning that characters are still read

line by line and it is not possible to read them vertically without storing them in a large buffer. However, it is noticeable that the Plankalkül's syntax seems to be of a visual nature. With the improvement of the graphics capabilities of computers and the increasingly establishing studies on GUIs, using visual editors to write Plankalkül programs is a very feasible approach.

Zuse (1989) also mentions that in order to realize an implementation of the Plankalkül, it has to be converted into a linear form first. And indeed, even though the language's syntax is not mainstream, it can be observed that parse trees can still be constructed from it, which is typical of context-free grammars. Thus, any program written in the Plankalkül can be rewritten using a linear representation.

Recognition for Zuse's Works

Zuse was more focused on the hardware aspects of his endeavors rather than the software. He founded a company called Zuse KG which is the first computer company in Germany, and through this company he was able to design different computers. Their first task was the restoration of the Z4 and its delivery to ETH Zürich.

Actually, he considered implementing the Plankalkül in one of his computers, but due to the limited resources (monetary and human) he decided not to (Giloi, 1997; Zuse, 1989). And so, for a long period of time, the pieces of paper containing the specifications of the language was kept in Zuse's drawer.

On another note, in (Zuse, 1989), it can be seen that Zuse wanted recognition for his work on the Plankalkül. He was bitter over his claim that Rutishauser, Bauer and the others involved in the birth of Algol who knew of his work ignored the concepts contained in the Plankalkül. He wrote:

> The indifference towards the PK [Plankalkül] was somewhat disappointing for me, when the official discussions about Algol started in 1955. Some of the participants had sufficient knowledge of the PK to cooperate and in my opinion it would only have been fair if they had openly announced and utilized the ideas anticipated in the PK.

In (Bauer, 1980), more statements by Zuse are mentioned, which are translated here from German for convenience.

> The languages Fortran and Algol were developed around 1955. These were also primarily geared towards numerical calculations, but otherwise correspond to the Plankalkül. It is hard to determine today, to what extent ideas of the Plankalkül were adopted.

And in another statement, Zuse said:

> I myself was at that time busy building the company and only occasionally could have conversations with the creators of Algol, such as Rutishauser and Bauer. Most often we were talking at cross purposes. The basic idea of the Plankalkül – to build a programming language systematically from the ground up – seemed excessive or was perceived as a burden.

Bauer then rebutted:

> The Algol procedure concept, the type concept with specifications, the assignment concept was conceptually based on similar ideas in the Plankalkül. Quite frequently, this was done by the European Algol members subconsciously.

And also, he said (with omissions):

> Certainly, for numerical work, and this was mainly the purpose of Algol 58, there was neither a need nor an advantage for describing algorithms down to the bit. Moreover, the Plankalkül had a horrible notation, absolutely unsuited for the input to a compiler. . . Moreover, Rutishauser had already in 1952 found the convenient notation for a for-clause, which is lacking in the Plankalkül together with many other helpful notational devices. Conceptually the Plankalkül has had its maximal influence on the European Algol development; fortunately for Algol, it has left no scars on Algol notationally.

And by this, Bauer concludes that the Plankalkül had actually influenced the history of programming languages through Algol.

After the death of Zuse in 1995, his son Horst, along with a group of people headed by Raúl Rojas, spread his legacy though the Internet. An Internet archive of Zuse's works can be found in `http://www.zib.de/zuse`. Also, Horst Zuse wrote a biography of his father, which can be found in Zuse (n.d.).

1.2.4 The 2000 Implementation

Rojas et al. (2000) mentions the development of what can be said as the first implementation of the Plankalkül. There were two parts to their implementation: a visual editor which allowed the user to write the program using the two-dimensional Plankalkül syntax and then translates it into a linear representation, and an implementation of the linear representation of the language. Furthermore, the latter is implemented as a compiler which emits virtual machine code that they devised. However, they have only

implemented a subset of the language, without support for the predicate calculus and list operations, as well as support for variable-sized lists.

It is to be noted that they had made some assumptions in the development of their implementation. To implement a simple type system, they required that all variables have their type declared, and that these variables cannot be casted between types. In addition, in case there are multiple ways to represent components of the language, e.g. bits, they only chose one. However, their implementation has not been totally consistent with the original syntax of the Plankalkül, as they have represented the notation for list structures differently (n.0 as opposed to the original 1.n for the an n-sized list of bits and n.σ as opposed to n $\times$ σ for an n-sized list of data with type σ).

1.3 Objectives of this Project

As a consequence of its lack of acceptance, not a lot of information on the Plankalkül is available. A formal semantics of the language is needed to thouroughly assess its significance in the history of computing. Only with such a formal approach can we be able to verify or falsify Zuse's claims, which, unfortunately, was not done by him. This project attempts to fill this gap.

This project investigates the semantics of a subset of the Plankalkül, which contains most of the its procedural language features, in Chapter 2. This is achieved through a linear representation called *linPlank*, whose syntax is claimed to be in one-to-one correspondence with the syntax of the Plankalkül. This linear representation is consistent with the syntax and semantics of the Plankalkül, and by deriving its semantics, the semantics of the Plankalkül can be similarly conceptualized. In Chapter 3, a simple type system for *linPlank* will be presented. An implementation of *linPlank* shall be discussed in Chapter 4, and issues regarding the development of the implementation will be addressed. Finally, in the concluding Chapter 5, the significance of the Plankalkül will be assessed in light of the new discoveries presented in the thesis. In addition, the limitations of the project will be highlighted, and details of what is needed to be done to conclude the historical assessment of the Plankalkül will be given.

Chapter 2

A Semantics for the Plankalkül through *linPlank*

2.1 *linPlank*– The Linear Representation of the Plankalkül

2.1.1 Preliminaries

As mentioned in the previous chapter, one of the problems in the development of an implementation of the Plankalkül is its two-dimensional syntax. Konrad Zuse himself mentioned in The Plankalkül (1989) that if an implementation of the Plankalkül is to be undertaken, there is a need to design an intermediate language which is linear in nature. And indeed, as (Rojas et al., 2000) proposes, an implementation would require two parts: an "editor" which translates raw Plankalkül syntax to the linear representation, and the core implementation of the linear language.

In this section *linPlank*, my proposed linear representation of the Plankalkül, is presented. In designing this language, certain factors need to be carefully considered. First of all, it should closely resemble the raw Plankalkül syntax. Also, the semantics of the language should be similar to that of the Plankalkül. Thus, it should be sufficient to provide the semantics of *linPlank* in order to get an idea of what programs written in the Plankalkül would mean.

2.1.2 Rewriting Plankalkül Programs in *linPlank*

The two-dimensional nature of the Plankalkül is due to the representation of variables. Consider the following notation:

	V	$\wedge$	Z	$\Rightarrow$	R
V	0		0		0
K	0				
S	1.n		0		0

The last three lines are labeled V, K, and S, which signify the address, component index, and structure of the variable or expression respectively. In *linPlank*, the statement above is written as:

```
V0[:0:]:1.n & Z0:0 => R0:0;
```

The V component is put beside the variable name (either V, Z, R or i, see later sections), the K component is enclosed with square brackets and colons, and the S component follows a colon. In case a component is not provided in the Plankalkül notation, this component, along with its delimiters, will also not be present in the *linPlank* representation. As we go along, the various features of the Plankalkül and how they are translated to *linPlank* notation will be presented.

2.1.3 Methodology in Defining the Semantics of *linPlank*

There are different methods in presenting the semantics of programming languages. The three most established methods are Axiomatic Semantics, Operational Semantics and Denotational Semantics. Readers are referred to Schmidt (1988) to know the differences between the three.

Presenting the semantics of *linPlank* using Denotational Semantics is chosen, mainly because of its expressive power and the ease of developing an implementation based on it. The denotational semantics of *linPlank* will be presented using the notations of Stoy (1977) and Schmidt (1988).

The semantics will be presented using an incremental approach, meaning that the presentation will start from a minimal subset of the language, gradually adding features through every iteration. This way the reader can grasp the concepts easily, just like how a baby has to be fed smaller spoonfuls so that he can easily digest the food.

2.2 *linPlank0* - A Bit-Calculator Language

2.2.1 Syntax

linPlank0 is a simple language, only supporting bit values and types (denoted ”S0” in the Plankalkül), as well as boolean expressions and assignment and conditional statements. Also, programs written in this language subset will only have one subprogram, that is, the main subprogram, as this subset does not have support for subprogram calls.

Like the Plankalkül, variables in *linPlank0* are also classified as input (V), output (R), or local (Z) variables. Right now, they can only have bit values, meaning they can only have the structure (or type, in modern terminology) S0.

In the Plankalkül, there are two ways of representing bit values. One way is by using the symbols “$+$” for a true value and “$-$” for a false value, and the other is by using the symbols “L” for a true value and “o” for a false value. The latter is used, as it is more similar to modern bit representations. However, this similarity will only be evident later, when bit strings are already supported.

Assignment statements in the Plankalkül take the form $E \Rightarrow V$, where E is an expression and V is a variable. Meanwhile, conditional statements take the form $E \underset{\cdot}{\rightarrow} S$ where E is an expression and S is a statement. Since the symbols used in the notations are not supported in standard ASCII, *linPlank0* uses the symbols `=>` and `-.>` instead of $\Rightarrow$ and $\underset{\cdot}{\rightarrow}$ respectively.

There are six boolean operations supported in the Plankalkül, which are conjunction (AND, $\wedge$), disjuction (OR, $\vee$), negation (NOT, $\neg$, or $\overline{E}$, where E is an expression), implication (IMPLIES, $\rightarrow$), bit equality (XNOR, $\sim$), and bit inequality (XOR, $\nsim$). However, in *linPlank0*, the symbols `&`, `|`, `!`, `->`, `~`, and `!~` are used for the respective operations.

To summarize, below is the abstract syntax definition for *linPlank0*:

- $P \in$ Programs

  $Sp \in$ Subprograms

  $fid, idx \in \mathbb{N}$

  $I \in$ InputVariableLists

  $O \in$ OutputVariableLists

  $S \in$ Statements

  $E \in$ Expressions

  $Var \in$ Variables

$V \in$ InputVariables

$R \in$ OutputVariables

$Z \in$ LocalVariables

$B \in \{\texttt{0}, \texttt{L}\}$

- $P \longrightarrow Sp$

  $Sp \longrightarrow \texttt{P}\ fid\ \texttt{R}(I)\ \texttt{=>}\ (O)\ [S]$

  $I \longrightarrow V, I \mid V$

  $O \longrightarrow R, O \mid R$

  $S \longrightarrow E\ \texttt{=>}\ Var \mid E\ \texttt{-.>}\ [S] \mid S_1; S_2$

  $E \longrightarrow B \mid Var \mid E_1 \texttt{\&} E_2 \mid E_1 \texttt{|} E_2 \mid \texttt{!}\, E \mid E_1 \texttt{->} E_2 \mid E_1 \texttt{\~{}} E_2 \mid E_1 \texttt{!\~{}} E_2 \mid (E)$

  $Var \longrightarrow V \mid R \mid Z$

  $V \longrightarrow \texttt{V}\ idx$

  $R \longrightarrow \texttt{R}\ idx$

  $Z \longrightarrow \texttt{Z}\ idx$

Notice that the statement inside the conditional in the definition above is enclosed with square brackets. This signifies that this statement is contained in a block in the abstract syntax. This is implied in Zuse (1989). For now, it is sufficient to define the block as a group of statements. However in the next subset of the language, its purpose will be revealed.

2.2.2 The Semantic Framework

After defining the syntax, the semantic framework shall now be laid out. The framework is presented by first introducing the semantic domains, defining their nature and identifying operations between elements of these said domains. After that the semantic functions will be defined over the syntactic elements of *linPlank0*.

Semantic Domains

Basic Values As *linPlank0* only supports bit values, the only basic values defined for this language subset are bit values. In the semantic framework, these bit values can have the value 0 or 1. Over this domain, six operations are defined, namely *and*, *or*, *not*, *implies*, *xnor* and *xor*. As these operations are the standard bit operations, their behavior will not be explicitly defined.

- Domain: $DV = BV + \{\perp\}$
- Domain: $BV = \{0, 1\}$
  Operations:
  - $and : BV \times BV \to BV$
  - $or : BV \times BV \to BV$
  - $not : BV \to BV$
  - $implies : BV \times BV \to BV$
  - $xnor : BV \times BV \to BV$
  - $xor : BV \times BV \to BV$

Notice that $\{\perp\}$ is included in the definition of the domain of denotable values (*DV*). This is the set containing the error value $\perp$. We define that all operations involving denotable values are irrecoverable, meaning that if any one of the inputs to a particular operation is the error value, the result is still the error value.

Stores Stores are the representations of the computer memory allocation system. For simplicity of implementation, the store in *linPlank0* is defined as a triple, consisiting of three substores, one for each of the types of variables. These substores, usually represented using the greek characters ν, ρ and ζ for the input, output and local stores respectively, each have two operations: $\sigma[n \mapsto v]$, where the store σ is extended with a mapping for the location n to the value v; and $\sigma(n)$, where the value mapped to the location n in store σ is returned. Also, the symbol $\varnothing$ represents an empty store.

- Domain: $VSto = (\mathbb{N} \to DV) + \{\perp\}$
  Operations:
  - $\varnothing : VSto$
  - $\mapsto$: $VSto \to \mathbb{N} \to DV \to VSto$
- Domain: $RSto = (\mathbb{N} \to DV) + \{\perp\}$
  Operations:
  - $\varnothing : RSto$

  - $\mapsto$: $RSto \to \mathbb{N} \to DV \to RSto$

- Domain: $ZSto = (\mathbb{N} \to DV) + \{\bot\}$

  Operations:

  - $\varnothing : ZSto$
  - $\mapsto$: $ZSto \to \mathbb{N} \to DV \to ZSto$

- Domain: $Sto = VSto \times RSto \times ZSto$

For store access operations, the error value $\bot$ is returned if there is no value stored under the given index. On the other hand, if a store is updated with an error value, the error value is returned. Again, operations involving error values are irrecoverable.

Semantic Functions

The semantic functions for *linPlank0* will now be defined.

- $\mathcal{P}$: Program $\longrightarrow DV \times DV \times \cdot \times DV \longrightarrow DV \times DV \times \cdot \times DV$
  - $\mathcal{P}[\![Sp]\!](\alpha_1, \alpha_2, \ldots, \alpha_n) = \mathcal{S}[\![Sp]\!](\alpha_1, \alpha_2, \ldots, \alpha_n)$
- $\mathcal{S}$: Subprogram $\longrightarrow DV \times DV \times \cdot \times DV \longrightarrow DV \times DV \times \cdot \times DV$
  - $\mathcal{S}[\![\texttt{P}\ fid\ \texttt{R(V0, V1, ..., Vn)=>(R0, R1, ..., Rm)}\ [S]]\!](\alpha_1, \alpha_2, \ldots, \alpha_{n+1}) =$
    $(\zeta'(0), \zeta'(1), \ldots, \zeta'(m))$, where $(\nu', \rho', \zeta') = \mathcal{X}[\![S]\!](\nu, \varnothing, \varnothing)$ and
    $\nu = \varnothing[0 \mapsto \alpha_1][1 \mapsto \alpha_2] \ldots [n \mapsto \alpha_{n+1}]$
- $\mathcal{X}$: Statement $\longrightarrow Sto \longrightarrow Sto$
  - $\mathcal{X}[\![E\texttt{=>V}\ idx]\!](\nu, \rho, \zeta) = (\nu[idx \mapsto \mathcal{E}[\![E]\!](\nu, \rho, \zeta)], \rho, \zeta)$
  - $\mathcal{X}[\![E\texttt{=>R}\ idx]\!](\nu, \rho, \zeta) = (\nu, \rho[idx \mapsto \mathcal{E}[\![E]\!](\nu, \rho, \zeta)], \zeta)$
  - $\mathcal{X}[\![E\texttt{=>Z}\ idx]\!](\nu, \rho, \zeta) = (\nu, \rho, \zeta[idx \mapsto \mathcal{E}[\![E]\!](\nu, \rho, \zeta)])$
  - $\mathcal{X}[\![E\texttt{-.>}[S]]\!]\sigma = \begin{cases} \mathcal{X}[\![S]\!]\sigma & \text{if } \mathcal{E}[\![E]\!]\sigma = 1 \\ \sigma & \text{otherwise} \end{cases}$
  - $\mathcal{X}[\![S_1; S_2]\!]\sigma = \mathcal{X}[\![S_2]\!]\sigma'$, where $\sigma' = \mathcal{X}[\![S_1]\!]\sigma$

- $\mathcal{E}$: Expression $\longrightarrow Sto \longrightarrow DV$
  - $\mathcal{E}[\![B]\!]\sigma = \mathcal{B}[\![B]\!]$
  - $\mathcal{E}[\![\texttt{V}\ idx]\!](\nu, \rho, \zeta) = \nu(idx)$
  - $\mathcal{E}[\![\texttt{R}\ idx]\!](\nu, \rho, \zeta) = \rho(idx)$
  - $\mathcal{E}[\![\texttt{Z}\ idx]\!](\nu, \rho, \zeta) = \zeta(idx)$
  - $\mathcal{E}[\![E_1 \texttt{\&} E_2]\!]\sigma = and(\mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)$
  - $\mathcal{E}[\![E_1 \texttt{|} E_2]\!]\sigma = or(\mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)$
  - $\mathcal{E}[\![\texttt{!}\, E]\!]\sigma = not(\mathcal{E}[\![E]\!]\sigma)$
  - $\mathcal{E}[\![E_1 \texttt{->} E_2]\!]\sigma = implies(\mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)$
  - $\mathcal{E}[\![E_1 \texttt{~} E_2]\!]\sigma = xnor(\mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)$
  - $\mathcal{E}[\![E_1 \texttt{!~} E_2]\!]\sigma = xor(\mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)$

While *linPlank0* only allows for one main subprogram, and the use of two semantic functions to give meaning to this might seem redundant, we use two semantic functions $\mathcal{P}$ and $\mathcal{S}$ in preparation for the extensions presented in the following sections.

An interesting thing to note is the execution of subprograms of a subprogram, represented by the semantic function $\mathcal{S}$. It includes a mapping from each of the input variables to each of the respective input values in an empty store before executing the body of the subprogram. After execution of the body, the values of the output variables in the store are retrieved and returned.

Note that the definitions of $\mathcal{E}$ for the boolean expressions (AND, OR, etc.) use the operations for bit values. While it might be alright at this stage, things will get messy when we introduce tuple values in *linPlank2*, the third subset of the language. By convention, if input values to an operation is not a member of the domain, the error value is returned, as if there is a wrapping function which checks for compatibility. However, it is the job of the type system, which will be conceptualized in Chapter 3, to ensure that input values to an operation are members of the domains in which that operation is defined.

Finally, the semantic function $\mathcal{B}$ translates Plankalkül bit strings (and numeric constants later in *linPlank2*) to their semantic data model representations. It shall not be explicitly defined here.

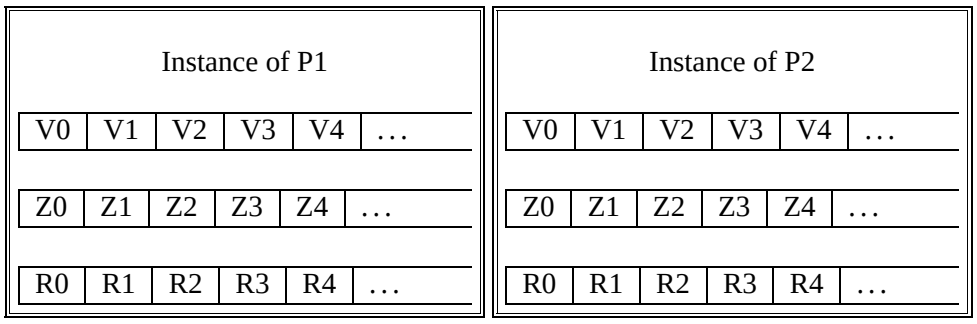

Figure 2.1: Conceptual visualization of stores

2.2.3 Assessment

At this point, it can be seen that the semantics of *linPlank* (and of the Plankalkül) is similar to that of procedural programming languages such as C and Pascal. However, the notion of global variables are not defined in the Plankalkül which is why every instance of the execution of a subprogram will have its own store.

Moreover, while usually the store is defined as mappings from variables to memory locations, the semantics defined in this chapter does not reflect memory management. This is because the scope of all variables are local to a subprogram, and consequently no cross-references to memory locations between subprograms are present. It is sufficient to say that the stores for instances of the execution of subprograms are allocated some non-overlapping region in the memory, and this region is further partitioned to hold values for the different types of variables (see Figure 2.1). However, the semantics defined above can still be rewritten such that memory management is taken into account.

2.3 *linPlank1* - Extending the Language with Loops

2.3.1 Extensions to the Syntax

linPlank1 is the next incremental subset to *linPlank*. The subset adds loops and an instruction to exit blocks, which will also be redefined in this section, and loops.

Loops in *linPlank1* come in the form of $\mathtt{W}[S]$, where S is a statement or a group of statements. Normally S would consist of a groups of conditionals, the last one being the complement of all the

previous cases, as illustrated below (Plankalkül notation on the left, *linPlank* on the right):

$$
\begin{array}{c|c|c}
 & \mathrm{W} & \left(\begin{array}{ccccc} \mathrm{V} & \dot{\rightarrow} & \mathrm{Z} & \Rightarrow & \mathrm{R} \\ 0 & & 0 & & 0 \\ 0 & & 0 & & 0 \end{array}\right) \\
V & & \\
S & & \\
\hline
 & & \left(\begin{array}{ccccc} \mathrm{V} & \dot{\rightarrow} & \mathrm{Z} & \Rightarrow & \mathrm{R} \\ 1 & & 1 & & 0 \\ 0 & & 0 & & \end{array}\right) \\
V & & \\
S & & \\
\hline
 & & \left(\begin{array}{ccccc} \overline{\mathrm{V}} & \wedge & \overline{\mathrm{V}} & \Rightarrow & \mathrm{Fin}^2 \\ 0 & & 1 & & \end{array}\right) \\
V & &
\end{array}
$$

```
W [
    V0:0 -.> Z0:0 => R0:0;
    V1:0 -.> Z1:0 => R0;
    !V0 & !V1 => Fin 2;
]
```

In fact, this last complementing condition may be omitted, and thus an implementation would be required to intelligently add the said statement to the body of a loop.

On the other hand, Fin statements are the instructions to exit blocks and loops. A Fin statement exits the current block. However, the Fin statement can also include a number which indicates the number of blocks it will exit. Fin statements are also used to terminate loops, being used as the executing statement in the last complementing conditional statement inside a loop. It has to be noted that the semantic interpretation of loops is like that two levels of blocks, as illustrated below:

$$\mathtt{W}[S] \Rightarrow [[S][S][S]\ldots],$$

which is the reason why the Fin statement used in in the last complementing conditional statement is of the second degree, i.e. `Fin 2`. Therefore, a `Fin` statement and a `Fin 2` statement within the scope of the loop corresponds to the `continue` and `break` statements of C and Pascal.

Notice also that the last complementing statement inside the loop (such as the previous example) uses a different notation than the conventional conditional statement. This is the conditional Fin statement, a single statement in which the effects of the Fin statement only happens if the condition included is met. One would wonder why a separate definition for a conditional Fin statement is needed. Remember in section 2.2 that the executing statement of a conventional conditional statement is enclosed in a block. Therefore the effect a Fin statement inside that conditional is limited to the scope of that block, i.e. the

statement E`-.>Fin` does not make sense.

However, the exact behavior of the Fin statement is not clear, based alone on what is written in Zuse (1989). It appears that what is decribed above corresponds to the behavior Zuse wanted to imply when he wrote the specification of the Plankalkül. But then, some of the example programs seem to contradict this claimed behavior. To be consistent with Zuse's implied behavior and the one descibed above, those programs, though significant in number, are labeled as incorrect.

One last thing to note is that blocks can appear anywhere. It can be used to group together a number of statements, which is useful when one of the statements is a Fin statement, in which the execution of the Fin statement is limited to the scope of the block. Thus, one can explicitly define control through the use of blocks and Fin statements, similar to the `goto` statement of programming languages such as Pascal and BASIC.

The syntax of *linPlank1* will have the following extensions to the syntax of *linPlank0*:

- $S \longrightarrow \ldots \mid$ `W`$[S] \mid$ `Fin` $idx \mid E$`=>Fin` $idx \mid [S]$

Note that the these syntactic structures are added to the structures for S given is section 2.2 as indicated by the ellipsis. The last production in the above definition represents the general block, as mentioned earlier.

2.3.2 Additions to the Semantic Framework

The *fin* Flag

First of all, there is a need for a mechanism to exit blocks and loops. This is achieved with the use of the *fin* flag. The behavior of the *fin* flag is as follows:

- When a Fin statement is encountered, the *fin* flag is set to the value of the index included with that Fin statement.
- If the *fin* flag is not 0, the statements following the currently executing statement in the same block are not executed.
- At the end of a block, if the *fin* flag is not zero, its value will be decremented by 1.

However, this mechanism, while it is quite intuitive, also takes up space in the store. Thus, there is a need to include the *fin* flag in the definition of the store.

- Domain: $Sto = VSto \times RSto \times ZSto \times \mathbb{N}$

In the semantic function definitions, the *fin* flag is represented by the Greek symbol ϕ.

New Definitions for the Semantic Functions

Reflecting on the points discussed in the sections above, the following are the extensions made to the semantic framework.

- $$\mathcal{X}[\![\texttt{W}[S]]\!](\nu,\rho,\zeta,\phi) = \begin{cases} \mathcal{X}[\![\texttt{W}[S]]\!](\nu',\rho',\zeta',\phi') & \text{if } \phi' = 0 \\ \mathcal{X}[\![\texttt{W}[S]]\!](\nu',\rho',\zeta',\phi'-1) & \text{if } \phi' = 1 \\ (\nu',\rho',\zeta',\phi'-1) & \text{otherwise} \end{cases}$$
  where $(\nu',\rho',\zeta',\phi') = \mathcal{X}[\![S]\!](\nu,\rho,\zeta,\phi)$
- $\mathcal{X}[\![\texttt{Fin } idx]\!](\nu,\rho,\zeta,\phi) = (\nu,\rho,\zeta,idx)$
- $$\mathcal{X}[\![E\texttt{=>Fin } idx]\!](\nu,\rho,\zeta,\phi) = \begin{cases} (\nu,\rho,\zeta,idx) & \text{if } \mathcal{E}[\![E]\!](\nu,\rho,\zeta,\phi) = 1 \\ (\nu,\rho,\zeta,\phi) & \text{otherwise} \end{cases}$$
- $$\mathcal{X}[\![S_1;S_2]\!](\nu,\rho,\zeta,\phi) = \begin{cases} \mathcal{X}[\![S_2]\!](\nu',\rho',\zeta',\phi') & \text{if } \phi' = 0 \\ (\nu',\rho',\zeta',\phi') & \text{otherwise} \end{cases}$$
  where $(\nu',\rho',\zeta',\phi') = \mathcal{X}[\![S_1]\!](\nu,\rho,\zeta,\phi)$
- $$\mathcal{X}[\![[S]]\!](\nu,\rho,\zeta,\phi) = \begin{cases} (\nu',\rho',\zeta',\phi') & \text{if } \phi' = 0 \\ (\nu',\rho',\zeta',\phi'-1) & \text{otherwise} \end{cases}$$
  where $(\nu',\rho',\zeta',\phi') = \mathcal{X}[\![S]\!](\nu,\rho,\zeta,\phi)$
- $$\mathcal{X}[\![E\texttt{-.>}[S]]\!]\sigma = \begin{cases} \mathcal{X}[\![[S]]\!]\sigma & \text{if } \mathcal{E}[\![E]\!]\sigma = 1 \\ \sigma & \text{otherwise} \end{cases}$$
- All other definitions are rewritten to include ϕ, the *fin* flag.

2.3.3 Assessment

It has been seen that the looping constructs of the Plankalkül are similar to that of procedural programming languages such as C and Pascal. However, the general loop defined in this section is semantically

defined to loop forever, only stopping when the *fin* flag is set. In addition, functionality similar to `break`, `continue`, and `goto` statements are achieved using the Fin statement.

At the same time, one of the inconsistencies with the specification of the Plankalkül is encountered, which is the behavior of the Fin statement. This is addressed with the assumption that the examples given in Zuse (1989) are not entirely correct.

An alternative solution to providing the semantics of these control constructs is by using continuations. However, as it is intended to provide a simple semantics of the Plankalkül, the approach presented in this section is preferred.

2.4 *linPlank2* - A More Practical Subset of *linPlank*

2.4.1 Extensions to the Syntax

The language defined in *linPlank1* has no practical use, except for a few bit manipulations. This language will be extended in *linPlank2* with arrays, tuples, more loops and subprogram definitions and calls.

Arrays in the Plankalkül are denoted with the structure S$n \times \sigma$, where σ is another Plankalkül structure. A special case is bit arrays, which are denoted with the structure S1.n. Tuples, on the other hand, are denoted with the structure S$(\sigma_1, \sigma_2, \ldots, \sigma_n)$, where each σ_i is a Plankalkül structure. The individual components or array and tuple values are accessed using the K row of the Plankalkül syntax. In *linPlank*, we denote `V0`$[:k:] : s$ as the component k of the variable V0 having the structure s.

In the previous subset, the general loop was introduced. In *linPlank2*, additional loops are supported, and they are aptly names W0 to W4. The W0 loop executes the statement the number of times equal to the argument supplied with the loop. The W1, on the other hand, behaves like the W0 loop, but also opens up access to a loop variable which is initialised to 0 and gets incremented during every iteration. The W2 to W4 loops behave like W1, but have differences in the way the attached loop variable behaves. For instance, the W2 loop's variable decrements during every iteration starting from one less that the value of the argument supplied, while the W3 loop's variable starts with the value of the first argument, and increments every iteration until it is equal to the value of the second argument. The loop variable of W4 does the reverse. From the behavior of the loop variables, it can be concluded that loop variables always have numeric values.

linPlank2 also supports multiple subprograms. In a set of subprograms, the last one is the main program. Also, a subprogram cannot call itself; it can only call subprograms above it. This way, recursion

is eliminated.

The syntax of *linPlank2* will have the following extensions to *linPlank1*:

- $St \in$ Structures

  $K \in$ Components

  $N \in \mathbb{N}$

  $B \in$ Bitstrings $= L((\texttt{0}|\texttt{L})(\texttt{0}|\texttt{L})*)$

- $P \longrightarrow Sp\ P \mid Sp$

  $S \longrightarrow \ldots \mid \texttt{W0}(E)[S] \mid \texttt{W1}(E)[S] \mid \texttt{W2}(E)[S] \mid \texttt{W3}(E_1, E_2)[S] \mid \texttt{W4}(E_1, E_2)[S]$

  $E \longrightarrow \ldots \mid N \mid Var[:K:] : St \mid \texttt{i}\ idx \mid \texttt{i}\ idx[:K:] \mid \texttt{N}(E) \mid \texttt{R}\ fid(E_1, E_2, \ldots, E_n)idx$

  $\quad \mid (E_1, E_2, \ldots, E_n)$

  $St \longrightarrow \texttt{0} \mid St' \mid \varepsilon$

  $St' \longrightarrow \texttt{1}.NExp \mid NExp * St' \mid (St_1, St_2, \ldots, St_n)$

  $NExp \longrightarrow N \mid NVar \mid NExp_1 + NExp_2 \mid NExp_1 - NExp_2 \mid (NExp)$

  $NVar \longrightarrow \texttt{n} \mid \texttt{n}N$

  $K \longrightarrow E \mid \varepsilon$

2.4.2 Additions to the Semantic Framework

Tuple Values

There is a need for a data model to support values inside arrays and tuples. A few considerations need to be taken into account [1]:

- Are the structures S2×1.3 and S$(1.3, 1.3)$ one and the same? In general, is a structure S$n \times \sigma$ equivalent to the structure S$(\underbrace{\sigma, \sigma, \ldots, \sigma}_{n})$?
- How can arrays be assigned values without accessing their individual components?

To address both issues, tuple values are introduced in the semantic domain, in which they are regarded as denotable values. Tuple values can be classified into two groups: *homogeneous* and *heterogeneous* tuple values. Homogeneous tuple values are tuple value whose elements have the same structure,

[1] Although the notion of tuple values appear to address these issues, the type system eventually does not permit transparency between list types and tuple types. More details by the end of Section 3.5.

as in the case of arrays. Heterogeneous tuple values, on the other hand, are tuple values whose elements have different structures, as in the case of general tuples.

There are a few operations involving tuples. The operation *maketuple* takes a set of denotable values and turns them into a tuple. The operation *getcomponent*, on the other hand, gets the i-th component of a tuple value, for some number i less than the size of the tuple. Meanwhile the operations *converttotuple* and *converttonumber* makes conversions between numberic and tuple values. However, *converttonumber* is not defined for every tuple value: it is only defined for homogeneous tuples whose elements are bit values. Finally, the *getsize* operation returns the size of a tuple value.

- Domain: $TV = DV \times DV \times \ldots \times DV$

  Operations:

  - $maketuple : DV \times DV \times \ldots \times DV \to TV$
  - $getcomponent : TV \to \mathbb{N} \to DV$
  - $converttotuple : \mathbb{N} \to TV$
  - $converttonumber : TV \to \mathbb{N}$
  - $getsize : TV \to \mathbb{N}$

- Domain: $DV = BV + TV + \{\bot\}$

The Loop Variable Store

The nature of loop variables were discussed in the section above. As all variables, the values these hold require some space in memory, and thus would require its own store. Like the other stores, the loop variable store is denoted as the Greek letter ι, and have the same operations. The only difference is that the loop variable store is a mapping from numeric values to numeric values, as loop variables always have numeric values.

- Domain: $ISto = (\mathbb{N} \to (\mathbb{N} + \{\bot\})) + \{\bot\}$

  Operations:

  - $\varnothing : ISto$
  - $\mapsto$: $ISto \to \mathbb{N} \to N \to ISto$

The *level* Flag

To facilitate the access restriction to the loop variable store when executing any of the loops W1 to W4, the *level* flag is introduced. In the semantic framework, it is denoted by the Greek letter τ. Whenever a loop of any of the types W1 to W4 is executed, the *level* flag is incremented to allow access for the next loop variable in the store. Upon termination of execution of that loop, the *level* flag is decremented to disallow access to the loop variable just used by the loop.

Like the *fin* flag introduced in the previous subset, the *level* flag takes some space in the computer's memory. With this and the loop variable store, the definition of the store is modified as below:

- Domain: $Sto = VSto \times RSto \times ZSto \times ISto \times \mathbb{N} \times \mathbb{N}$

Function Stores

To facilitate subprogram calls, a store is needed to remember the information about each subprogram. This store is called the function store, and has the same operations as the previously defined stores. However, while the extension of the previous stores modifies them, extension of the function store is non-destructive. This means that a copy of the function store is made and that copy is then modified with the extension.

- Domain: $FSto = \mathbb{N} \rightarrow DV \times DV \times \ldots \times DV \rightarrow DV \times DV \times \ldots \times DV$
  Operations:
  - $\varnothing : FSto$
  - $\mapsto$: $FSto \rightarrow \mathbb{N} \rightarrow (DV \times DV \times \ldots \times DV \rightarrow DV \times DV \times \ldots \times DV) \rightarrow FSto$

  The function store is denoted by the Greek symbol δ in the semantic framework.

New Definitions for the Semantic Functions

Having laid out the necessary tools needed to extend the framework, the modifications to the semantic functions are as follows:

- $\mathcal{P}[\![\texttt{P}\ \textit{fid}\ \texttt{R(V0, V1, ..., Vn)=>(R0, R1, ..., Rm)}\ [S]\ P]\!]\delta(\alpha_1, \alpha_2, \ldots, \alpha_{n+1}) =$
  $\mathcal{P}[\![P]\!]\delta'(\alpha_1, \alpha_2, \ldots, \alpha_{n+1}),$
  where $\delta' = \delta[\textit{fid} \mapsto (\mathcal{S}[\![\texttt{P}\ \textit{fid}\ \texttt{R(V0, V1, ..., Vn)=>(R0, R1, ..., Rm)}\ [S]]\!]\delta)]$

- $\mathcal{P}[\![Sp]\!]\delta(\alpha_1, \alpha_2, \ldots, \alpha_{n+1}) = \mathcal{S}[\![Sp]\!]\delta(\alpha_1, \alpha_2, \ldots, \alpha_{n+1})$
- $\mathcal{X}[\![\texttt{W0}(E)[S]]\!]\delta\sigma = \mathcal{X}[\![\underbrace{[[S][S]\ldots[S]]}_{n}]\!]\delta\sigma \qquad$ where $n = \mathcal{E}[\![E]\!]\delta\sigma$
- $\mathcal{X}[\![\texttt{W1}(E)[S]]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau) = (\nu', \rho', \zeta', \iota', \phi', \tau),$

  where $(\nu', \rho', \zeta', \iota', \phi', \tau') = \mathcal{X}'[\![\texttt{W1}(E)[S]]\!]\delta(\nu, \rho, \zeta, \iota[\tau \mapsto 0], \phi, \tau + 1)$

  $$\mathcal{X}'[\![\texttt{W1}(E)[S]]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau) = \begin{cases} \mathcal{X}'[\![\texttt{W1}(E)[S]]\!]\delta(\nu', \rho', \zeta', \iota'[\tau - 1 \mapsto \iota'(\tau - 1) + 1], \phi', \tau), \\ \qquad\qquad \text{if } \iota(\tau - 1) < n \text{ and } \phi' = 0 \\ \mathcal{X}'[\![\texttt{W1}(E)[S]]\!]\delta(\nu', \rho', \zeta', \iota'[\tau - 1 \mapsto \iota'(\tau - 1) + 1], \phi' - 1, \tau), \\ \qquad\qquad \text{if } \iota(\tau - 1) < n \text{ and } \phi' = 1 \\ (\nu', \rho', \zeta', \iota', \phi' - 1, \tau), \\ \qquad\qquad \text{if } \iota(\tau - 1) < n \text{ and } \phi' > 0 \\ (\nu, \rho, \zeta, \iota, \phi, \tau), \\ \qquad\qquad \text{if } \iota(\tau - 1) \geq n \end{cases}$$

  where $(\nu', \rho', \zeta', \iota', \phi', \tau') = \mathcal{X}[\![S]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau)$

  and $n = converttonumber(\mathcal{E}[\![E]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau))$

  The loops W2 to W4 can be similarly defined.

- $\mathcal{E}[\![N]\!]\delta\sigma = \mathcal{B}[\![N]\!]$
- $\mathcal{E}[\![\texttt{V}\ idx[:E:]]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau) = getcomponent(\nu(idx), \mathcal{E}[\![E]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau))$
- $\mathcal{E}[\![\texttt{R}\ idx[:E:]]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau) = getcomponent(\rho(idx), \mathcal{E}[\![E]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau))$
- $\mathcal{E}[\![\texttt{Z}\ idx[:E:]]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau) = getcomponent(\zeta(idx), \mathcal{E}[\![E]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau))$
- $$\mathcal{E}[\![\texttt{i}\ idx]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau) = \begin{cases} converttotuple(\iota(idx)), & \text{if } idx < \tau \\ \bot, & \text{otherwise} \end{cases}$$
- $$\mathcal{E}[\![\texttt{i}\ idx[:E:]]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau) = \begin{cases} getcomponent(converttotuple(\iota(idx)), n), \\ \qquad\qquad \text{if } idx < \tau \\ \bot, \\ \qquad\qquad \text{otherwise} \end{cases}$$

  where n = $\mathcal{E}[\![E]\!]\delta(\nu, \rho, \zeta, \iota, \phi, \tau)$

- $\mathcal{E}[\![\texttt{N}(E)]\!]\delta\sigma = converttotuple(getsize(\mathcal{E}[\![E]\!]\delta\sigma))$
- $\mathcal{E}[\![\texttt{R}\, fid(E_1, E_2, \ldots, E_n)idx]\!]\delta\sigma = getcomponent(t, idx)$
  where $t = maketuple(\delta(fid)(\mathcal{E}[\![E_1]\!]\delta\sigma, \mathcal{E}[\![E_2]\!]\delta\sigma, \ldots, \mathcal{E}[\![E_n]\!]\delta\sigma))$
- $\mathcal{E}[\![(E_1, E_2, \ldots, E_n)]\!]\sigma\delta = maketuple(\mathcal{E}[\![E_1]\!]\delta\sigma, \mathcal{E}[\![E_2]\!]\delta\sigma, \ldots, \mathcal{E}[\![E_n]\!]\delta\sigma)$
- All the other definitions are modified to include δ, the function store, ι, the loop variable store, and τ, the *level* flag, when appropriate.

While the meaning of W0 loops might seem unclear, it just means that its meaning is equivalent to a block-bounded finite sequence of $[S]$. The meaning of W1 to W4 loops, however, is defined using two semantic functions. The first one takes care of the behavior of the *level* flag, while the other defines the execution of each iteration of the loop, as well as the behavior of the loop variable.

2.4.3 Assessment

One of the design decisions made when this semantics was formed is to have a bit-centric data model, which was intended by Zuse from the beginning (Zuse, 1989). This means that the only native operations in the semantic framework are boolean operations. Thus, the arithmetic operations between numbers have to be expressed in terms of series of boolean operations.

One problem with this design is illustrated by the following. Take for example the program for adding 1 to an argument:

```
P4 R(V0:1.n) => (R0:1.(n+1), R1:0) [
    L => Z0:0;
    W1(N(V0)) [
        V0[:i:]  => Z1:0;
        Z0 !~ Z1 => R0[:i:];
        Z0 & Z1 => Z0;
    ]
    Z0 => R0[:N(V0):];
    Z0 => R1;
]
```

Note that this program is defined using a W1 loop, where the loop variable i is incremented by 1 every iteration. There are two problems with this. One is that the Plankalkül does not support recursive calls. And even if the Plankalkül did support recursive calls, this program will not terminate. This clearly shows a design flaw, which uses addition in its own definition.

The solution presented by the provided semantics is to represent values for loop variables as numeric values and allow arithmetic and comparison operations on these values. Conversion between numeric values and tuple values are also provided to ensure compatibility with the bit-centric data model. Apparently, this is also the reason why the loops W0 to W4 cannot be rewritten as variations of the W loop.

2.5 Concluding Remarks

At this point there is a basic idea how the semantics of the Plankalkül is like. The semantics revolved around the procedural aspects of the language, which is quite similar to that of modern procedural languages such as C, Pascal, and BASIC.

However there are a few issues not taken into account. One of them is memory management, where assumptions regarding memory handling are made. A more serious issue is the assurance that values are compatible with the underlying operations, e.g. the boolean operations under the domain of bit values. An attempt to solve this is presented in the next chapter, where the type system is discussed.

Chapter 3

A Type System for *linPlank*

3.1 Motivation

After defining the denotational semantics for *linPlank*, which defines how programs in *linPlank* and, in a similar way, the Plankalkül behave, a type system for *linPlank* will be designed. Knowing the structural nature of the data model of the Plankalkül, it is only natural that an attempt to design a type system for it would be undertaken.

There are three major structures in the Plankalkül: the bit type, denoted as S0; the list type, denoted by S1.n for an array of bits and S$n \times \sigma$ for an array of data of type σ; and the tuple type, denoted by S$(\sigma_1, \sigma_2, \ldots, \sigma_n)$ for a tuple consisiting of data of types σ_1, σ_2, etc. These structures, according to Zuse (1989), are able to define most data.

While it is indeed possible to define most data using compositions of these basic structures, there are issues involved. One of which is that it is possible for variables to appear in these structures, which may imply that some form of polymorphism or type inference is involved. Another is that while the types of V and R variables are declared in the marginal data abstract of the subprogram, it is possible for Z variables to appear with no type declaration whatsoever. Again this may imply some form of polymorphism or type inference.

3.2 Some Assumptions

To address the issues mentioned above, particularly the second, some assumptions are made. For instance, it is assumed that the first instance of any Z variable shall have its type declared. It is also assumed that if only a component of a Z variable has its type declared, the type of the Z variable is an array with components having the same type as the type declaration.

3.3 Types in the Semantic Domain and the Type Translation Relation

The types in the semantic domain are defined as follows:

$$t ::= b \mid array(t) \mid (t_1, t_2, \ldots, t_n),$$

where the array type refers to an array of undefined size.

As one may notice, this definition is not the same as the syntactic type definition mentioned above. Thus, a translation relation, $\Rightarrow$, is introduced. $\sigma \Rightarrow \tau$ means that the syntactic type σ translates to the semantic type τ. In particular, the translation between syntactic types of *linPlank* and the semantic types defined above are as the following inductive rules:

$$\frac{}{\texttt{S0} \Rightarrow b} \qquad \frac{}{\texttt{S1.n} \Rightarrow array(b)}$$

$$\frac{\sigma \Rightarrow \tau}{\texttt{Sn}{*}\sigma \Rightarrow array(\tau)} \qquad \frac{\sigma_1 \Rightarrow \tau_1 \qquad \sigma_2 \Rightarrow \tau_2 \qquad \cdots \qquad \sigma_n \Rightarrow \tau_n}{\texttt{S}(\sigma_1, \sigma_2, \ldots, \sigma_n) \Rightarrow (\tau_1, \tau_2, \ldots, \tau_n)}$$

3.4 Type Environments

Type environments work like stores in the denotational semantics of *linPlank*. However, unlike stores, type environments are mappings from variable names to semantic types. In the type system of the *linPlank*, there are two type environments, one for subprograms and one for variables. They are denoted as Γ_s and Γ_v respectively.

The operations involving type environments are access and extension. Type environment access, denoted by $\Gamma(\mathit{Var})$, where Var is a variable name, returns the type of the argument, provided that a mapping between the argument and some type exists in the environment. Type environment extension,

on the other hand, includes a mapping for the given variable name to the given type, and is denoted by $\Gamma[\,Var \mapsto t]$. This operation is non-destructive, which means that a copy of the type environment is first made, and this new copy is then extended.

3.5 Well-typedness of Expressions

The main point of having a type system is to ensure that elements in a program written in some language is well-typed. Well-typedness ensures that the data behaves as it is supposed to be; when a conditional statement requires its conditional expression to have type S0, then any expression used in a conditional statement should have type S0 for it to be well-typed.

The relation for well-typedness of expressions is defined as

$$(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright t \mid \Gamma'_v$$

where E is an expression, t is the type of the expression and Γ'_v is the returned type environment for variables due to the test of the well-typedness of E.

One may wonder why there is a need to return a type environment along with the type of the expression when checking its well-typedness. This is because there are no separate type declaration constructs in the Plankalkül; variables are declared as they appear in an expression. Since the type environment is modified after the check and inductive rules are used to define the well-typedness of expressions, returning a type environment along with the type is the solution.

Below are the definitions for well-typedness of the various expressions in *linPlank*:

$$\frac{}{(\Gamma_s, \Gamma_v) \vdash \texttt{0} \blacktriangleright b \mid \Gamma_v} \qquad \frac{}{(\Gamma_s, \Gamma_v) \vdash \texttt{L} \blacktriangleright b \mid \Gamma_v} \qquad \frac{b \in \text{Bitstrings}}{(\Gamma_s, \Gamma_v) \vdash b \blacktriangleright \mathit{array}(b) \mid \Gamma_v}$$

$$\frac{n \in \mathbb{N}}{(\Gamma_s, \Gamma_v) \vdash n \blacktriangleright \mathit{array}(b) \mid \Gamma_v} \qquad \frac{\exists t\,(V\ \mathit{idx}, t) \in \Gamma_v \qquad V \in \{\texttt{V}, \texttt{R}, \texttt{Z}\}}{(\Gamma_s, \Gamma_v) \vdash V\ \mathit{idx} \blacktriangleright t \mid \Gamma_v}$$

$$\frac{}{(\Gamma_s, \Gamma_v) \vdash \texttt{i}\ \mathit{idx} \blacktriangleright \mathit{array}(b) \mid \Gamma_v} \qquad \frac{\forall t\,(V\ \mathit{idx}, t) \notin \Gamma_v \qquad V \in \{\texttt{V}, \texttt{R}, \texttt{Z}\} \qquad \sigma \Rightarrow \tau}{(\Gamma_s, \Gamma_v) \vdash V\ \mathit{idx} : \sigma \blacktriangleright \tau \mid \Gamma_v[V\ \mathit{idx} \mapsto \tau]}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E_1 \blacktriangleright b \mid \Gamma'_v \qquad (\Gamma_s, \Gamma'_v) \vdash E_2 \blacktriangleright b \mid \Gamma''_v}{(\Gamma_s, \Gamma_v) \vdash E_1\ \theta\ E_2 \blacktriangleright b \mid \Gamma''_v} \qquad \text{where } \theta \in \{\texttt{\&}, \texttt{|}, \texttt{->}, \texttt{\~{}}, \texttt{!\~{}}\}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright b \mid \Gamma'_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{!}E \blacktriangleright b \mid \Gamma'_v} \qquad \frac{(\Gamma_s, \Gamma_v) \vdash V\ idx \blacktriangleright array(\tau) \mid \Gamma'_v \qquad V \in \{\texttt{V}, \texttt{R}, \texttt{Z}, \texttt{i}\}}{(\Gamma_s, \Gamma_v) \vdash V\ idx[\texttt{:}E\texttt{:}] \blacktriangleright \tau \mid \Gamma'_v}$$

$$\frac{\forall t\ (V\ idx, t) \notin \Gamma_v \qquad V \in \{\texttt{V}, \texttt{R}, \texttt{Z}\} \qquad \sigma \Rightarrow \tau}{(\Gamma_s, \Gamma_v) \vdash V\ idx[\texttt{:}E\texttt{:}] \texttt{:}\sigma \blacktriangleright \tau \mid \Gamma_v[V\ idx \mapsto array(\tau)]} \qquad \text{(based on second assumption)}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash V\ idx \blacktriangleright (t_1, t_2, \ldots, t_n) \mid \Gamma'_v \qquad V \in \{\texttt{V}, \texttt{R}, \texttt{Z}\}}{(\Gamma_s, \Gamma_v) \vdash V\ idx[\texttt{:}n\texttt{:}] \blacktriangleright t_n \mid \Gamma'_v} \qquad \frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright array(t) \mid \Gamma'_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{N}(E) \blacktriangleright array(b) \mid \Gamma'_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E_1 \blacktriangleright t_1 \mid \Gamma_v^{[1]} \qquad (\Gamma_s, \Gamma_v^{[1]}) \vdash E_2 \blacktriangleright t_2 \mid \Gamma_v^{[2]} \qquad \cdots \qquad (\Gamma_s, \Gamma_v^{[n-1]}) \vdash E_n \blacktriangleright t_n \mid \Gamma_v^{[n]}}{(\Gamma_s, \Gamma_v) \vdash (E_1, E_2, \ldots, E_n) \blacktriangleright (t_1, t_2, \ldots t_n) \mid \Gamma_v^{[n]}}$$

$$\frac{\begin{array}{c}(\Gamma_s, \Gamma_v) \vdash E_1 \blacktriangleright t_1 \mid \Gamma_v^{[1]} \qquad (\Gamma_s, \Gamma_v^{[1]}) \vdash E_2 \blacktriangleright t_2 \mid \Gamma_v^{[2]} \qquad \cdots \qquad (\Gamma_s, \Gamma_v^{[n-1]}) \vdash E_n \blacktriangleright t_n \mid \Gamma_v^{[n]} \\ \Gamma_s(\mathit{fid}) = (t_1, t_2, \ldots, t_n) \rightarrow (t'_1, t'_2, t'_m)\end{array}}{(\Gamma_s, \Gamma_v) \vdash \texttt{R}\ idx(E_1, E_2, \ldots, E_n)\ \mathit{fid} \blacktriangleright t'_{idx} \mid \Gamma_v^{[n]}}$$

Note that since the array type has an undefined size, we cannot determine in the type-checking stage if two expressions have an array type with the same size. Array size checks are therefore checked in runtime.

At this point, the questions raised in Section 2.4.2 will be addressed. Take a look at the second to the last typing rule above, which is the rule for type-checking tuple expressions. If we allow the list structure S$n \times \sigma$ to be equivalent to the tuple structure S$(\underbrace{\sigma, \sigma, \ldots, \sigma}_{n})$, the following rule can be added:

$$\frac{(\Gamma_s, \Gamma_v) \vdash E_1 \blacktriangleright t \mid \Gamma_v^{[1]} \qquad (\Gamma_s, \Gamma_v^{[1]}) \vdash E_2 \blacktriangleright t \mid \Gamma_v^{[2]} \qquad \cdots \qquad (\Gamma_s, \Gamma_v^{[n-1]}) \vdash E_n \blacktriangleright t \mid \Gamma_v^{[n]}}{(\Gamma_s, \Gamma_v) \vdash (E_1, E_2, \ldots, E_n) \blacktriangleright array(t) \mid \Gamma_v^{[n]}}$$

However, as the definition of the array type does not care for sizes, i.e., a bit array with size 5 is type-equivalent to another bit array of size 3, the expression `(V0:1.5, V1:1.3)` would be typed

as $array(array(b))$, which is misleading. Thus, the claim that S$n \times \sigma$ is equivalent to the tuple structure S$(\underbrace{\sigma, \sigma, \ldots, \sigma}_{n})$ is false.

3.6 Well-typedness of Statements

Statements in *linPlank* have no type. However, like the check for well-typedness of expressions, the check for well-typedness of statements need to return the type environment for variables, since it is possible for new mappings to be included in that environment as expressions are included in some expressions.

The relation for well-typedness of statements is defined as

$$(\Gamma_s, \Gamma_v) \vdash S \mid \Gamma'_v$$

where S is a statement and Γ'_v is the returned type environment for variables due to the test of the well-typedness of S.

Below are the definitions for well-typedness of the various statements of *linPlank*:

$$\frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright t \mid \Gamma'_v \qquad (\Gamma_s, \Gamma'_v) \vdash V\ idx \blacktriangleright t \mid \Gamma''_v \qquad V \in \{\texttt{V}, \texttt{R}, \texttt{Z}\}}{(\Gamma_s, \Gamma_v) \vdash E\ \texttt{=>}\ V\ idx \mid \Gamma''_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright b \mid \Gamma'_v \qquad (\Gamma_s, \Gamma'_v) \vdash S \mid \Gamma''_v}{(\Gamma_s, \Gamma_v) \vdash E\ \texttt{->}\ [S] \mid \Gamma''_v} \qquad \frac{(\Gamma_s, \Gamma_v) \vdash S \mid \Gamma'_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{W}[S] \mid \Gamma'_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright array(b) \mid \Gamma'_v \qquad (\Gamma_s, \Gamma'_v) \vdash S \mid \Gamma''_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{W0}(E)[S] \mid \Gamma''_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright array(b) \mid \Gamma'_v \qquad (\Gamma_s, \Gamma'_v) \vdash S \mid \Gamma''_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{W1}(E)[S] \mid \Gamma''_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright array(b) \mid \Gamma'_v \qquad (\Gamma_s, \Gamma'_v) \vdash S \mid \Gamma''_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{W2}(E)[S] \mid \Gamma''_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E_1 \blacktriangleright \mathit{array}(b) \mid \Gamma'_v \qquad (\Gamma_s, \Gamma_v) \vdash E_2 \blacktriangleright \mathit{array}(b) \mid \Gamma''_v \qquad (\Gamma_s, \Gamma''_v) \vdash S \mid \Gamma'''_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{W3}(E_1, E_2)[S] \mid \Gamma'''_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash E_1 \blacktriangleright \mathit{array}(b) \mid \Gamma'_v \qquad (\Gamma_s, \Gamma_v) \vdash E_2 \blacktriangleright \mathit{array}(b) \mid \Gamma''_v \qquad (\Gamma_s, \Gamma''_v) \vdash S \mid \Gamma'''_v}{(\Gamma_s, \Gamma_v) \vdash \texttt{W4}(E_1, E_2)[S] \mid \Gamma'''_v}$$

$$\frac{}{(\Gamma_s, \Gamma_v) \vdash \texttt{Fin}\ n \mid \Gamma_v} \qquad \frac{(\Gamma_s, \Gamma_v) \vdash E \blacktriangleright b \mid \Gamma'_v}{(\Gamma_s, \Gamma_v) \vdash E \texttt{ => Fin}\ n \mid \Gamma'_v}$$

$$\frac{(\Gamma_s, \Gamma_v) \vdash S_1 \mid \Gamma'_v \qquad (\Gamma_s, \Gamma'_v) \vdash S_2 \mid \Gamma''_v}{(\Gamma_s, \Gamma_v) \vdash S_1; S_2 \mid \Gamma''_v} \qquad \frac{(\Gamma_s, \Gamma_v) \vdash S \mid \Gamma'_v}{(\Gamma_s, \Gamma_v) \vdash [S] \mid \Gamma'_v}$$

3.7 Well-typedness of Programs

Finally, the well-typedness of programs, as well as that of its subprogram components, shall be discussed. For the well-typedness of subprograms, the function type is introduced. The function type is a mapping from a tuple of basic types to another tuple of basic types, i.e.,

$$t_f ::= (t_1, t_2, \ldots, t_n) \rightarrow (t'_1, t'_2, \ldots, t'_m)$$

At the same time, the relation for well-typedness of subprograms is as follows:

$$\Gamma_s \vdash Sp : t_f$$

where Sp is a subprogram and t_f is a function type. Note that no environment is returned. The definition for well-typedness of subprograms is as follows:

$$\frac{\begin{array}{c}(\Gamma_s, \Gamma_v) \vdash S \mid \Gamma'_v \qquad \sigma_1 \Rightarrow \tau_1 \qquad \sigma_2 \Rightarrow \tau_2 \qquad \cdots \qquad \sigma_{n+1} \Rightarrow \tau_{n+1} \\ \sigma'_1 \Rightarrow \tau'_1 \qquad \sigma'_2 \Rightarrow \tau'_2 \qquad \cdots \qquad \sigma'_{m+1} \Rightarrow \tau'_{m+1}\end{array}}{\begin{array}{c}\Gamma_s \vdash \texttt{P}\ \mathit{fid}\ \texttt{R}(\texttt{V0} : \sigma_1, \texttt{V1} : \sigma_2, \ldots, \texttt{V}n : \sigma_{n+1}) \texttt{ => } (\texttt{R0} : \sigma'_1, \texttt{R1} : \sigma'_2, \ldots, \texttt{R}m : \sigma'_{m+1})[S] \\ \blacktriangleright (\tau_1, \tau_2, \ldots, \tau_{n+1}) \rightarrow (\tau'_1, \tau'_2, \ldots, \tau'_{m+1})\end{array}}$$

where $\Gamma_v = \varnothing[\texttt{V0} \mapsto \tau_1][\texttt{V1} \mapsto \tau_2] \ldots [\texttt{V}n \mapsto \tau_{n+1}][\texttt{R0} \mapsto \tau'_1][\texttt{R1} \mapsto \tau'_2] \ldots [\texttt{R}m \mapsto \tau'_{m+1}]$. This

explains the need for a separate type environment for variables and for subprograms, as the type environment for variables only gets filled upon entry into a particular subprogram.

For the well-typedness of programs, the following relation is considered:

$$\Gamma_s \vdash P$$

and is defined by the following definitions:

$$\frac{\Gamma_s \vdash \texttt{P}\ \mathit{fid}\ \texttt{R}(\texttt{V0} : \sigma_1, \texttt{V1} : \sigma_2, \ldots, \texttt{V}n : \sigma_{n+1})\ \texttt{=>}\ (\texttt{R0} : \sigma'_1, \texttt{R1} : \sigma'_2, \ldots, \texttt{R}m : \sigma'_{m+1})[S] \blacktriangleright t_f \qquad \Gamma_s[\mathit{fid} \mapsto t_f] \vdash P}{\Gamma_s \vdash \texttt{P}\ \mathit{fid}\ \texttt{R}(\texttt{V0} : \sigma_1, \texttt{V1} : \sigma_2, \ldots, \texttt{V}n : \sigma_{n+1})\ \texttt{=>}\ (\texttt{R0} : \sigma'_1, \texttt{R1} : \sigma'_2, \ldots, \texttt{R}m : \sigma'_{m+1})[S]\ P}$$

$$\frac{\Gamma_s \vdash \mathit{Sp} \blacktriangleright t_f}{\Gamma_s \vdash \mathit{Sp}}$$

Finally, the check for well-typedness of the whole program is achieved by:

$$\frac{\varnothing \vdash P}{\vdash P}$$

This just means that the type system starts with an empty type environment for subprograms to check for the well-typedness of the program P.

3.8 Concluding Remarks

This chapter described a simple type system for the Plankalkül. However, the bit-centric data model of the language suggests a more complicated type system that might involve polymorphism or type inference. The simple type system ignores the complications by not taking into account the sizes of list values, and lets the run-time semantics take care of it. This will be discussed in the next chapter, particularly Section 4.3.5, which discusses issues regarding the implementation of the language.

Chapter 4

An Implementation of *linPlank*

4.1 Overview

In this chapter, an attempt on an implementation of *linPlank* will be discussed. This implementation is based on the theories formulated in the last two chapters, and problems encountered with applying these theories will be addressed.

Note that the implementation of *linPlank* is only part of the full implementation of the Plankalkül. The other part, which is the translation of Plankalkül programs to their respective linear representations in *linPlank*, is not covered in this thesis. Furthermore, only a subset of the Plankalkül, which was defined in the previous chapters, is implemented.

4.2 Implementation Design Issues

4.2.1 Interpreter as the Nature of the Implementation

Interpreters are implementations that directly execute statements of the target language. This is the type of implementation that this project has opted for, mainly because the semantics formulated in the previous chapters can be directly used as the basis for the implementation.

The semantics of *linPlank* presented in Chapter 2 does not handle memory management, and by developing an interpreter, the memory management capabilities of the language used in the development can be exploited to realize this.

However, it is to be noted that it was the intention of Zuse to compile Plankalkül programs into the native machine language of the computers he designed (Zuse, 1989, 1980; Knuth & Trabb Pardo, 1980). He planned to design a special computer that translates Plankalkül code into some native machine language, which he called “Planfertigungsgerät”. This has the same function as the compilers we use today.

4.2.2 Java as Choice of Language

Since Denotational Semantics makes use of domains and operations between them, it is only natural to use an object-oriented programming language in the development of the interpreter. This is because the theory can be directly applied without worrying too much about software design. For example, stores can be represented as object classes, having methods to allow access to its contents as well as insertion of data.

Furthermore, as mentioned in the previous section, the language used in the development of the interpreted has to have good memory management capabilities, since the semantics, and thus the implementation, does not explicitly handle memory management. The Java programming language, which is endorsed by Sun Microsystems, is reputed to have good memory management capabilities, as it can perform garbage collection on the memory every now and then.

Based on these arguments, as well as the fact that it is one of the simplest among the object-oriented programming languages, the Java programming language is chosen as the development language of the interpreter.

4.2.3 Overall System Design

The interpreter uses a modular design, in which most of the components are partially independent of each other. Figure 4.1 shows the process flow for the implementation.

4.3 Modules

4.3.1 Preprocessor

The preprocessor in the interpreter adds the standard subprograms needed to execute some constructs, such as arithmetic and comparison operators. These subprograms are written in linPlank as well, to demonstrate the capabilities of the bit-centric data model of the Plankalkül. These are labeled as P1 to

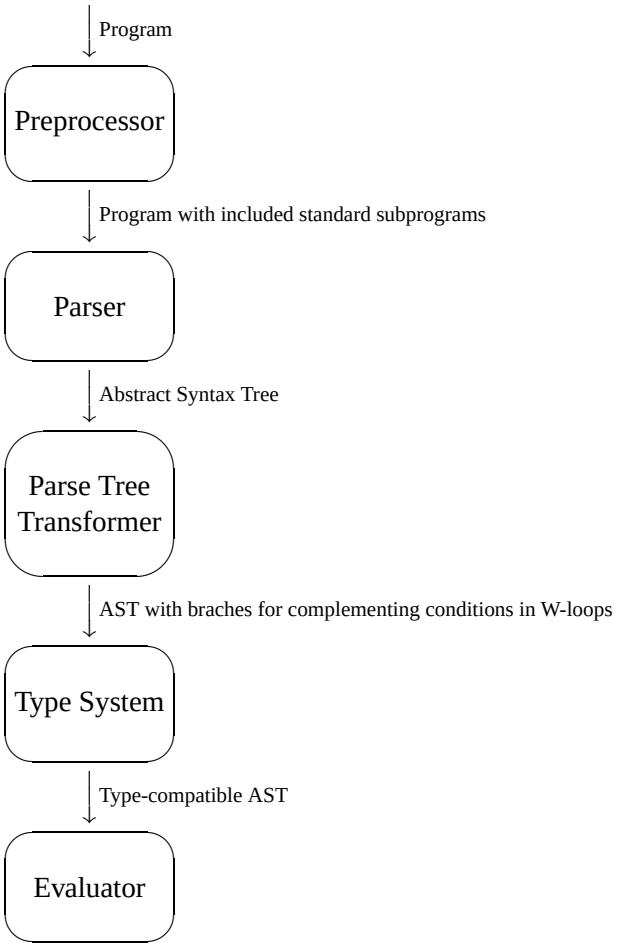

Figure 4.1: Overall Implementation Design

P11, and thus any other program must be labeled as P12 or higher. Other than this, there are no serious issues involved.

4.3.2 Parser

The parser is done using the Java CUP parser generator and the JLex lexical analyzer. The nodes of the abstract syntax tree generated, for each of the constructs of *linPlank*, are represented as classes. It is interesting to note that the notion of inheritance is exploited here. Statements and expressions are represented by interfaces, and each kind of statement or expression implements either one of the interfaces.

4.3.3 Parse Tree Transformer

In Section 2.3.1, it was mentioned that the body of W-loops is usually composed of a sequence of conditional statements. Furthermore, the complementing conditional statement, which is the negation of all the previous conditions, can be omitted and assumed present. This module attempts to find out the complementing conditional statement based on the collected conditions inside a loop.

The first step done by the Parse Tree Transformer is to find all the W-loops inside the tree. For each of these W-loops, the body is scanned for conditional statements and their conditions are gathered in a vector. The complementing condition is generated by getting the conjunction of the negations of each of the conditions in the vector. For example, for the following loop,

W [
    C_1 -.> S_1;
    C_2 -.> S_2;
]

the complementing conditional statement generated is !C_1 & !C_2 => Fin 2.

4.3.4 Type System

The type system module is directly based on the type system formulated in Chapter 3. Type environments are represented by objects which have the behavior of hashtables. These hashtables map variable names and indices to types, which we represented as classes separate from the abstract syntax tree node classes.

Again, the notion of inheritance is exploited here.

4.3.5 Run-time Evaluator

The evaluator is based on the semantic functions described in Chapter 2. At the same time, this module addresses the issue not handled by the type system defined in Chapter 3 – array size compatibility.

To address this issue, tuple values are allowed to be non-fixed, meaning that it is possible for non-fixed tuple values to be casted to have a fixed size. This is used to evaluate numeric constants, which we defined to be tuple values with variable size, and also used to create values for variables which we assumed to be of the list type, but do not know its size (refer to Section 3.2 for the explanation).

Casting is done on two occasions. One is during evaluation of the assignment statement, where the type and size of the variable at the right hand side of the statement is known. The other is during the execution of subprograms, where values for any list size variables are computed based on the list size expressions in the structures of the input variables and the nature of the passed arguments.

Size expressions are limited to addition and subtraction because allowing multiplication and division can complicate the computation of the values of list size variables [1]. So how are the values for list size variables computed? First the list size expression parse trees are simplified, such that all variables are in the left side of the tree, as in Figure 4.2. Then the right branch of the tree is cut off and the value of the node at those branches are subtracted from or added to, depending on the sign of the root node, the size of the given tuple value argument. This is done until the tree only contains variables, and if the parse tree only contains one variable, that variable is assigned the value of the manipulated size (See Figure 4.3). Finally, the set of list size expressions, being a system of linear diophantine equations, is converted to a matrix and then solved using an algorithm described in Blankinship (1966).

4.4 Concluding Remarks

This chapter roughly explained an implementation of *linPlank* and addressed certain issues involved. One of these issues is list size compatibility, in which list sizes should be the same upon assignment and upon passing to subprograms as arguments. A related issue would be computation of the values of list size variables in which a rough methodology was developed.

[1] In particular, it is possible to arrive at a situation similar to Hilbert's Tenth Problem, which is proven unsolvable by Yuri Matiyasevich in 1970. This problem has something to do with the solution to Diophantine equations.

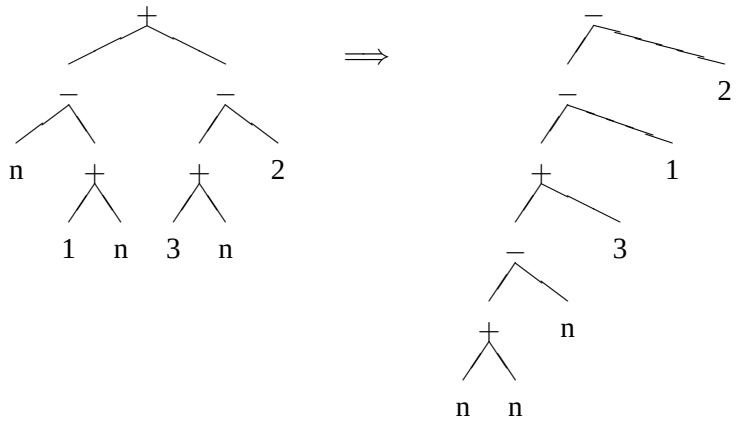

Figure 4.2: Transformation of List Size Expression Parse Trees

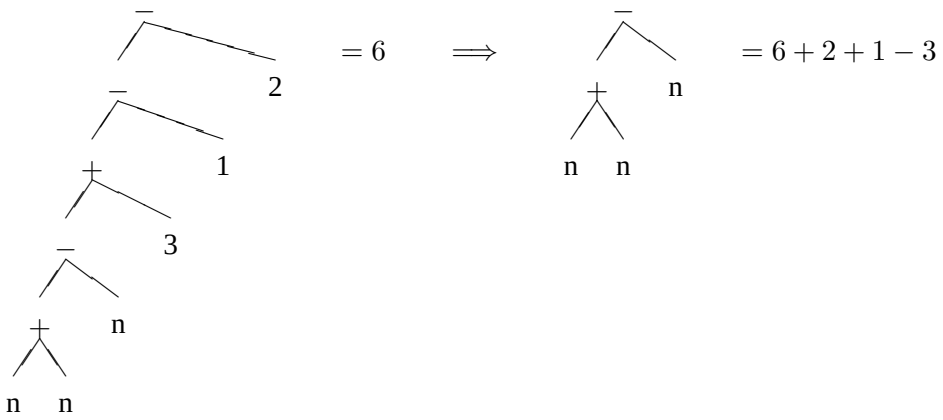

Figure 4.3: Computation of Values of List Size Variables

Chapter 5

Conclusion

5.1 The Effects of the Plankalkül in the History of Computing

The Plankalkül, being the first high-level programming language, can be said to be feature-rich. It has the capabilities of modern procedural programming languages such as C, Pascal and BASIC. Furthermore, the bit-centric data model of the language implies a sophisticated type system that might have involved polymorphism or type inference, though it was simplified in this thesis. This bit-centric data model can represent most data, as presented by Zuse in his “Statements of a Theory of General Calculation” (Zuse, 1989).

However, claims that the Plankalkül was a complete high-level programming language are questionable. This is because there are internal inconsistencies with respect to primitive arithmetic operations (see Section 2.4.3). At the same time, the implicit type system can only be made to work with serious restrictions (see Section 3.2). Otherwise, a more complicated type system has to be used.

Based on the findings gathered in this project, the Plankalkül might have some influence over succeeding procedural languages, in particular, Algol. However, this influence, if there was any, was limited for several reasons. One of these is that the language was much too ambitious, in the sense that it had a complicated type system, and at the same time it employed a two-dimensional syntax, at the time when the study of parsers and the whole area of compiler design had not yet been born. Also, there was no proper design, even for a subset of the language, ready for implementation. The formal methods of defining the semantics of programming languages only came much later. In particular, the denotational

semantics approach was only devised in the late 1970s (Stoy, 1977). Finally, there were errors in the documentation, such as program "bugs", and inconsistencies with the definitions of terms (see Section 2.3.1).

From these, we can conclude that Zuse was bound to fail in his Plankalkül, but he failed in an extremely interesting way. This is because he had the courage to take a user-centric approach in the design of the language, at a time when the formal study of programming languages did not exist. Nevertheless, the visionary nature of the Plankalkül deserves recognition, not from a practical, but from a historical point of view.

There is still one question to ask: what if the Plankalkül had actually been published during its period and people knew about it? In my opinion, people might have been able to embrace its procedural aspects and might have applied it to existing computers. However, the advanced features of the language, such as constructs for predicate calculus, might have been too far ahead of their time, and might not have been applicable so soon.

5.2 Limitations of this Project and Suggestions for Further Study

The implementation of the Plankalkül presented in this thesis is far from complete. Not every feature of the language was covered, such as the predicate calculus constructs. A better picture of the language can be visualized if a formal description of the semantics these features were formulated. Furthermore, memory management could be described in the formal semantics as well, to give a better picture of how an implementation would behave.

Also, it would be better if the Plankalkül was implemented using a compiler that generates machine code for one of Zuse's computers. A virtual machine that uses this machine code can then be developed. This can give a good simulation of what history would have been if the Plankalkül was implemented in one of Zuse's computers.

Finally, a study about how programs written in the Plankalkül's raw syntax can be translated into the linear representation *linPlank* can be undertaken. As the Plankalkül suggests a visual syntax, a visual editor could be of good use.

References

Bauer, F. L. (1980). Between Zuse and Rutishauser – the early development of digital computing in Central Europe. In *A history of computing in the twentieth century* (pp. 505–524). New York: Academic Press, Inc.

Bauer, F. L., & Wössner, H. (1972). The "Plankalkül" of Konrad Zuse: A forerunner of today's programming languages. *Communications of the ACM, 15*(7).

Blankinship, W. A. (1966). Algorithm 288: Solution of simultaneous linear diophantine equations. *Communications of the ACM, 9*(7).

Giloi, W. K. (1997). Konrad Zuse's Plankalkül: The first high-level, "non von Neumann" programming language. *IEEE Annals of the History of Computing, 19*(2).

Knuth, D. E., & Trabb Pardo, L. (1980). The early development of programming languages. In *A history of computing in the twentieth century* (pp. 197–273). New York: Academic Press, Inc.

Rojas, R., et al. (2000). *Plankalkül: The first high-level programming language and its implementation* (Tech. Rep. No. B-3/2000). Berlin: Freie Universität Berlin Institut für Informatik.

Schmidt, D. A. (1988). *Denotational semantics: A methodology for language development*. Dubuque, Iowa: Wm. C. Brown Publishers.

Stoy, J. E. (1977). *Denotational semantics: The Scott-Strachey approach to programming language semantics*. Cambridge, Massachusetts: MIT Press.

Zuse, H. (n.d.). *The life and work of Konrad Zuse.* Retrieved from `http://www.epemag.com/zuse`.

Zuse, K. (1980). Some remarks on the history of computing in Germany. In *A history of computing in the twentieth century* (pp. 611–627). New York: Academic Press, Inc.

Zuse, K. (1989). *The Plankalkül: Beritche der Gesellschaft für Mathematik und Datenverarbeitung Nr. 175*. München; Wien: Oldenbourg.