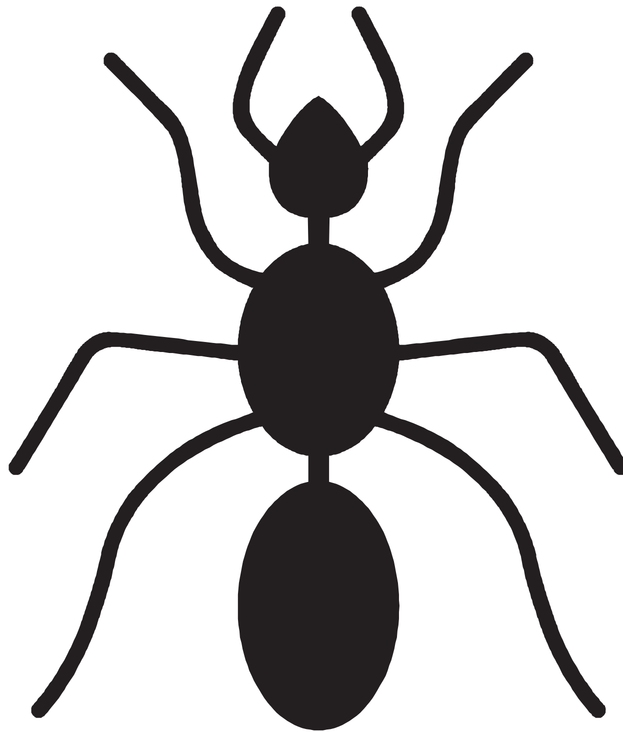
AWL
Ant War Language



Group E2-208, DAT2
Aalborg University

May 30th, 2003

Ant War Language

Written by

Tim Boesen,
Dennis Kjærulff Pedersen,
Thomas Pryds Lauritsen,
Jakob Rutkowski Olesen,
Carl Christian Sloth Andersen

On the semester

DAT2 for the course

SS, Syntax & Semantics

During the period of

February 3rd to May 30th, 2003

Synopsis

In this report we develop the programming language AWL (Ant War Language).

We define the syntax for the language and use it as a basis for the definition of the operational semantics. Next we define the abstract machine AWLAM and the operational semantics for its instructions.

The report then proceeds to prove the correctness of AWL by using induction in the length of transitions and induction on the shape of the derivation tree.

Preface

This report is written by project group E2-208 on DAT 2 at Department of Computer Science at Aalborg University during the spring semester of 2003. The report documents the development of a programming language made specifically for the game Ant War.

Thanks to Laurynas Speicys for supervising the project, and to Hans Hüttel for helping us.

Tim Boesen

Dennis Pedersen

Christian Andersen

Jakob Olesen

Thomas Lauritsen

Contents

1 Introduction **11**

1.1 Problem Area 11

1.2 Solution to the Problem 11

1.3 Outline of the project 12

1.4 Notation 13

1.4.1 Total and partial function space 13

1.4.2 Values 13

1.4.3 States 13

1.5 Summary 13

2 Syntax **14**

2.1 World 14

2.2 Memory 14

2.3 Rules 15

2.4 Commands 15

2.5 Parameters 16

2.6 Ant Type Declarations 16

2.7 Team Declarations 17

2.8 Variable Declarations 17

2.9 Expressions 17

2.10 Various 18

2.11 Summary 18

3 Operational Semantics for AWL **21**

3.1 Big Step Semantics 21

3.2 Abstract Syntax 22

3.2.1 Syntactic Categories 22

3.2.2 Constructs for the Syntactical Categories 23

3.3 Environment 23

3.3.1 Storage Structure 24

3.3.2 Variable Environment . . . 26
3.3.3 Procedure Environment . . . 26
3.3.4 Literal Functions . . . 27
3.3.5 Other Functions . . . 27
3.4 Transition Systems . . . 28
3.4.1 Arithmetic Expressions . . . 28
3.4.2 Boolean Expressions . . . 31
3.4.3 Direction Expressions . . . 33
3.4.4 Variable Declarations . . . 35
3.4.5 Array Declarations . . . 36
3.4.6 Rule Declarations . . . 37
3.4.7 Turn and Ant Type Declarations . . . 38
3.4.8 Common Memory Declarations . . . 38
3.4.9 Team Memory Declarations . . . 39
3.4.10 Private Memory Declarations . . . 40
3.4.11 Commands . . . 40
3.4.12 Formal and Actual Parameters . . . 44
3.4.13 Team Declaration . . . 44
3.4.14 World . . . 46
3.5 Standard Environment . . . 47
3.6 Derivation Tree . . . 47
3.7 Summary . . . 48

4 AWLAM **49**
4.1 Definition of AWLAM . . . 49
4.1.1 Notation and Definitions . . . 50
4.1.2 Instruction Set of AWLAM . . . 52
4.2 Operational Semantics of AWLAM . . . 53
4.3 Program Example . . . 57
4.4 Summary . . . 58

5 Code Generation **60**
5.1 Protocols . . . 60
5.2 Functions . . . 61
5.3 Code Generation . . . 62
5.3.1 Arithmetic Expressions . . . 62
5.3.2 Boolean Expressions . . . 64
5.3.3 Direction Expressions . . . 65

5.3.4 Variable Declarations . . . 66
5.3.5 Array Declarations . . . 66
5.3.6 Ant Type Declarations . . . 66
5.3.7 Rule Declarations . . . 67
5.3.8 Turn Declarations . . . 67
5.3.9 Team Declarations . . . 68
5.3.10 Common Memory Declarations . . . 68
5.3.11 Teambrain and Private Memory Declarations . . . 69
5.3.12 Formal and Actual Parameters . . . 70
5.3.13 World . . . 70
5.3.14 Commands . . . 71
5.4 Summary . . . 72

6 Provable Correct Implementation **73**

6.1 Correctness . . . 73
6.2 Proof Techniques . . . 73
6.3 Meaning of Commands . . . 74
6.4 Notation . . . 75
6.5 Variable Declarations . . . 75
6.6 Array Declarations . . . 76
6.7 Arithmetic Expressions . . . 77
6.8 Commands . . . 79
6.9 Summary . . . 81

7 Implementation **84**

7.1 Scanning and Parsing . . . 84
7.2 Identification and Type Checking . . . 84
7.3 Code Generation . . . 86
7.4 AWLAM Implementation . . . 86
7.4.1 The Evaluation Stack . . . 86
7.4.2 Registers . . . 86
7.4.3 Memory . . . 87
7.4.4 Interpretation . . . 87
7.5 Screenshots . . . 88
7.6 Summary . . . 89

8 Conclusion **90**

A Provable Correct implementation **94**

A.1 Variable Declarations 94

A.1.1 Variables 94

A.1.2 Arrays 95

A.1.3 Common Memory 96

A.1.4 Teambrain Memory 97

A.1.5 Private Memory 98

A.2 Parameters 99

A.2.1 Formal 99

A.2.2 Actual 100

A.3 Expressions 101

A.3.1 Arithmetic 101

A.3.2 Boolean 106

A.3.3 Direction 109

A.4 Commands 109

B SableCC Generated File **121**

Chapter 1

Introduction

In this chapter we will start out by giving a full description of the problem area. We will on basis of this description, write the problem statement, in which we outline the problems in key sentences. Following this, we will describe how we will solve the problems, and give an example off what the product of this rapport could be used for. We will summarize this with a few key sentences, which we consider to be the goal of the project. We will finish this chapter off by describing the layout of the report, and describing some of the notation used throughout the report.

1.1 Problem Area

There is a Danish programming game called Myrekrig (Ant war)[1]. The concept is that people program an ant algorithm (From now on just called an ant) in a programming language (originally C), adhering to a set of predefined rules. To determine the best algorithm, two or more ants will be run in a simulation engine, to determine which algorithm is the best.

As such C is just fine for the task, but there are several issues which justify designing a special purpose language for the simulations. First of all, C has a lot of functionality, which is not needed in designing an ant. This can be quite overwhelming for the unexperienced programmer. Furthermore it would be nice to have a higher level of abstraction, than C provides. For example, it would be more intuitive to read and write commands like $walk(LEFT)$ for moving an ant to the left, or $if(examine(RIGHT)) = FOOD\ then\ MoveAnt(RIGHT)$ for moving an ant to the right if it sees food. Designing a new language enables the designer to include only the important language constructions, and it is also possible to design a syntax which supports the underlying concept of the game.

The problem in key sentences is:

- At present time there is no specific language for creating a game of antwar
- Existing programming languages are often very complex, and does not provide constructs specifically for what will be required when creating a game of ant war.

1.2 Solution to the Problem

The solution to the problem is to create a special purpose programming language specifically for Ant War. We will call this language AWL (Ant War Language).

Here we will give a description of how a game of ant wars could look like using AWL, we will highlight certain terms, that will be used throughout the report, the first time they appear.

[1] The home-site for myrekrig is: `http://www.myrekrig.dk/`

We define a **world** consisting of a 2D-board of $n \times n$ fields/squares, in which **teams** can exist. Each team will consist of a number of **ants**. The team owners will define algorithms for their teams of ants and once the game is started, time will show who has the best algorithm. On their own hands, the ants will venture out into the world, gather **food**, and bring it back to their respective **bases** and in exchange get another ant. The size of the world, the maximum number of ants and the number of food pieces that exist in a single execution of the program is defined by the programmer. In the world the programmer can define some **common memory** which all teams can access and modify if they so choose. Furthermore each team will have some **team memory** which all ants on the team can access but which ants of other teams cannot access. Finally each ant has a **pr/home/tim/uni/awlcvs/awl/docs/awl_main.psivate memory** which other ants cannot access. It is now up to the programmer to define the world as he sees fit, this could include making sure that if an ant wanders out over the side of the defined board it will "magically" re-appear on the other side of the board. The programmer also defines how the game should end, if indeed he wants it to end and not run "forever".

Below we will, in key sentences, describe what the programming language must provide:

- AWL has to contain high level language constructs, such as in C.
- AWL has to provide specific constructs for the programmer, so that rules such as $walk(LEFT)$are easy to create.
- AWL has to provide a construct to allow the programmer to move the focus from one ant and team to another ant and team
- AWL has to provide some "ant memory" which differ in scope.
- AWL has to provide a construct for creating teams.
- AWL has to compile to an abstract machine, which we will call AWLAM(AWL Abstract Machine).

The goal of this project will then be to:

- Define the grammar of the high level language AWL, using Backus Naur Form(BNF)
- Define an operational big step semantic for AWL[2]
- Define the abstract machine AWLAM and have the AWL compile to this.
- Prove that the translated code is actually equivalent with the original AWL code.

1.3 Outline of the project

The project is divided into eight chapters, each covering their own topic of the project. This chapter is an introduction to the project. Chapter 2 is dealing with the syntax of our programming language, called AWL. We will describe the grammar of AWL, by the BNF notation, and will go into detail with what each single syntactic element does. Further on we will give an explanation of why the syntax is designed the way it is.

Chapter 3 deals with the operational semantic of AWL. We will describe the semantic details of AWL, and of course describe it in detail. Chapter 4 gives the definition of the abstract machine AWLAM, and give an operational semantic for this machine as well. The abstract machine's instruction set will be our target language, and in chapter 5 we will give the details on how the translation is actually done. To prove that the translation of AWL to AWLAM machine code is correct we will, in chapter 6, give proof of correctness for the translation. Finally we will describe the actual implementation of the AWLAM interpreter in chapter 7. In each of the chapters 1 through 7 there will be a summary which will serve as a sub conclusion for the chapter. Chapter 8 will contain the overall conclusion for the project.

[2]Operational big step semantics are also know as "natural semantics".

Besides that, the project contains an appendix with some of the proofs from chapter 6, and an appendix showing some of the details on the AWL to AWLAM compiler. There will also be a list on the literature used in the project.

1.4 Notation

In this report we will assume that the reader is familiar with basic set theory, though, we do use notations not normally considered part of basic set theory. Generally we use the notation of [1] as this has been the literature of our studies. In this section, the notation used throughout this report is briefly explained and examples are shown. However, at some points we might use a notation not discussed here. This is primarily the case when a notation is only used in a small part of the report. In these cases we will shortly address the notation in the beginning of the chapter or section in which it is used.

1.4.1 Total and partial function space

Given the sets A and B, the notation $A \to B$ describes the set of functions from A to B. Any element in that set is therefore a function that takes something from the set A and returns something from the set B. Such a function f is written as $f : A \to B$ (or as $f \in A \to B$). Therefore, $\mathit{Env} = \mathit{Var} \to \mathit{Loc}$ describes that the set Env consists of elements, where each element is a function from Var to Loc.Referring to environments and locations as in the example above, it would, however, be more correct written as $\mathit{Env} = \mathit{Var} \hookrightarrow \mathit{Loc}$. $\mathit{Var} \hookrightarrow \mathit{Loc}$ indicates that there is not necessarily an element of Loc defined for every element of Var. We call the elements of $\mathit{Var} \hookrightarrow \mathit{Loc}$ partial functions[3].

1.4.2 Values

Throughout the report we will be using the following notation regarding values of certain types.

$z \in$	$\mathbb{Z}$	(Numbers)
$b \in$	**Bool**	(Booleans)
$d \in$	**Dir**	(Directions)
$v \in$	$\mathbb{Z} \cup$ **Bool** $\cup$ **Dir**	(Union of numbers, booleans and directions)

1.4.3 States

A state s is a (partial) function, described as in section 1.4.1. Also it might be represented as $[a \mapsto 4, b \mapsto 3, c \mapsto 2]$ which means the state where the variable a maps to 4, b maps to 3, and c maps to 2 (or $a = 4$, $b = 3$, and $c = 2$). Often we use this notation for showing changes in a state. Consider the state s above. If we want to refer to a new state s' that is equal to s, except that $b = 5$, we will describe s' as $s[b \mapsto 5]$ (s except that b maps to 5).

1.5 Summary

In this chapter we have described the problem that this project aims to solve. Furthermore we have listed some demands that AWL must adhere to, and some general demands for the project. Following this we have described the outline of the report and described some notations used trough-out the report.

[3]Another notation for this is $\mathit{Var} \rightharpoonup \mathit{Loc}$, but we will not use it in this report.

Chapter 2

Syntax

In this chapter we define the grammar for AWL expressed in BNF notation (Backus Naur Form). The grammar should make it easy for the programmer to understand the concepts of AWL, and it should clarify distinctions between constructs which are semantically different.

The grammar has been divided into intuitive sub-categories to ease the reading and understanding, each with its own heading. It should however, be seen as a whole grammar. On the right side of each rule there is a combination of letter(s) and number(s) in parentheses. This is only meant as an easy reference to the individual rules, and are not part of the grammar itself.

2.1 World

These rules describe the general program construction. (P1) shows that an AWL program consists of a world. (P2) tells us that the world is constructed by three integer literal parameters and by defining the memory, which will be accessible for the ants. Also a world contains declarations of rules and ant types and finally a main section. The three integer literals are used in the initialization of the world and tells us the size of the world, the maximum number of ants there can be on each team and the maximum amount of food in the world. (P3) describes the construction of main, which consists of declarations and commands.

Program	::=	*World*	(P1)
World	::=	**world**(*IntLiteral*, *Intliteral*, *IntLiteral*)	(P2)
		{*Memory NTBRuleDecls TBRuleDecls AntTypeDecls Main*}	
Main	::=	**main**{*TeamDecls VarInits ArrayInits Commands*}	(P3)

The basic idea is that a program should be divided into some logical parts. First of all we want to separate the ants from the world, so that they can not access everything. This has been enforced through the memory and rule concepts. The only data available to the ant (besides itself) is the ant memory, and the only methods available to the ant are the ones declared as rules.

2.2 Memory

There are three different scopes when declaring memory for the ants. A piece of memory can be either private for each ant, private for the team or common to all ants on all teams. They have to be declared in a specific order as seen in (M1). Common memory is initialized when declared, so that the programmer can give ants access to some common attributes like worldsize. Team and private memory are only

declared, and it is up to the ants what they put in those variables. The memory defined is also persistent, this means that it will be saved from turn to turn[1].

Memory	::=	*CommonDecls Teambraindecls PrivateDecls*	(M1)
CommonDecls	::=	**common** *VarInit CommonDecls* $\mid \epsilon$	(M2)
TeambrainDecls	::=	**teambrain** *VarDecl TeambrainDecls* $\mid \epsilon$	(M3)
PrivateDecls	::=	**private** *VarDecl PrivateDecls* $\mid \epsilon$	(M4)

This construction has been included to be able to control which data is available to the ant and which is not. It should be possible for the world constructor to define how much memory each team should have access to. There will be a basis for difference in the behavior of the ants if they have access to an unlimited number of integers as opposed to only one integer.

2.3 Rules

The category Rules describes the syntax for adding functionality which the ant types can use. A rule can be one of two different types: NTBRule (R1) or TBRule (R4). TB means "turn-based", and NTB means "non turn-based". The difference between the two types is that a TB rule can only be called once every turn for each ant, while the NTB can be called several times (e.g. an ant can only walk once every turn, but may do several calculations on some number). Also the NTB rule may have a return type, which is a SimpleType. Both rule types can take parameters, and have their code embraced in curly brackets.

NTBRuleDecls	::=	*NTBRuleDecl NTBRuleDecls* $\mid \epsilon$	(R1)
NTBRuleDecl	::=	**rule** *Identifier*(*FormalParmList*) *ReturnType* { *Commands* }	(R2)
ReturnType	::=	: *SimpleType*	(R3)
TBRuleDecls	::=	*TBRuleDecl TBRuleDecls* $\mid \epsilon$	(R4)
TBRuleDecl	::=	**turn** *Identifier*(*FormalParmList*) { *Commands* }	(R5)

The reason for the Rule part is very similar to the memory. The world constructor must have some way to control that the ants do not access functions, which they should not have any knowledge about. Again it is up to the world constructor to define what should be accessible. It has been necessary to divide the rules into two categories. Some functions could be called several times, and some should only be called once. As an example it makes sense that it is only possible to call the walk command once for each turn. But of course it is possible for the world implementor to specify otherwise.

2.4 Commands

This part of the grammar describes the basic commands in AWL. (C1) shows that commands are either a single Command, or a Command followed by several other Commands. The first command is the assign command (C2), which is used to assign some value to an existing variable. The *mem* commands (C3) are used for assigning values to the memory variables of the ants. There is one command for each memory type, which is common, team and private. These commands are used in the ant types to access ant memory variables. The (C4) command is for assigning a value to an array. The location in the array is specified by the expression inside the square brackets. (C5) is a call to a declared rule with a parameter list. Rules (C6) and (C7) are general constructs for selection and iteration: the if-else selection, and the while loop.

Calling a TBRule and calling a NTBRule is semantically different (since a TBRule will end a turn, and a NTBRule will not). Because of this, it should not be possible to confuse a call to a TBRule call with a

[1] This is a difference from memory declared by the team programmer inside the ant specification in that the latter is not persistent.

call to a NTBRule (or the other way around) in the syntax of AWL. Therefore a call to a TBRule must have the prefix **endturn**. The return command is used to return a value from a rule. Skip does nothing but to skip, which means: nothing.

The command *process* will process one specific ant from one specific team using a given ant type. A call to *process* will normally return when a TBRule is called with endturn. *setProperty* takes two expressions, storage location and assign value). The command manipulates directly with memory, which means that the programmer potentially has access to everything.

Commands	::=	*Command Commands* $\mid \epsilon$	(C1)
Command	::=	*Identifier* = *Expr* ;	(C2)
	|	**cmem** *Identifier* = *Expr* ; | **tmem** *Identifier* = *Expr* ;	(C3)
	|	**pmem** *Identifier* = *Expr* ;	
	|	*Identifier* [*Expr*] = *Expr* ;	(C4)
	|	*Identifier* (*ActParmList*) ;	(C5)
	|	**if**(*Expr*){*Commands*}**else**{*Commands*}	(C6)
	|	**while**(*Expr*){*Commands*}	(C7)
	|	**endturn** *Identifier* (*ActParmList*) ;	(C8)
	|	**return** *Expr* ;	(C9)
	|	**skip** ;	(C10)
	|	**process**(*Expr*,*Expr*,*Identifier*) ;	(C11)
	|	**setProperty**(*Expr*, *Expr*) ;	(C12)

Most of these constructs are quite similar to those of other existing languages, while some of the constructs are unique to AWL. Section xxx in chapter xxx describes how the *setProperty* command can be useful when implementing a standard environment.

2.5 Parameters

This category describes the way parameters should be declared and used in rules. One part is the declaration of the formal parameters (PF2), which specifies the type of parameter required in a function, and the other is the actual parameters (PA4), which are the parameters actually used as input to a rule, when it is executed. All actual parameters are expressions, and expressions are explained a few categories below.

FormalParmList	::=	*FormalParm FormalParmList* $\mid \epsilon$	(PF1)
FormalParm	::=	*VarDecl* ;	(PF2)
ActParmList	::=	*ActParm ActParmList* $\mid \epsilon$	(PA3)
ActParm	::=	*Expr* ;	(PA4)

The reason we need these construct is to enable that rules can have arguments as input.

2.6 Ant Type Declarations

The AntType category describes the syntax for creating an AntType, i.e. a “race”. The declaration of an ant can be a single declaration or may consist of several declarations one after another. Each ant declaration must consist of the keyword **anttype** followed by an identifier, which is the name of the type. Inside curly brackets the commands of the declared anttype will be used.

AntTypeDecls	::=	*AntTypeDecl AntTypeDecls* $\mid \epsilon$	(AT1)
AntTypeDecl	::=	**anttype** *Identifier*{*Commands*}	(AT2)

With the separation of the world and the anttypes it is possible to change ants without actually changing much in the world. Each anttype is defined as a sort of procedure. Though it is quite different from the rules, and it makes it possible to easily see the difference of an ant and a rule, and thereby it helps us to separate the ants from the rules.

2.7 Team Declarations

A team declaration (T2) defines how a team is initialized. The createTeam keyword is followed by an identifier signifying the anttype, inclosed in brackets. A team declaration can be single declaration or several declarations (T1).

$$\begin{array}{lcll} \mathit{TeamDecls} & ::= & \mathit{TeamDecl}\ \mathit{TeamDecls} \mid \epsilon & \text{(T1)} \\ \mathit{TeamDecl} & ::= & \mathbf{createTeam}(\mathit{Identifier})\ ; & \text{(T2)} \end{array}$$

Each team is as said, of a certain anttype, and each team also have access to some memory, which is common memory and team memory.

2.8 Variable Declarations

The category variable declarations covers variable initialization, variable declarations, array initializations and array declarations. Variable initializations (V1) can be a single initialization or several initializations. An initialization (V2) consists of a declaration and an assignment. Each variable declaration (V3) must me prefixed with the keyword **var** and have a name and a simpletype (V7). The Array initialization and declaration is done much the same way, except that the keyword *array* is used.

$$\begin{array}{lcll} \mathit{VarInits} & ::= & \mathit{VarInit}\ \mathit{VarInits} \mid \epsilon & \text{(V1)} \\ \mathit{VarInit} & ::= & \mathit{VarDecl} = \mathit{Expr}\ ; & \text{(V2)} \\ \mathit{VarDecl} & ::= & \mathbf{var}\ \mathit{Identifier} : \mathit{SimpleType} & \text{(V3)} \\ \mathit{ArrayInits} & ::= & \mathit{ArrayInit}\ \mathit{ArrayInits} \mid \epsilon & \text{(V4)} \\ \mathit{ArrayInit} & ::= & \mathit{ArrayDec} = \mathit{Expr}\ ; & \text{(V5)} \\ \mathit{ArrayDec} & ::= & \mathbf{array}\ \mathit{Identifier} : \mathit{SimpleType} & \text{(V6)} \\ \mathit{SimpleType} & ::= & \mathbf{integer} \mid \mathbf{boolean} \mid \mathbf{direction} & \text{(V7)} \end{array}$$

These are standard declarations which is found in many programming languages such as C and Java.

2.9 Expressions

These rules show how expressions are made. A primary expression can be an expression in parentheses, a literal, a reference to an array element, a rule call or a reference to a ant memory variable (E1). It can also be one of the constructs *getProperty* or *random*. An expression can be an arithmetic, a relational or a boolean expression. Since the operators have different precedence, they have been organized so that arithmetic expressions will be evaluated first, then relational expressions and finally boolean expression.

PrimaryExpr	::=	(*Expr*) | *Literal*	(E1)
	|	*Identifier* [*Expr*] | *Identifier* (*ActParmList*)	
	|	**cmem** *Identifier* | **tmem** *Identifier* | **pmem** *Identifier*	
	|	**getProperty**(*Expr*)	
	|	**random**(*Expr*)	
Expr	::=	*OrExpr*	(E2)
OrExpr	::=	*OrExpr* **or** *AndExpr* | *AndExpr*	(E3)
AndExpr	::=	*AndExpr* **and** *EqualExpr* | *EqualExpr*	(E4)
EqualExpr	::=	*EqualExpr EqualOperator RelationalExpr* | *RelationalExpr*	(E5)
EqualOperator	::=	= | !=	(E6)
RelationalExpr	::=	*RelationalExpr RelationalOperator AddExpr* | *AddExpr*	(E7)
RelationalOperator	::=	< | > | <= | >=	(E8)
AddExpr	::=	*AddExpr AddOperator MultExpr* | *MultExpr*	(E9)
AddOperator	::=	+ | −	(E10)
MultExpr	::=	*MultExpr MultOperator UnaryExpr* | *UnaryExpr*	(E11)
MultOperator	::=	* | /	(E12)
UnaryExpr	::=	*UnaryOperator PrimaryExpression* | *PrimaryExpression*	(E13)
UnaryOperator	::=	− | !	(E14)

In general the expression part is quite similar to other languages. The only thing that could be different is the precedence rules. We have chosen to use standard evaluation for arithmetic operators, and the same for boolean.The reason for this is that most people are already familiar with these precedence rules.

The *random* expression is introduced to AWL, since the programmer will need a way to make his ants take different choices. Otherwise all ants dedicated to a given ant type would follow the exact same pattern. *getProperty* is the counterpart to the command *setProperty*. It gives direct access to fetch values from any storage location, and can be very useful when implementing a standard environment.

2.10 Various

These rules describe the general types in AWL. An integer literal consist of an amount of digits. An identifier always begins with a letter and may be followed by an arbitrary number of letters and digits. A letter is an element in the English alphabet, and a number is a sequence of digits, which might be a floating point number. The Direction literal is included to be be able to specify a direction.

Literal	::=	*BoolLiteral* | *IntLiteral* | *DirectionLiteral*	(V1)
BoolLiteral	::=	**true** | **false**	(V2)
IntLiteral	::=	*Digit IntLiteral* | ϵ	(V3)
DirectionLiteral	::=	**left** | **right** | **up** | **down** | **center**	(V4)
Identifier	::=	*Letter* | *Identifier Letter* | *Identifier IntLiteral*	(V5)
Letter	::=	**a**|**b**|**c**|**d**|**e**|**f**|**g**|**h**|**i**|**j**|**k**|**l**|**m**|**n**|**o**|**p**|**q**|**r**|**s**|**t**|**u**|**v**|**w**|**x**|**y**|**z**	(V6)
	|	**A**|**B**|**C**|**D**|**E**|**F**|**G**|**H**|**I**|**J**|**K**|**L**|**M**|**N**|**O**|**P**|**Q**|**R**|**S**|**T**|**U**|**V**|**W**|**X**|**Y**|**Z**	
Digit	::=	**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**	(V7)

We have decided to include some of the most common types. Besides that we have included a Direction literal, which will be useful, and more intuitive to use when you for example want to specify, the direction of an ant, rather than using numbers.

Table 2.1 on the facing page show how a program can be build from the syntax just defined. The left column shows the code, while the right column contains some comments on what is happening.

2.11 Summary

In this chapter we have defined the grammar for AWL. We have tried to define a syntax, which is simpler than that of other existing high level languages. On the other hand, we also wanted to use some constructs

world{500,200,100}{	**World creation** A new world is created with the size 500x500. Each team will have a maximum of 200 ants, and there will be a maximum of 100 pieces of food in the world at any given time.

common var cx : integer = 0;
teambrain var tx : integer = 0;
private var px : integer = 0;
	Ant memory is declared
	Common variables are stored one place in memory. Teambrain variables are stored once for every created team, and private variables are stored once for each ant.

rule myRule(var d : integer){
var x : integer = 0;
r = r + d;
}
	The rule myRule is declared
	The rule name is associated with the formal parameters, the declarations and the commands. The current variable environment is also stored with the rule (static variable binding).

anttype myAntType{
var y : integer = 0;
r(left);
}
	The ant type myAntType is declared
	The type name is associated with the declarations and the commands. The current variable environment is also stored with the ant type.

main{
createTeam(team1);
createAnt(team1);
process(team1, 0, myAntType);
}
	The main section of the program.
	A new team is created.
	A ant is created for team1.
	Ant 0 on team1 is processed using the ant type myAntType

}	**End of the world**

Table 2.1: Example AWL program

of general purpose languages, since users will be familiar with those. The purpose of the syntax is also to support the conceptual ideas in the problem area of this report - namely defining an ant world. Clearly AWL have no usage besides defining a world of ants.

The primary influences from modern high level languages, such as C and Java, are the basic variable types and the selection and loop constructs. We have left out several data types, like floating point numbers, to keep the programming language relatively simple.

We have aimed at constructing a language with some level of encapsulation. Each ant and team will explore the world on its own, so they are defined with their own scope of variables. Each ant will only have access to declared rules, which are defined by the programmer of the actual world. This has the effect of encapsulating the ants. Since it is not necessary to have such encapsulation in other parts of the programming language, we have decided to use this solution rather than a more general construct.

As the process has evolved we have realized that even simple additions to the language syntax can have a huge impact on the complexity of the language. Especially in the operational semantics, which is the subject of the next chapter.

Chapter 3

Operational Semantics for AWL

In this chapter we will define the operational semantics for AWL.

Defining the operational semantics for any programming language consists of the following steps:

- Defining an abstract syntax with syntactic categories and constructs.
- Defining semantical sets and functions.
- Defining transition systems.

Operational semantics tells us *how* to execute our program, and thus how it should be understood.

When all this is done we will describe how AWL can be extended to include a standard environment.

3.1 Big Step Semantics

The semantics defined in this chapter is big step (or natural) semantics. This means that a whole calculation is done in one transition.

We will define the configurations for a transition system in the following manner (example):

$$\Gamma_{DecVar} = (DecVar \times Env_V \times Store) \cup Env_V \times Store$$

which means that a configuration in the category *DecVar* consists of one or more declarations, a variable environment with updated variable bindings and a store with updated storage bindings.

We will use the following notation to define our transitions:

$$\frac{Premises}{Conclusion}$$

where
Conditions

An example of this could be the transition for multiplying two arithmetic expressions:

$$\frac{env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_V, sto \vdash ae_1 * ae_2 \rightarrow_{ae} z}$$
where $z = z_1 \cdot z_2$

The above transition should be read like "ae_1 multiplied with ae_2 will give the result z, if ae_1 evaluates to z_1 and ae_2 evaluates to z_2, with the condition that $z_1 \cdot z_2$ equals z".

3.2 Abstract Syntax

The semantics of a programming language is based on the syntax of the language. Since we don't need to check a program for syntactical errors at this point, we don't need the entire concrete syntax from the previous chapter. Instead we will define an abstract syntax, whose purpose is to describe the structure of the different syntactical constructs. Later we can define semantic rules for each syntactical construct, and group them by their categories.

3.2.1 Syntactic Categories

For each category we specify a meta variable. In the category Var the meta variable is x, which means that when x is written in the syntactical rules, it could be any variable in Var.

We have three different kinds of literals in AWL – integer literals, boolean literals and direction literals.

$n \in$	$IntLit$	(Integer literals)
$bl \in$	$BoolLit$	(Boolean literals)
$dl \in$	$DirLit$	(Direction literals)

Likewise, we have three kinds of expressions. We also define the category exp, which is the set of all expressions. We will use this to shorten our syntactical rules later in this chapter.

$ae \in$	$AExpr$	(Arithmetic expressions)
$be \in$	$BExpr$	(Boolean expressions)
$de \in$	$DExpr$	(Direction expressions)
$exp \in$	$AExpr \cup BExpr \cup DExpr$	(All expressions)

Variables, rules, turns, teams and ant types are identified with names in AWL,

$x \in$	Var	(Variables)
$r \in$	$RuleName$	(Non turn based rule names)
$t \in$	$TurnName$	(Turn based rule names)
$at \in$	$AntTypeName$	(Ant type names)

and can all be declared.

$D_V \in$	$DecVar$	(Variable declarations)
$D_A \in$	$DecArray$	(Array declarations)
$D_{MC} \in$	$DecMemCommon$	(Common Memory declarations)
$D_{MT} \in$	$DecMemTeam$	(Teambrain Memory declarations)
$D_{MP} \in$	$DecMemPrivate$	(Private Memory declarations)
$D_R \in$	$DecRule$	(Non turn based rule declarations)
$D_T \in$	$DecTurn$	(Turn based rule declarations)
$D_{TEAM} \in$	$DecTeam$	(Team declarations)
$D_{AT} \in$	$DecAntType$	(Ant type declarations)
$type \in$	$Type$	(Types)

Rules and turns can have multiple parameters, so we define a category for the formal parameters and a category for the actual parameters.

$P_F \in$	$FParm$	(Formal parameter list)
$P_A \in$	$AParm$	(Actual parameter list)

And last, but not least, we have the commands and the world construct.

$S \in$	Com	(Commands)
$w \in$	$World$	(World program)

3.2.2 Constructs for the Syntactical Categories

In this section we will define the structure of the elements in the syntactical categories. The structure of literals, variables and the names of rules, turns, ant types and teams are given in the concrete syntax, and since those categories have no real semantic value, we will not specify their structure again here.

The interesting categories are the ones with actual semantic value. Table 3.1 shows the structure of the constructs for those categories.

ae	$::=$	$n \mid x \mid x[ae] \mid \textbf{cmem}\,x \mid \textbf{tmem}\,x \mid \textbf{pmem}\,x \mid r(P_A) \mid ae_1{+}ae_2 \mid ae_1{-}ae_2$
	$\mid$	$ae_1{*}ae_2 \mid ae_1/ae_2 \mid (ae) \mid \textbf{getProperty}(ae)$
be	$::=$	$bl \mid x \mid x[ae] \mid \textbf{cmem}\,x \mid \textbf{tmem}\,x \mid \textbf{pmem}\,x \mid r(P_A) \mid ae_1{==}ae_2 \mid ae_1{<}ae_2$
	$\mid$	$ae_1{>}ae_2 \mid ae_1{<=}ae_2 \mid ae_1{>=}ae_2 \mid ae_1! =ae_2 \mid be_1{==}be_2$
	$\mid$	$be_1! =be_2 \mid !be \mid (be) \mid de_1{==}de_2 \mid de_1! =de_2 \mid \textbf{getProperty}(ae)$
de	$::=$	$dl \mid x \mid x[ae] \mid \textbf{cmem}\,x \mid \textbf{tmem}\,x \mid \textbf{pmem}\,x \mid r(P_A) \mid (de)$
	$\mid$	$\textbf{getProperty}(ae)$
S	$::=$	$x{=}exp; \mid r(P_A); \mid \textbf{endturn}\,t(P_A);$
		$\textbf{cmem}\,x = exp; \mid x[ae] = exp;$
	$\mid$	$\textbf{tmem}\,x = exp; \mid \textbf{pmem}\,x\ = exp; \mid S_1\ S_2 \mid \textbf{skip};$
	$\mid$	$\textbf{return}\,exp; \mid \textbf{if}(be)\ \{S_1\}\ \textbf{else}\ \{S_2\}$
	$\mid$	$\textbf{while}(be)\ \{S\} \mid \textbf{process}(ae, ae, at); \mid \textbf{setProperty}(ae, exp)$
D_V	$::=$	$\textbf{var}\,x : type = exp;\ D_V \mid \epsilon$
D_A	$::=$	$\textbf{array}\,x[n] : type = exp; \mid \epsilon$
D_R	$::=$	$\textbf{rule}\,r(P_F)\ \{D_V\,D_A\,S\}\ D_R \mid \textbf{rule}\,r(P_F) : type\ \{D_V\,D_A\,S\}\ D_R \mid \epsilon$
D_T	$::=$	$\textbf{turn}\,t(P_F)\ \{D_V\,D_A\,S\}\ D_T \mid \epsilon$
D_{AT}	$::=$	$\textbf{anttype}\,at\ \{D_V\,D_A\,S\}\ D_{AT} \mid \epsilon$
D_{TEAM}	$::=$	$\textbf{createTeam}(x);\ D_{TEAM} \mid \epsilon$
D_{MC}	$::=$	$\textbf{private var}\,x : type;\ D_{MC} \mid \epsilon$
D_{MT}	$::=$	$\textbf{teambrain var}\,x : type;\ D_{MT} \mid \epsilon$
D_{MP}	$::=$	$\textbf{common var}\,x : type;\ D_{MP} \mid \epsilon$
P_F	$::=$	$\textbf{var}\,x : type;\ P_F \mid \epsilon$
P_A	$::=$	$exp;\ P_A \mid \epsilon$
w	$::=$	$\textbf{world}(z_1, z_2)\{D_{MC}\ D_{MT}\ D_{MP}\ D_T\ D_R\ D_{AT}\ \textbf{main}\{D_{TEAM}\ D_V\ D_A\ S\}\}$

Table 3.1: Abstract syntax for AWL

3.3 Environment

We will use the basics of the environment-store model[1], when defining our environments. In short this means that e.g. variables bind to storage locations, and storage locations bind to values.

So we have a set of locations **Loc** (like the memory of a normal computer). We will call elements in this set l. Also, we have a function which, given a location, will return a value. The values of our model are the natural numbers $\mathbb{Z}$, boolean values B and directions D.

$$\textbf{Store} = \textbf{Loc} \hookrightarrow \textbf{Z} \cup \textbf{B} \cup \textbf{D}$$

Elements in **Store** are called *sto*. Note that the relationship between **Store** and **Loc** is a partial function, since not all locations need to have a value. We assume that we have an unlimited source of storage locations.

We define an update notation for **Store**. The store $sto\,[l \mapsto v]$ is the store sto' defined by

$$sto'\,(l') = \begin{cases} sto\,(l') & \text{if } l' \neq l \\ v & \text{if } l' = l \end{cases}$$

[1] [1, p. 86]

$WORLD$	=	0
$MAXANTS$	=	1
$MAXFOOD$	=	2
$TEAMCOUNT$	=	3
$CURRENTTEAM$	=	4
$CURRENTANT$	=	5
$COMMONDECLS$	=	6
$TEAMBRAINDECLS$	=	7
$PRIVATEDECLS$	=	8
$FOODBASE$	=	9

Table 3.2: Location constants.

$ANTSIZE$	=	$ANTALLOC + sto(PRIVATEDECLS)$ This calculation results in the number of storage location s allocated to each ant
$TEAMANTSSIZE$	=	$sto(ANTCOUNT) \cdot ANTSIZE$ This calculation results in the number of storage locations allocated to all ants on each team.
$FIRSTTEAM$	=	$COMMONBASE + sto(COMMONDECLS) + sto(MAXFOOD) \cdot 2$ This calculation results in the location where the first team is allocated.
$TEAMSIZE$	=	$TEAMALLOC + sto(TEAMBRAINDECLS) + TEAMANTSIZE$ This calculation results in the number of storage locations allocated to each team.
$COMMONBASE$	=	$FOODBASE + FOODCOUNT \cdot 2$ This calculation results in the base storage location of common memory variables.

Table 3.3: Predefined calculations.

3.3.1 Storage Structure

The data elements of AWL has a somewhat predefined storage location. The first section of memory is reserved to a number of keys values, such as the size of the ant world and the maximum number of ants. In this part of the memory we also find the common ant memory variables, and the ant food coordinates.

The next section of memory is dedicated to the ant teams. Each team have some basic describing values, and their own copy of the declared teambrain variables. Also each team has allocated storage for the maximum number of ants possible, and each ant has its own copy of the declared private variables (as well as its coordinates). Figure 3.1 shows the structure of the storage.

Because of this structure, some semantic rules need to make list of calculations. To simplify the calculations we define location constants to use instead of numbers. The constants are defined in table 3.2.

Looking at figure 3.1 we also see that an ant has two coordinates allocated (besides the private memory), and that a team has four. We define the constant values $ANTALLOC = 2$ and $TEAMALLOC = 4$.

Even with the defined constants, the calculations can get quite long. Even though each small calculation is not complicated, a long list of simple calculations still looks confusing. We will therefore predefine some of the calculations here, and refer to them in the semantic rules. The predefined calculations are shown in table 3.3. To understand the calculations refer to storage structure illustrated in figure 3.1.

WORLD	0	World size
MAXANTS	1	Max number of ants
MAXFOOD	2	Max pieces of food
TEAMCOUNT	3	Number of teams
CURRENTTEAM	4	Current team
CURRENTANT	5	Current ant
COMMONDECLS	6	Common variable count
TEAMDECLS	7	Team variable count
PRIVATEDECLS	8	Private variable count
FOODBASE	9	food - 0 -X
	10	food - 0 - Y
	11	food - 1 -X
	12	food - 1 - Y
	13	food - I -X
	14	food - I -Y
COMMONBASE	15	Common var - 0
	16	Common var -1
	17	Common var -i

STARTTEAM	18	Team - 0 - no
	19	Team - 0 - nextant
	20	Team - 0 - Base X
	21	Team -0 - Base Y
	22	Team -0-team var-0
	23	Team -0-team var-1
	24	Team -0-team var-j
	25	Ant - 0 - X Coord
	26	Ant - 0 - Y Coord
	27	Ant - 0 - private var - 0
	28	Ant - 0 - private var -1
	29	Ant - 0 - private var - k
	30	Ant - 1 - X Coord
	31	Ant - 1- Y Coord
	32	Ant - 1 - private var - 0
	33	Ant - 1 - private var -1
	34	Ant - 1 - private var - k
	35	Ant - i - X Coord
	36	Ant - i - Y Coord
	37	Ant - i - private var - 0
	38	Ant - i - private var -1
	39	Ant - i - private var - k
	40	Team - 1 - no
	41	Team - 1 - nextant
	42	Team - 1 - Base X
	43	Team -1 - Base Y
	44	Team -1-team var-0
	45	Team -1-team var-1
	46	Team -1-team var-j

Figure 3.1: Memory ordering with locations

3.3.2 Variable Environment

Our variable environment is a partial function from variables to locations (partial because not all possible variable names are necessarily bound to a location). We have to consider arrays, which means that we need to keep track of array sizes. Also in the semantic rules regarding ant memory variables, we need to store an index number (this is elaborated in the particular section).

There are three different variable types in AWL, and this should also be stored in the variable environment. We can now define the set of variable environments as

$$\begin{aligned}\mathbf{Env_V} = \quad & (\mathbf{Var} \hookrightarrow \mathbf{Type} \times \mathbf{Loc}) \cup \\ & (\mathbf{Var} \hookrightarrow \mathbf{Type} \times \mathbf{Loc} \times Z) \cup \\ & (\mathbf{Var} \hookrightarrow \mathbf{Type} \times Z) \cup \\ & (\{next, return\} \hookrightarrow \mathbf{Loc})\end{aligned}$$

We introduce the elements *next* and *return* with a special purpose in our variable environment. The element *next* is used as a pointer to the next free location. The element *return* is used to store return values for rules in AWL.

We define an update notation for $\mathbf{Env_V}$. The environment $env_V\,[x \mapsto (type, l)]$ is the environment env'_V defined by

$$env'_V\,(y) = \begin{cases} env_V\,(y) & \text{if } y \neq x \\ (type, l) & \text{if } y = x \end{cases}$$

The same notation applies for the rest of this environment as well as the rest of the environments defined in this report.

3.3.3 Procedure Environment

There are two different procedure-like constructs in AWL - rules and ant types. We will store both of them in the same procedure environment.

Rules in AWL can be both TB (turn based) rules and NTB (non turn based) rules. Both can have multiple parameters, but only NTB rules can return a value. Rules in AWL have static variable bindings. Since it is impossible to declare additional rules after initializing the main section of a program, it is meaningless to have static rule bindings, so we choose to have dynamic rule bindings.

An ant type is similar to a rule, except it does not take any parameters, and it can not return any value. Like rules, ant types have static variable bindings, dynamic rule bindings.

With that in mind, the definition of our procedure environment looks like this.

$$\begin{aligned}\mathbf{Env_P} = \quad & (\mathbf{RuleName} \hookrightarrow \mathbf{Com} \times \mathbf{FParm} \times \mathbf{Env_V} \times \mathbf{DecVar} \times \mathbf{DecArray} \times \mathbf{Type} \cup \\ & (\mathbf{RuleName} \hookrightarrow \mathbf{Com} \times \mathbf{FParm} \times \mathbf{Env_V} \times \mathbf{DecVar} \times \mathbf{DecArray}) \cup \\ & (\mathbf{TurnName} \hookrightarrow \mathbf{Com} \times \mathbf{FParm} \times \mathbf{Env_V} \times \mathbf{DecVar} \times \mathbf{DecArray}) \cup \\ & (\mathbf{AntTypeName} \hookrightarrow \mathbf{Com} \times \mathbf{Env_V} \times \mathbf{DecVar} \times \mathbf{DecArray})\end{aligned}$$

For each procedure we store the commands, the parameters, the current variable environment, and the variable and array declarations in our rule environment[2]. For rules with a return type, we store return type as well.

At the time of procedure declaration, only ant memory variables will have been previously declared. This means that these are the only outside variables that can be accessed inside a procedure (disregarding the *setProperty* command, which can alter any memory location).

[2]We also store the declarations because AWL does not have a nested block structure

3.3.4 Literal Functions

We have three kinds of literals in AWL; integer literals, boolean literals and direction literals. Since the literals are only syntactic representations, we need a way to define the meaning of a literal.

We define the function $\mathcal{N}$ which, for any integer literal, returns the corresponding numeric value.

$$\mathcal{N}: \quad \mathbf{IntLit} \rightarrow \mathbf{Z}$$

So $N[\![4]\!] = \mathit{4}$ where 4 is an integer literal and 4 is the numeric value 4 and so on.

For boolean literals we define the function $\mathcal{B}$.

$$\mathcal{B}: \quad \mathbf{BoolLit} \rightarrow \mathbf{Bool}$$

where $\mathcal{B}[\![true]\!] = tt$ and $\mathcal{B}[\![false]\!] = ff$. So $true$ is a syntactic representation and tt is the value. The same applies to $false$ and ff.

Last, we have direction literals for which we define the function $\mathcal{D}$.

$$\mathcal{D}: \quad \mathbf{DirLit} \rightarrow \mathbf{Dir}$$

The specific definition of $\mathcal{D}$ is shown in the table below.

DirLit	**Dir**
left	*ll*
right	*rr*
up	*uu*
down	*dd*
center	*cc*

3.3.5 Other Functions

To make our semantic rules as simple as possible, we define different functions that we use when building the rules.

Our variable environment contained the element $next$, which was a pointer to the next free storage location. However we have to update $next$ manually, and for that we need a function. We define the function new.

$$new: \quad (\mathbf{Loc} \cup \mathbf{Loc} \times \mathbf{Z} \rightarrow \mathbf{Loc})$$

and more specific

$$new\,(l) = l + 1 \text{ and } new\,(l, z) = l + z$$

Another function which is used is the ran function. This function takes a natural number greater than zero and returns a value in the range 0 to the incoming number minus one. The formal description of this function is as follows.

$$ran: \quad \mathbf{Z} \rightarrow \mathbf{Z}$$

Finally we need to define two functions, which will help us determine the base storage location of a specific team and a specific ant. AWL has a well defined storage structure, which means that calculations are needed to reach these base locations. We define the two functions $teamLoc$ and $antLoc$.

$$\begin{aligned} teamLoc: &\quad \mathbf{Z} \to \mathbf{Z} \\ antLoc: &\quad \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z} \end{aligned}$$

We define the precise definition of *teamLoc* to be

$$teamLoc(z_{team}) = FIRSTTEAM + z_{team} \cdot TEAMSIZE$$

and the precise definition of *antLoc* to be

$$antLoc(z_{team}, z_{ant}) = teamLoc(z_{team}) + z_{ant} \cdot ANTSIZE$$

3.4 Transition Systems

In this section we will define the transition systems of our operational semantics. We will define one transition system per syntactical category (only categories with a semantic value). For each system, we will define the configurations, the transition relation, and the end configurations.

Each transition system is defined by a 3-tuple:

$$(\Gamma, \to, T)$$

where Γ is the configurations (states) of the transition system, and T is the end configurations. $\to$ is the transition relation, which defines how to get from one configuration to another.

The transition relation for a given transition system will be defined by creating a semantical rule for every syntactical construct.

As a last note before going through the different syntactical categories, we define a semantical rule [Expression] that acts as a synonym for the three different kinds of expressions in AWL. This means that whenever we write

$$env_P, env_V.sto \vdash exp \to_{exp} v$$

it covers the following:

$$\begin{aligned} env_P, env_V, sto &\vdash ae \to_{ae} z \\ env_P, env_V, sto &\vdash be \to_{be} b \\ env_P, env_V, sto &\vdash de \to_{de} d \end{aligned}$$

3.4.1 Arithmetic Expressions

The transition system for arithmetic expressions should evaluate arithmetic expressions to their values, which are numbers. So we define the transition system $(\Gamma_{AExpr}, \to_{ae}, T_{AExpr})$, where $\Gamma_{AExpr} = AExpr \cup \mathbb{Z}$ and $T_{AExpr} = \mathbb{Z}$.

Transitions are on the form $env_P, env_V, sto \vdash ae \to_{ae} z$.

We define $\to_{ae}$ by the semantical rules in table 3.4.

The rule [ae-add] shows that the syntactic construct $ae_1 + ae_2$ will evaluate to the number z, if ae_1 evaluates to the number z_1, and ae_2 evaluates to number z_2 where $z = z_1 + z_2$. Subtraction, multiplication and division follow the same pattern.

An arithmetic expression can be a single integer literal. In that case we use the rule [ae-lit], which states that the integer literal n will evaluate to the number z if $\mathcal{N}[\![n]\!] = z$. $\mathcal{N}$ was the function which given an integer literal returned the corresponding number. This rule is an axiom, since it has no premise, and

[ae-add]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 + ae_2 \rightarrow_{ae} z} \quad \text{where } z = z_1 + z_2$$

[ae-sub]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 - ae_2 \rightarrow_{ae} z} \quad \text{where } z = z_1 - z_2$$

[ae-mult]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 * ae_2 \rightarrow_{ae} z} \quad \text{where } z = z_1 \cdot z_2$$

[ae-div]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 / ae_2 \rightarrow_{ae} z} \quad \text{where } z = \frac{z_1}{z_2}$$

Table 3.4: Semantics for arithmetic calculations

[ae-par]

$$\frac{env_p, env_V, sto \vdash ae \rightarrow_{ae} z_1}{env_p, env_V, sto \vdash (ae) \rightarrow_{ae} z_1}$$

[ae-lit]

$$env_p, env_V, sto \vdash n \rightarrow_{ae} z \quad \text{if } \mathcal{N}[\![n]\!] = z$$

[ae-var]

$$env_p, env_V, sto \vdash x \rightarrow_{ae} z \quad \text{if } env_V(x) = (integer, l) \text{ and } sto(l) = z$$

[ae-array]

$$env_p, env_V, sto \vdash x[ae] \rightarrow_{ae} z \quad \begin{array}{l} \text{where } env_p, env_V, sto \vdash ae \rightarrow_{ae} z'' \\ \text{and if } env_V(x) = (integer, l, z') \text{ and} \\ z = sto(l + z'') \text{ and } z'' < z' \text{ and } z'' \geq 0 \end{array}$$

[ae-rulecall]

$$\frac{\begin{array}{l} \langle P_F, env'_V[next \mapsto new(l)] \rangle \rightarrow_{P_F} env''_V \\ env_P \vdash \langle P_A, env''_V[next \mapsto new(l)], sto \rangle \rightarrow_{P_A} (env^3_V, sto') \\ env_P \vdash \langle D_V, env^3_V, sto' \rangle \rightarrow_{D_V} (env^4_V, sto'') \\ env_P \vdash \langle D_A, env^4_V, sto'' \rangle \rightarrow_{D_A} (env^5_V, sto^3) \\ env_P, env^5_V[return \mapsto l] \vdash \langle S, sto^3 \rangle \rightarrow sto^4 \end{array}}{env_V, sto \vdash r(P_A) \rightarrow_{ae} z}$$

where $l = env_V(next)$ and $z = sto^4(env^5{}_V(return))$
and if $env_P(r) = (S, P_F, env'_V, D_V, D_A, integer)$

Table 3.5: Semantics for literals, parentheses, variables, arrays and rule calls.

[ae-getProperty] $env_p, env_V, sto \vdash \textbf{getProperty}(ae) \rightarrow_{ae} z$ where $env_V, sto \vdash ae \rightarrow_{ae} z_1$and $z = sto\,(z_1)$

Table 3.6: GetProperty for arithmetic expressions

[ae-common-var] $env_p, env_V, sto \vdash \textbf{cmem}\, x \rightarrow_{ae} z_1$ where $env_V\,(x) = (integer, z_2)$ and $z_1 = sto\,(COMMONBASE + z_2)$
[ae-team-var] $env_p, env_V, sto \vdash \textbf{tmem}\, x \rightarrow_{ae} z_1$ where $env_V\,(x) = (integer, z_2)$ and $z_1 = sto\,(teamLoc\,(sto\,(CURRENTTEAM)) + TEAMALLOC + z_2)$
[ae-private-var] $env_p, env_V, sto \vdash \textbf{pmem}\, x \rightarrow_{ae} z_1$ where $env_V\,(x) = (integer, z_2)$ and $z_1 = sto\,(antLoc\,(sto\,(CURRENTTEAM)\,, sto\,(CURRENTANT)) + ANTALLOC + z_2)$

Table 3.7: Memory variables for ae

would therefore be a leaf if we constructed a derivation tree of an AWL program, which included this construct. The rules [ae-var] and [ae-array] are also axioms.

Rule [ae-var] states that a variable x evaluates to the number z, if the variable is bound to storage location l in the variable environment and l is bound to z in the storage. The rules for arrays and rule calls are a little more complicated, so we will describe them more thoroughly.

To evaluate the arithmetic expression $x[ae]$, we first need to evaluate ae. Then we look up x in the variable environment, which returns the array type, the base storage location and the size of the array. Since we don't want to reference storage outside the boundaries of our array, we check that the index is greater than or equal to zero and smaller than or equal to the size of the array minus one (indexing starts at zero). If this is the case $x[ae]$ evaluates to the number z. Note that if the type of the array isn't integer, then this rule will not apply to the expression.

Only rules which have a return type can be expressions, so the rule we are calling is bound to a specific type. Like our semantic rule for arrays, [ae-rulecall] will only apply, if this rule returns an integer. If this is the case, we lookup the rule in the procedure environment $env_p\,(r)$ which returns all the information about the rule needed to execute it $(S, P_F, env'_V, D_V, D_A integer)$. First we execute the parameter declarations (if it has any), so each parameters get allocated a storage location. We then use the updated variable environment and storage when executing the local variable and array declarations. We also allocate a location for the return value. With all this done, we execute the commands of the rule, which hopefully will put a arithmetic value in the location allocated for the return value. The rule then evaluates to the return value.

In table 3.6 we use **getProperty**(ae) to retrieve a value from the storage. The expression takes another arithmetic expression to look up a specific location in the storage. This gives the programmer access to the entire memory.

In table 3.7 we show how to get the value from a ant memory variable. We find the value of a common memory variable by first using the function $env_V(x)$, which will return a type (in this case $integer$) and a number z_2, which is the relative address of x. Since the common memory variables have the base address $COMMONBASE$, we add z_2 to the base. We apply sto and get that $z = sto\,(COMMONBASE + z_1)$.

The principle is the same for [ae-team-var] and [ae-private-var] except that we are using a different base value. The base values are determined using the functions $teamLoc$ and $antLoc$ defined in this chapter.

[ae-random]

$env_p, env_V, sto \vdash \textbf{random}(ae) \rightarrow_{ae} z$

where $env_p, env_V, sto \vdash ae \rightarrow_{ae} z'$ and $z = ran(z')$

Table 3.8: Random command

[be-equals-ae-1]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 == ae_2 \rightarrow_{be} tt}$$

where $z_1 = z_2$

[be-equals-ae-2]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 == ae_2 \rightarrow_{be} ff}$$

where $z_1 \neq z_2$

[be-not-equals-ae-1]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 != ae_2 \rightarrow_{be} tt}$$

where $z_1 \neq z_2$

[be-not-equals-ae-2]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 != ae_2 \rightarrow_{be} ff}$$

where $z_1 = z_2$

Table 3.9: Boolean expressions for arithmetic equality

The random command in table 3.8 takes an ae expression and returns a random value z. This value is found by first having ae evaluated to a value z' and then applying this value to random function ran. $ran(z')$ returns a value z where $0 \leq z < z'$.

3.4.2 Boolean Expressions

Boolean expressions are expressions, which evaluate to truth/boolean values (ff or tt). So the transition system for $BExpr$ should evaluate a boolean expression to a boolean value b. We define the system $(\Gamma_{BExpr}, \rightarrow_{be}, T_{BExpr})$, where the configurations $\Gamma_{BExpr} = BExpr \cup \{tt, ff\}$ and the end configurations $T_{BExpr} = \{tt, ff\}$. Transitions are on the form $env_p, env_V, sto \vdash be \mapsto b$.

We define $\rightarrow_{be}$ by the semantical rules below. The rules are divided into smaller groups to maintain a good overview.

The rules in table 3.9 define how we determine if an arithmetic expression is or is not equal to another arithmetic expression. The rules [be-equals-ae-1] and [be-equals-ae-2] show that if we want to check whether ae_1 and ae_2 are equal to each other, we first evaluate the two expressions to the numbers z_1 and z_2. If these numbers are equal (in a mathematical sense), then the construct $ae_1 == ae_2$ evaluates to tt - otherwise it evaluates to ff.

In table 3.10 we perform relational arithmetics on two arithmetic expressions, ae_1 and ae_2. In [be-lower-than-1] and [be-lower-than-2] the two expressions are evaluated down to two values, z_1 and z_2. What happens next is that if $z_1 < z_2$, we can apply the rule [be-lower-than-1] and the rule will yield a tt. Otherwise we can apply the other be-lower-than rule [be-lower-than-2] and the value yielded is a ff. The other three pairs of rules are very similar to [be-lower-than] pair in their construct, and will also yield either tt or ff.

[be-greater-than-1]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 > ae_2 \rightarrow_{be} tt}$$

where $z_1 > z_2$

[be-greater-than-2]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ea_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 > ae_2 \rightarrow_{be} ff}$$

where $z_1 \leq z_2$

[be-lower-than-1]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 < ae_2 \rightarrow_{be} tt}$$

where $z_1 < z_2$

[be-lower-than-2]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 < ae_2 \rightarrow_{be} ff}$$

where $z_1 \geq z_2$

[be-greater-than-or-equals-1]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 => ae_2 \rightarrow_{be} tt}$$

where $z_1 \geq z_2$

[be-greater-than-or-equals-2]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 => ae_2 \rightarrow_{be} ff}$$

where $z_1 < z_2$

[be-less-than-or-equals-1]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 =< ae_2 \rightarrow_{be} tt}$$

where $z_1 \leq z_2$

[be-less-than-or-equals-2]

$$\frac{env_p, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2}{env_p, env_V, sto \vdash ae_1 =< ae_2 \rightarrow_{be} ff}$$

where $z_1 > z_2$

Table 3.10: Semantics for greater and lower-than constructs

[be-and-1]

$$\frac{env_p, env_V, sto \vdash be_1 \rightarrow_{be} tt \quad env_p, env_V, sto \vdash be_2 \rightarrow_{be} tt}{env_p, env_V, sto \vdash be_1 \quad and \quad be_2 \rightarrow_{be} tt}$$

[be-and-2]

$$\frac{env_p, env_V, sto \vdash be_i \rightarrow_{be} ff}{env_p, env_V, sto \vdash be_1 \quad and \quad be_2 \rightarrow_{be} ff}$$

where $i \in \{1, 2\}$

[be-or-1]

$$\frac{env_p, env_V, sto \vdash be_1 \rightarrow_{be} ff \quad env_p, env_V, sto \vdash be_2 \rightarrow_{be} ff}{env_p, env_V, sto \vdash be_1 \quad or \quad be_2 \rightarrow_{be} ff}$$

[be-or-2]

$$\frac{env_p, env_V, sto \vdash be_i \rightarrow_{be} tt}{env_p, env_V, sto \vdash be_1 \quad or \quad be_2 \rightarrow_{be} tt}$$

where $i \in \{1, 2\}$

Table 3.11: Semantics for 'and' and 'or' constructs

The semantical rules in table 3.11 takes two boolean expressions and compare them and, depending on the rule, return a *tt* or a *ff* value. In [be-and-1] we say that given the environments env_P, env_V and a store *sto*, both the boolean expressions be_1 and be_2 will evaluate to *tt*, and thus yield a *tt*. On the other hand, in [be-and-2] we state that if just one of the two expressions does not yield a *tt* when evaluated, then the rule will yield an *ff*.

The rule [be-and-2] is actually two rules put into one by the use of i, where i is the set of values $\{1, 2\}$. This means that either be_1 or be_2 will be evaluated to *ff*. Which one is of no consequence for the end result which would, in any case, be *ff*.

What happens in the [be-or] pair is just the opposite. If either of the two boolean expressions evaluate to *tt*, then the rule will yield a *tt*.

When a boolean literal *bl* is encountered, we can apply [be-lit], found in table 3.12. This applies the function $\mathcal{B}[\![bl]\!]$ and gets a b in return. b is either *tt* or *ff*. The description of this can be found in section 3.3.4.

[be-lit]

$env_p, env_V, sto \vdash bl \rightarrow_{be} b$
if $\mathcal{B}[\![bl]\!] = b$

[be-parenthesis]

$$\frac{env_p, env_V, sto \vdash be \rightarrow_{be} b}{env_p, env_V, sto \vdash (be) \rightarrow_{be} b}$$

[be-not-1]

$$\frac{env_p, env_V, sto \vdash be \rightarrow_{be} tt}{env_p, env_V, sto \vdash !be \rightarrow_{be} ff}$$

[be-not-2]

$$\frac{env_p, env_V, sto \vdash be \rightarrow_{be} ff}{env_p, env_V, sto \vdash !be \rightarrow_{be} tt}$$

Table 3.12: Semantics for boolean literals, parentheses and negations

[be-var]

$env_p, env_V, sto \vdash x \rightarrow_{be} b$ if $env_V\,(x) = l$ and $sto\,(l) = b$

[be-array]

$env_p, env_V, sto \vdash x[ae] \rightarrow_{ae} b$ where $env_P, env_V, sto \vdash ae \rightarrow_{ae} z''$
and if $env_V\,(x) = (boolean, l, z')$ and $b = sto\,(l + z'')'$
and $z'' < z'$ and $z'' \geq 0$

[be-rulecall]

$$\frac{\begin{array}{l} \langle P_F, env'_V\,[next \mapsto new\,(l)]\rangle \rightarrow_{P_F} env''_V \\ env_P \vdash \langle P_A, env''_V\,[next \mapsto new\,(l)], sto\rangle \rightarrow_{P_A} (env^3_V, sto') \\ env_P \vdash \langle D_V, env^3_V, sto'\rangle \rightarrow_{D_V} (env^4_V, sto'') \\ env_P \vdash \langle D_A, env^4_V, sto''\rangle \rightarrow_{D_A} (env^5_V, sto^3) \\ env_P, env^5_V\,[return \mapsto l] \vdash \langle S, sto^3\rangle \rightarrow sto^4 \end{array}}{env_V, sto \vdash r(P_A) \rightarrow_{be} b}$$

where $l = env_V\,(next)$ and $b = sto^4\,(env^5{}_V\,(return))$
and if $env_R\,(r) = (S, P_F, env'_V, D_V, D_A, integer)$

Table 3.13: Semantics for variable, array and rule calls

What happens in [be-parenthesis] is that the parenthesis are removed and the boolean expression be_1 is evaluated to a b. The two 'not' rules [be-not-1] and [be-not-2] are very similar. They take a boolean expression be, evaluate it to a b and then reverse the result, so that tt becomes ff and vice versa.

Assuming an environment env_V and a store sto we can, in the rule [be-var] in table 3.13, find the boolean b of a variable x, if the variable x is bound to the storage location l ($env_V\,(x) = l$) and l is bound to b in the storage $(sto\,(l) = b)$.

The other two rules in table 3.13 are similar to the ones in table 3.5.

In tables 3.14 and 3.15 rules resembling those in table 3.9 are defined. What makes them different is the types of expressions that are compared. In [be-equals(be)-1] and [be-equals(be)-2] we have two boolean expressions be_1 and be_2. These will – given an instance of the variable environment env_V, the procedure environment env_P and an instance of the store sto – evaluate to b_1 and b_2. b_1 and b_2 are then compared and the result is returned. In [be-not-equals(be)-1] and [be-not-equals(be)-2] the same happens, except that the result is negated.

Table 3.16 is identical to 3.6, except that it evaluates to a boolean value. The same applies to table 3.17 which shows the semantic rules of accessing boolean ant memory variables.

3.4.3 Direction Expressions

Directional expressions resemble boolean expressions in that they can only result in a finite amount of values. These values are (cc, ll, rr, uu, dd). The transition system is defined as $(\Gamma_{DExpr}, \rightarrow_{de}, T_{DExpr})$, where the configurations $\Gamma_{DExpr} = DExpr \cup \{cc, ll, rr, uu, dd\}$, and the end configuration T_{DExpr} is the set $\{cc, ll, rr, uu, dd\}$. We define our transitions to have the form $env_P, env_V, sto \vdash de \mapsto d$, which means

[be-equals(be)-1]

$$\frac{env_p, env_V, sto \vdash be_1 \rightarrow_{be} b_1 \quad env_p, env_V, sto \vdash be_2 \rightarrow_{be} b_2}{env_p, env_V, sto \vdash be_1 == be_2 \rightarrow_{be} tt}$$

where $b_1 = b_2$

[be-equals(be)-2]

$$\frac{env_p, env_V, sto \vdash be_1 \rightarrow_{be} b_1 \quad env_p, env_V, sto \vdash be_2 \rightarrow_{be} b_2}{env_p, env_V, sto \vdash be_1 == be_2 \rightarrow_{be} ff}$$

where $b_1 \neq b_2$

[be-not-equals(be)-1]

$$\frac{env_p, env_V, sto \vdash be_1 \rightarrow_{be} b_1 \quad env_p, env_V, sto \vdash be_2 \rightarrow_{be} b_2}{env_p, env_V, sto \vdash be_1 != be_2 \rightarrow_{be} tt}$$

where $b_1 \neq b_2$

[be-not-equals(be)-2]

$$\frac{env_p, env_V, sto \vdash be_1 \rightarrow_{be} b_1 \quad env_p, env_V, sto \vdash be_2 \rightarrow_{be} b_2}{env_p, env_V, sto \vdash be_1 != be_2 ff}$$

where $b_1 = b_2$

Table 3.14: Semantics for equality of boolean expressions

[be-equals(de)-1]

$$\frac{env_p, env_V, sto \vdash de_1 \rightarrow_{de} d_1 \quad env_p, env_V, sto \vdash de_2 \rightarrow_{de} d_2}{env_p, env_V, sto \vdash de_1 == de_2 \rightarrow_{be} tt}$$

where $d_1 = d_2$

[be-equals(de)-2]

$$\frac{env_p, env_V, sto \vdash de_1 \rightarrow_{de} d_1 \quad env_p, env_V, sto \vdash de_2 \rightarrow_{de} d_2}{env_p, env_V, sto \vdash de_1 == de_2 \rightarrow_{be} ff}$$

where $d_1 \neq d_2$

[be-not-equals(de)-1]

$$\frac{env_p, env_V, sto \vdash de_1 \rightarrow_{de} d_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{de} d_2}{env_p, env_V, sto \vdash de_1 != de_2 \rightarrow_{be} tt}$$

where $d_1 \neq d_2$

[be-not-equals(de)-2]

$$\frac{env_p, env_V, sto \vdash de_1 \rightarrow_{de} d_1 \quad env_p, env_V, sto \vdash ae_2 \rightarrow_{de} d_2}{env_p, env_V, sto \vdash de_1 != de_2 \rightarrow_{be} ff}$$

where $d_1 = d_2$

Table 3.15: Semantics for equality of direction expressions

[be-getProperty]

$env_p, env_V, sto \vdash \mathbf{getProperty}(ae) \rightarrow_{ae} b$

where $env_P, env_V, sto \vdash ae \rightarrow_{ae} z$ and $b = sto\,(z)$

Table 3.16: getProperty for boolean expression

[be-common-var]

$env_p, env_V, sto \vdash \mathbf{cmem}\, x \rightarrow_{ae} b$

where $(integer, z) = env_V\,(x)$ and $b = sto\,(COMMONBASE + z)$

[be-team-var]

$env_p, env_V, sto \vdash \mathbf{tmem}\, x \rightarrow_{ae} b$

where $(integer, z) = env_V\,(x)$ and

$b = sto\,(teamLoc\,(sto\,(CURRENTTEAM)) + TEAMALLOC + z)$

[be-private-var]

$env_p, env_V, sto \vdash \mathbf{pmem}\, x \rightarrow_{ae} b$

where $(integer, z) = env_V\,(x)$ and

$b = sto\,(antLoc\,(sto\,(CURRENTTEAM)\,, sto\,(CURRENTANT)) + ANTALLOC + z)$

Table 3.17: Memory variables for boolean expressions

[de-lit]
$env_P, env_V, sto \vdash dl \rightarrow_{de} d$
where $\mathcal{B}[\![dl]\!] = d$

[de-parenthesis]
$$\frac{env_P, env_V, sto \vdash de \rightarrow_{de} d}{env_P, env_V, sto \vdash (de) \rightarrow_{de} d}$$

[de-rulecall]
$$\frac{\begin{array}{l}\langle P_F, env'_V\,[next \mapsto new\,(l)]\rangle \rightarrow_{P_F} env''_V \\ env_P \vdash \langle P_A, env''_V\,[next \mapsto new\,(l)]\,, sto\rangle \rightarrow_{P_A} (env^3_V, sto') \\ env_P \vdash \langle D_V, env^3_V, sto'\rangle \rightarrow_{D_V} (env^4_V, sto'') \\ env_P \vdash \langle D_A, env^4_V, sto''\rangle \rightarrow_{D_A} (env^5_V, sto^3) \\ env_P, env^5_V\,[return \mapsto l] \vdash \langle S, sto^3\rangle \rightarrow sto^4\end{array}}{env_V, sto \vdash r(P_A) \rightarrow_{de} d}$$
where $l = env_V\,(next)$ and $d = sto^4\,(env^5{}_V\,(return))$
and if $env_R\,(r) = (S, P_F, env'_V, D_V, D_A, direction)$

Table 3.18: Semantics for literals, parenthesis and rule calls

[de-getProperty]
$env_V, sto \vdash \textbf{getProperty}(ae) \rightarrow_{de} d$
where $env_V, sto \vdash ae \rightarrow_{ae} z_1$
$d = sto\,(z_1)$

Table 3.20: getproperty for direction expression

that a direction expression will give a d given a variable environment env_V, a procedure environment env_p and a store sto.

In the rules below we will be giving the semantic rules for direction expressions $\rightarrow_{de}$. Since all the semantic rules for this transition system are almost identical to those of the already defined systems, they will stand uncommented.

[de-var]
$env_P, env_V, sto \vdash x \rightarrow_{de} d$ if $env_V\,(x) = (direction, l)$ and $sto\,l = d$

[de-array]
$env_P, env_V, sto \vdash x[ae] \rightarrow_{de} d$ where $env_P, env_V, sto \vdash ae \rightarrow_{ae} z''$
and if $env_V\,(x) = (direction, l, z')$ and $d = sto\,(l + z'')$
and $z'' < z'$ and $z'' \geq 0$

Table 3.19: Semantics for [de-var] and [de-array]

3.4.4 Variable Declarations

We define the transition system for DecVar to the 3-tuple $(\Gamma_{DecVar}, \rightarrow_{D_V}, T_{DecVar})$. The configurations Γ_{DecVar} are defined as $(DecVar \times Env_V \times Store) \cup Env_V \times Store$, and the end configuration $T_{DecVar} = Env_V \times Store$.

The transition relation is on the following form

$$env_P \vdash \langle D_V, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto')$$

[de-common-var]
$env_P, env_V, sto \vdash \textbf{cmem}\, x \rightarrow_{de} d$
where $env_V\,(x) = (integer, z)$ and $d = sto\,(COMMONBASE + z)$

[de-team-var]
$env_P, env_V, sto \vdash \textbf{tmem}\, x \rightarrow_{de} d$
where $env_V\,(x) = (integer, , z)$ and
$d = sto\,(teamLoc\,(sto\,(CURRENTTEAM)) + TEAMALLOC + z)$

[de-private-var]
$env_P, env_V, sto \vdash \textbf{pmem}\, x \rightarrow_{de} d$
where $env_V\,(x) = (integer, z)$ and
$d = sto\,(antLoc\,(sto\,(CURRENTTEAM)\,, sto\,(CURRENTANT)) + ANTALLOC + z)$

Table 3.21: memory variables for direction

and is defined by the semantic rules below. Note that normal variables in AWL must by declared with a value. This is the reason that the transitions alter both the variable environment and the store.

[Dv-variable declaration]

$$\frac{env_P \vdash \langle D_V, env_V\,[x \mapsto (type, l)]\,[next \mapsto new\,(l)]\,, sto\,[l \mapsto v]\rangle \rightarrow_{D_V} (env'_V, sto')}{env_P \vdash \langle \textbf{var} \quad x \quad : \quad type = exp; D_V, env_V, sto\rangle \rightarrow_{D_V} (env'_V, sto')}$$

where $env_P, env_V, sto \vdash exp \rightarrow_{exp} v$ and $l = env_V\,(next)$

[Dv-variable-declaration-empty]
$env_P \vdash \langle \varepsilon, env_V, sto\rangle \rightarrow_{D_V} (env_V, sto)$

Table 3.22: Variable declaration

Table 3.22 shows that the declaration of a variable results in

- the variable name is bound to the next free storage location, and
- that storage location is bound to the value of the variable.

The pointer to the next free location *next* is updated to point on the next free address. We use the function *new* to accomplish this. We define the empty declaration rule to end a list of variable declarations.

3.4.5 Array Declarations

For the transition system $(\Gamma_{DecArray}, \rightarrow_{D_A}, T_{DecArray})$ we have the following configurations

$$\Gamma_{DecArray} = (DecArray \times Env_V \times Store) \cup Env_V \times Store$$

and the following end configurations

$$T_{DecVar} = Env_V \times Store$$

This is similar to that of variable declarations. The transition relation for this category is on the form

$$env_P \vdash \langle D_A, env_V, sto\rangle \rightarrow_{D_A} (env'_V, sto')$$

The transition results in a changed variable environment env'_V and a changed store sto', since arrays must be declared with a value The transition relation is defined by the semantic rules below.

[Da-array-declaration]

$$\frac{env_P \vdash \langle D_A, env_V[x \mapsto (type, l, z)][next \mapsto new\,(l, z))], sto[l_i \mapsto v]\rangle \rightarrow_{D_A} (env'_V, sto')}{env_P \vdash \langle \textbf{array}\; x[n] :type{=}exp; D_A, env_V, sto\rangle \rightarrow_{D_A} (env'_V, sto')}$$

where
$env_P, env_V, sto \vdash exp \rightarrow_{exp} v$ and $env_P, env_V, sto \vdash n \rightarrow_{ae} z$ and
$i \in [0..z-1]$ and $l = env_V\,(next)$ and $z > 0$

[Da-array-declaration-empty]

$$env_P \vdash \langle \varepsilon, env_V, sto\rangle \rightarrow_{D_A} (env_V, sto)$$

Table 3.23: Array declaration

In table 3.23 we show how an array declaration is applied in our semantics. The declaration of an array results in

- the array name x is bound the the first storage location of the array, and
- each storage location l_i in the array are bound to the applied value v.

We have the empty rule to end a list of array declarations.

3.4.6 Rule Declarations

The transition system for rule declarations is define as $(\Gamma_{DecRule}, \rightarrow_{D_R}, T_{DecRule})$, where $\Gamma_{DecRule} = (DecRule \times Env_P)$ and $T_{DecRule} = Env_P$.

So we have that a configuration is a procedure environment follow by more declarations, or just a procedure environment. The end configuration is when all rules have been declared, and thus we have only the updated procedure environment.

The transition relation is on the form $env_V \vdash \langle D_R, env_P\rangle \rightarrow_{D_R} env'_P$, and is defined by the semantic rules below.

[Dr-rule-without-return]

$$\frac{env_V \vdash \langle D_R, env_P[r \mapsto (S, P_F, env_V, D_V, D_A)]\rangle \rightarrow_{D_R} env'_P}{env_V \vdash \langle \textbf{rule}\; r(P_F)\;\{D_V, D_A S\}\; D_R, env_P\rangle \rightarrow_{D_R} env'_P}$$

[Dr-rule-with-return]

$$\frac{env_V \vdash \langle D_R, env_P[r \mapsto (S, P_F, env_V, D_V, D_A, type)]\rangle \rightarrow_{D_R} env'_P}{env_V \vdash \langle \textbf{rule}\; r(P_F) :type\;\{D_V, D_A, S\}\; D_R, env_P\rangle \rightarrow_{D_R} env'_P}$$

[Dr-rule-empty]

$$env_V, sto \vdash \langle \varepsilon, env_P\rangle \rightarrow_{D_R} env_P$$

Table 3.24: Rule declarations

Table 3.24 shows that each rule declaration will bind the rule name to its formal parameters, its declarations and its commands. For rules with a return type the return type is also stored. Since we have static variable bindings, we also store the variable environment as it looks at the time of the rule declaration. The empty rule declaration is defined to end a list of rule declarations.

When a rule is called (described in the transition system of S), we can fetch these stored values, and apply them to the computation as needed.

3.4.7 Turn and Ant Type Declarations

The transition system for DecTurn $((DecTurn \times Env_P) \cup Env_P, \rightarrow_{DT}, Env_P)$ and the transition system for DecAT $(DecAntType \times Env_P \cup Env_P, \rightarrow_{D_{AT}}, Env_P)$ resembles that of DecRule, and the definition of their transition relation is very similar.

Turn declarations are on the form $env_V \vdash \langle D_T, env_P \rangle \rightarrow_{D_T} env'_P$ and $\rightarrow_{DT}$ is defined in table 3.25. We see that the only difference from rule declarations is that turns can have no return type.

[Dt-turn]

$$\frac{env_V \vdash \langle D_T, env_P[t \mapsto (S, P_F, env_V, D_V, D_A)]\rangle \rightarrow_{D_T} env'_P}{env_V \vdash \langle \mathbf{turn}\, t(P_F)\, \{D_V D_A S\}\ D_T, env_P\rangle \rightarrow_{D_T} env'_P}$$

[Dt-turn-empty]

$$env_V \vdash \langle \varepsilon, env_P\rangle \rightarrow_{D_T} env_P$$

Table 3.25: Turn declaration

Ant type declarations are on the similar form $env_V \vdash \langle D_{TEAM}, env_P\rangle \rightarrow_{D_{AT}} (, env'_P)$, and $\rightarrow_{D_{AT}}$ is defined in table 3.26. Ant types can not take any parameters or return any value.

[Dat-anttype-declaration]

$$\frac{env_V \vdash \langle D_{AT}, env_P\ [at \longmapsto (S, env_V, D_V D_A)]\rangle \rightarrow_{D_{AT}} (env'_P)}{env_V \vdash \langle \mathbf{anttype}\ at\{D_V D_A S\} D_{AT}, env_P\rangle \rightarrow_{D_{AT}} (env'_P)}$$

Table 3.26: Ant type declaration

3.4.8 Common Memory Declarations

The transition system $(\Gamma_{DecMC}, \rightarrow_{D_{MC}}, T_{DecMC})$ is defined by the following configurations.

$$\Gamma_{DecMC} = (DecMC \times Env_V \times Store) \cup (Env_V \times Store)$$

So a configuration in this transition system can either be an updated variable environment and store, where we still have declarations to perform. Or all variables have been declared, and we have the updated variable environment and store.

We therefore have the end configurations defined as:

$$T_{DecMC} = Env_V \times Store$$

The transition relation for $\rightarrow_{D_{MC}}$ is on the form

$$env_P \vdash \langle D_{MC}, env_V, sto\rangle \rightarrow_{D_{MC}} (env'_V, sto')$$

and it is defined by the semantic rules of table 3.27.

[Dmc-common]

$$\frac{env_P \vdash \langle D_{MC}, env_V\,[x \mapsto (type, z)]\,[next \mapsto new(l)]\,, sto[l \mapsto v][COMMONDECLS \mapsto z+1]\rangle \rightarrow_{D_{MC}} (env'_V, sto')}{env_P \vdash \langle \textbf{common var}\ x\ :\ type\ =\ exp\,; D_{MC}, env_V, sto\rangle \rightarrow_{D_{MC}} (, env'_V, sto')}$$

where $env_P, env_V, sto \vdash exp \rightarrow v$ and $env_V\,(next) = l$ and $z = sto(COMMONDECLS)$

[Dmc-common-empty]

$$env_P \vdash \langle \epsilon, env_V, sto\rangle \rightarrow_{D_{MC}} (env_V, sto)$$

Table 3.27: Common memory

Common variables are very different from normal variables, both in the way they are declared and stored, and in the way they are references after declaration. As shown in figure 3.1 we have a storage location containing the number of common variable declarations made ($COMMONDECLS$). We increment this number by one every time a common variable is declared. Furthermore we store the value of the number, together with the type of variable, in the variable environment. This enables us to reference common variables by making the calculation $sto(COMMONBASE + z)$, where z is the index of the given variable.

We update the storage *sto* with the value of the variable, and moves the pointer *next* to the next free location.

3.4.9 Team Memory Declarations

The transition system $(\Gamma_{DecMT}, \rightarrow_{D_{MC}}, T_{DecMT})$ is defined by the configurations.

$$\Gamma_{DecMT} = (DecMT \times Env_V \times Store) \cup (Env_V \times Store)$$

and the end configurations

$$T_{DecMT} = Env_V \times Store$$

So a team memory declaration will result in an updated variable environment and an updated storage.

The transition relation for $\rightarrow_{D_{MT}}$ is on the form

$$env_P \vdash \langle D_{MT}, env_V, sto\rangle \rightarrow_{D_{MT}} (env'_V, sto')$$

and is defined by the semantic rules in table 3.28.

A team memory variable is declared without assigning a value to the variable. The semantic rule does however update the storage, since it increments the number of team variables declared by one (the number is found in the storage location $TEAMDECLS$). As with common variables, the variable name is bound the value of this number, and the variable type in the variable environment.

[Dmt-team]

$$\frac{env_P \vdash \langle D_{MT}, env_V\,[x \mapsto (type, z)]\,, sto[TEAMDECLS \mapsto z + 1]\rangle \rightarrow \left(env'_V, sto'\right)}{env_P \vdash \langle \mathbf{teambrain\ var}\ x\ :\ type\,; D_{MT}, env_V, sto\rangle \rightarrow \left(, env'_V, sto'\right)}$$

where and $z = sto(TEAMDECLS)$

[Dmt-team-empty]

$$env_P \vdash \langle \epsilon, env_V, sto\rangle \rightarrow_{D_{MC}} (env_V, sto)$$

Table 3.28: Team memory declaration

3.4.10 Private Memory Declarations

The transition system $(\Gamma_{DecMP}, \rightarrow_{D_{MP}}, T_{DecMP})$ is defined by the configurations $\Gamma_{DecMP} = (DecMP \times Env_V \times Store) \cup (Env_V$ and the end configurations $T_{DecMP} = Env_V \times Store$. So we have that a private variable declaration updates the variable environment and the storages.

The transition relation for $\rightarrow_{D_{MP}}$ is on the form

$$env_P \vdash \langle D_{MP}, env_V, sto\rangle \rightarrow_{D_{MP}} \left(env'_V, sto'\right)$$

and is defined by the semantic rules in table 3.29. The semantic rules are very similar to those of team variable declarations, and will stand uncommented.

[Dmp-private]

$$\frac{env_P \vdash \langle D_{MT}, env_V\,[x \mapsto (z, type)]\,, sto[PRIVATEDECLS \mapsto z + 1]\rangle \rightarrow \left(env'_V, sto'\right)}{env_P \vdash \langle \mathbf{private\ var}\ x\ :\ type\,; D_{MP}, env_V, sto\rangle \rightarrow \left(, env'_V, sto'\right)}$$

where $z = sto(PRIVATEDECLS)$

[Dmt-private-empty]

$$env_P \vdash \langle \epsilon, env_V, sto\rangle \rightarrow_{D_{MC}} (env_V, sto)$$

Table 3.29: Private memory declaration

3.4.11 Commands

Where declarations can alter both the environments and storage of AWL, commands can only alter the storage - e.g. assigning a new value to a variable. We therefore define the transition system

$$(Com \times Store \cup Store, \rightarrow_S, Store)$$

Transitions are on the form $env_P, env_V \vdash \langle S, sto\rangle \rightarrow sto'$, since we need to know about the bindings of procedures and variables to execute a command correctly.

The transition rules for $\rightarrow_S$ are defined in the semantic rules below.

[S-while-true]

$$\frac{env_P, env_V \vdash \langle S, sto\rangle \rightarrow_S sto'' \quad env_P, env_V \vdash \langle \mathbf{while}(be)\{S\}, sto''\rangle \rightarrow_S sto'}{env_P, env_V \vdash \langle \mathbf{while}(be)\{S\}, sto\rangle \rightarrow_S sto'}$$

where $env_P, env_V, sto \vdash be \rightarrow_{be} tt$

[S-while-false]

$$env_P, env_V \vdash \langle \mathbf{while}(be)\{S\}, sto\rangle \rightarrow_S sto$$

where $env_P, env_V, sto \vdash be \rightarrow_{be} ff$

Table 3.30: While command

Table 3.30 shows the semantics of a while command. If the condition of the command is true, then we execute the body, and apply the same while command on the updated storage. So the while command is defined recursively. If the condition is false, then there are no changes to the storage.

[S-if-true]

$$\frac{env_P, env_V \vdash \langle S_1, sto\rangle \rightarrow_S sto'}{env_P, env_V \vdash \langle \mathbf{if}(be)\{S_1\}\mathbf{else}\{S_2\}, sto\rangle \rightarrow_S sto'}$$

where $env_P, env_V, sto \vdash be \rightarrow_{be} tt$

[S-if-false]

$$\frac{env_P, env_V \vdash \langle S_2, sto\rangle \rightarrow_S sto'}{env_P, env_V \vdash \langle \mathbf{if}(be)\{S_1\}\mathbf{else}\{S_2\}, sto\rangle \rightarrow_S sto'}$$

where $env_P, env_V \vdash be \rightarrow_{be} ff$

Table 3.31: If-Else command

[S-assign]

$$env_P, env_V \vdash \langle x = exp, sto\rangle \rightarrow_S sto[l \mapsto v]$$

where $env_P, env_V, sto \vdash exp \rightarrow_{exp} v$ and $env_V\ (x) = l$

[S-comp]

$$\frac{env_P, env_V \vdash \langle S_1, sto\rangle \rightarrow_S sto'' \quad env_P, env_V \vdash \langle S_2, sto''\rangle \rightarrow_S sto'}{env_P, env_V \vdash \langle S_1\ S_2, sto\rangle \rightarrow_S sto'}$$

Table 3.32: Assign and Comp command

The semantic rules for the if-else command are defined in table 3.31. Again there are the two possibilities that the condition is either true or false. If the condition is true then the body of if is executed, otherwise the body of $else$ is executed. The semantic rule for an assign command shows that to update the value of a variable we first lookup the storage location, and then bind the new value to this location. The rule [S-comp] shows that to execute consecutive commands, we first execute the first command, and then execute the next command on the updated storage.

[S-assign-array]

$env_P, env_V \vdash \langle x[ae] = exp, sto\rangle \to_S sto[l_{loc} \mapsto v]$
where $env_P, env_V, sto \vdash ae \to_{ae} z_1$ and $env_V\,(x) = (type, l, z_2)$ and
$l_{loc} = l + z_1$ and $env_P, env_V, sto \vdash exp \to_{exp} v$ and
$z_1 \geq 0$ and $z < z_2$

Table 3.33: Array assign command

Table 3.33 shows how to assign a new value to an element in an array. We first evaluate the arithmetic expression to the index of the desired element, and lookup the base address of the array. We then add these two values together and lookup the location in the storage. To make sure that we do not access storage outside the array allocations, we check that the given index i between zero and the length of the array (minus one).

[S-common memory assign]

$env_P, env_V \vdash \langle \mathbf{cmem}\, x = exp, sto\rangle \to_S sto[l \mapsto v]$
where $(type, z) = env_V\,(x)$ and $l = COMMONBASE + z$

[S-team memory assign]

$env_P, env_V \vdash \langle \mathbf{tmem}\, x = exp, sto\rangle \to_S sto[l \mapsto v]$
where $(z, type) = env_V\,(x)$ and
$l = teamLoc\,(sto\,(CURRENTTEAM)) + TEAMALLOC + z$

[S-private memory assign]

$env_P, env_V \vdash \langle p\; x = exp, sto\rangle \to_S sto[l \mapsto v]$
where $(z_1, type) = env_V\,(x)$ and
$l = antLoc\,(sto\,(CURRENTTEAM)\,, sto\,(CURRENTANT)) + ANTALLOC + z$

Table 3.34: Memory assign commands

[S-rule-call]

$$\frac{\begin{array}{l}\langle P_F, env'_V\,[next \mapsto new\,(l)]\rangle \to env''_V\\ env_P \vdash \langle P_A, env''_V\,[next \mapsto new\,(l)]\,, sto\rangle \to \left(env^3_V, sto'\right)\\ env_P \vdash \left\langle D_V, env^3_V, sto'\right\rangle \to_{D_V} \left(env^4{}_V, sto''\right)\\ env_P \vdash \left\langle D_A, env^4_V, sto''\right\rangle \to_{D_A} \left(env^5_V, sto^3\right)\\ \quad env_P, env^5_V \vdash \left\langle S, sto^3\right\rangle \to_S sto^4\end{array}}{env_P, env_V \vdash \langle r(P_A);, sto\rangle \to_S sto^4}$$

where $l = env_V\,(next)$ and $env_P\,(r) = \left(S, P_F, env'_V, D_V, D_A\right)$

[S-turn-call]

$$\frac{\begin{array}{l}\langle P_F, env'_V\,[next \mapsto new\,(l)]\rangle \to env''_V\\ env_P \vdash \langle P_A, env''_V\,[next \mapsto new\,(l)]\,, sto\rangle \to \left(env^3_V, sto'\right)\\ env_P \vdash \left\langle D_V, env^3_V, sto'\right\rangle \to_{D_V} \left(env^4{}_V, sto''\right)\\ env_P \vdash \left\langle D_A, env^4_V, sto''\right\rangle \to_{D_A} \left(env^5_V, sto^3\right)\\ \qquad env_P, env^5_V \vdash \left\langle S, sto^3\right\rangle \to_S sto^4\end{array}}{env_P, env_V \vdash \langle \mathbf{endturn}\; t(P_A);, sto\rangle \to_S sto^4}$$

where $l = env_V\,(next)$ and $env_P\,(t) = \left(S, P_F, env'_V, D_V, D_A\right)$

Table 3.35: Rule and turn call commands

[S-setProperty]

$env_P, env_V \vdash \langle \mathbf{setProperty}(ae, exp);, sto\rangle \rightarrow_S sto\,[l \mapsto v]$

where $env_P, env_V, sto \vdash ae \rightarrow_{ae} z$ and $env_P, env_V, sto \vdash exp \rightarrow_{exp} v$ and $l = z$

Table 3.37: Set property command

[S-process]

$$\frac{\begin{array}{l} env_P, env_V \vdash \langle D_V, env'_V\,[next \mapsto new\,(l)]\,, sto\rangle \rightarrow_{D_V} (env''_V, sto') \\ env_P, env_V \vdash \langle D_A, env''_V, sto'\rangle \rightarrow_{D_V} (env'''_V, sto'') \\ env_P, env_V \vdash \langle S, sto''\,[l_1 \mapsto z_1]\,[l_2 \mapsto z_2]\rangle \rightarrow_S sto''' \end{array}}{env_P, env_V \vdash \langle \mathbf{process}(ae_1, ae_2, at);, sto\rangle \rightarrow_S sto'''}$$

where $\left(env'_V, D_V, D_A S\right) = env_P\,(at)$, and
$env_P, env_V, sto \vdash ae_1 \rightarrow_{ae} z_1$ and $env_P, env_V, sto \vdash ae_2 \rightarrow_{ae} z_2$
$l_1 = sto\,(CURRENTTEAM)$ and $l_2 = sto\,(CURRENTANT)$

Table 3.38: Process command

When assigning a value to a ant memory variable as done in table 3.34, we first lookup the variable's relative address. We then lookup the base address of the memory type (e.g. for common variables we have $COMMONBASE$). We can now update the storage at the base address added to the relative address with the new value. Calling a rule as a command is similar to calling a rule as an expression - except that there is no storage allocated for a return value.

[S-return]

$env_P, env_V \vdash \langle \mathbf{return}\, exp;, sto\rangle \rightarrow_S sto\,[l \mapsto v]$

where $l = env_V\,(return)$ and $env_P, env_V, sto \vdash exp \rightarrow v$

[S-skip]

$env_P, env_V \vdash \langle \mathbf{skip};, sto\rangle \rightarrow_S sto$

Table 3.36: Return and skip command

In table 3.36 we show the semantic rules for the return and the skip command. The skip command (obviously) does absolutely nothing. The return command evaluates its expression parameters, and stores the value in the location denoted by $env_V(return)$, which have been updated by the rule call that ultimately lead to this command. The *setProperty* command updates a given location directly in the storage. This is of course a powerful command, but also dangerous, since it makes it very easy for the programmer to make fatal mistakes.

The process command takes a team number, a ant number and an ant-type name. Basically this is just a procedure call to the given ant type, except that we update the storage locations $CURRENTTEAM$ and $CURRENTANT$ we the applied values. This achieves that the ant programmer do not need to worry about his team number, or which ant he is currently processing. The ant type can then be programmed as if there were just one ant.

3.4.12 Formal and Actual Parameters

The formal parameters of a procedure is basically variable declarations. We therefore define the transition system

$$(FParm \times Env_V \cup Env_V, \rightarrow_{P_F}, Env_V)$$

for formal parameter declarations, which shows that formal parameters only change the variable environment, since they do not assign a value to the parameter.

We also define the transition system

$$(AParm \times Env_V \times Store \cup Env_V \times Store, \rightarrow_{P_A}, Env_V \times Store)$$

for actual parameter assignments. Actual parameters primarily change the storage applying values to already declared formal parameters. We do however move the pointer *next* in the variable environment.

The transition relation for formal parameters $\rightarrow_{P_F}$ is on the form $\langle P_F, env_V \rangle \rightarrow_{P_F} env'_V$ and are defined by table 3.39. The semantic rules shows that each parameter will get its name bound to a storage location in the variable environment.

[Pf-formal parameters]

$$\frac{\langle P_F, env_V\,[x \mapsto l]\,[next \mapsto new\,(l)]\rangle \rightarrow env'_V}{\langle var\ x\ :\ type; P_F, env_V\rangle \rightarrow_S env'_V}$$

where $l = env_V\,(next)$

[Pf-formal parameters-empty]

$$\langle \epsilon, env_V \rangle \rightarrow_S env_V$$

Table 3.39: Formal parameters

The semantic rules defining the $\rightarrow_{P_A}$ are on the form $env_P \vdash \langle P_A, env_V, sto\rangle \rightarrow (env'_V, sto')$ and defined in table 3.40.

[Pa-actual parameters]

$$\frac{env_P \vdash \langle P_A, env_V\,[next \mapsto new\,(l)]\,, sto\,[l \mapsto v]\rangle \rightarrow (env'_V, sto')}{env_P \vdash \langle exp; P_A, env_V, sto\rangle \rightarrow_{P_A} (env'_V, sto')}$$

where $l = env_V\,(next)$ and $env_P, env_V, sto \vdash exp \rightarrow_{exp} v$

[Pa-actual parameters-empty]

$$env_P \vdash \langle \epsilon, env_V, sto\rangle \rightarrow_{P_A} (env_V, sto)$$

Table 3.40: Actual parameters

3.4.13 Team Declaration

When a team is declared in AWL, we allocate space for the following data:

- The primary team attributes - team number, next ant (currently active ant count) and the x , y coordinates of the team base.
- A copy of the declared teambrain memory variables.
- The number of ants (found in the location $MAXANTS$) and for each ant a copy of the declared private memory variables.

The primary attributes of a team can be assigned values at team declaration. The team number is just a incrementing number (stored at $TEAMCOUNT$), and the x , y coordinates are chosen randomly. Since there are no active ants on a newly declared team, the value of the next ant storage location should be zero.

So a team declaration will alter the variable environment and the storage. We define the transition system

$$(\Gamma_{DecTeam}, \rightarrow_{D_{TEAM}}, T_{DecTeam})$$

with the configurations

$$\Gamma_{DecTeam} = DecTeam \times Env_V \times Store \cup Env_V \times Store$$

and the end configurations

$$T_{DecTeam} = Env_V \times Store$$

Transitions will be on the form $env_P \vdash \langle D_{TEAM}, env_V, sto\rangle \rightarrow_{D_{TEAM}} (env'_V, sto')$ and the transition relation is defined in table 3.41. When a team is declared we lookup how many teams that have already been declared. We store this number as the team number at location l. The storage location denoted by l_{nt} is the next free storage location (where the next team can be stored). The calculation of l_{nt} takes the base of the team being declared, and adds the total amount of locations allocated to each team. l_{na} is the storage location storing the currently active ant count, which we set to zero. Finally we store the random constructed coordinates, and increment the number of declared teams by 1.

[Dteam-createTeam]

$$\frac{\begin{array}{l} env_P \vdash < D_{TEAM}, \\ env_V\,[x \mapsto (integer, l)]\,[next \mapsto l_{nt}], \\ sto\,[l \mapsto teamno]\,[l_{na} \mapsto 0]\,[l_x \mapsto x]\,[l_y \mapsto y]\,[TEAMCOUNT \mapsto (z+1)] \\ \rightarrow_{D_{TEAM}} (env'_V, sto') \end{array}}{env_P \vdash \langle \mathbf{createTeam}(x); D_{TEAM}, env_V, sto\rangle \rightarrow_{D_{TEAM}} (env'_V, sto')}$$

where
$l = env_V\,(next)$,
$l_{na} = new(l)$, $l_x = new(l,2)$, $l_y = new(l,3)$,
$teamno = sto\,(TEAMCOUNT)$,
$l_{nt} = new(l, TEAMSIZE)$,
$x = ran\,(sto\,(WORLDSIZE))$ and $y = ran\,(sto\,(WORLDSIZE))$

[Dteam-createTeam-empty]

$env_P \vdash \langle \varepsilon, env_V, sto\rangle \rightarrow (env_V, sto)$

Table 3.41: team declaration

3.4.14 World

The definition of a world is a complete AWL program. All other syntactical constructs are derivated from the world construct, and thus it is what ties an AWL program together. The result of a world construct is that storage has changed, since all declarations are out of scope, when we step out of the defined world.

The transition system defining World is $(\mathit{World} \times \mathit{Store}, \rightarrow_w, \mathit{Store})$. Transitions are on the form $\langle w, sto\rangle \rightarrow_w (sto')$, which illustrates the point that only storage changes. The transition relation $\rightarrow_w$ is defined in table 3.42.

[w-world]

$$\frac{\begin{array}{l} env_P \vdash \langle D_{CM}, env_V\,[next \mapsto l]\,, sto\,[WORLDSIZE \mapsto z_1]\,[MAXANTS \mapsto z_2]\,[MAXFOOD \mapsto z_3]\rangle \rightarrow_{D_{CM}} (env, sto^1) \\ env_P \vdash \langle D_{TM} env, sto^1\rangle \rightarrow_{D_{TM}} (env, sto^2) \quad env_P \vdash \langle D_{PM} env, sto^2\rangle \rightarrow_{D_{PM}} (env, sto^3) \\ env_P \vdash \langle D_{TEAM}, env_V^3, sto^3\rangle \rightarrow_{D_{TEAM}} (env_V^4, sto^4) \\ env_V^4 \vdash \langle D_R, env_P\rangle \rightarrow_{D_R} env_P^1 \qquad env_V^4 \vdash \langle D_T, env_P^1\rangle \rightarrow_{D_T} env_P^2 \\ env_V^4 \vdash \langle D_{AT}, env_P^2\rangle \rightarrow_{D_{AT}} (env_P^3) \\ env_P^3 \vdash \langle D_V, env_V^4, sto^4\rangle \rightarrow_{D_V} (env_V^5, sto^5) \qquad env_P^3 \vdash \langle D_A, env_V^5, sto^5\rangle \rightarrow_{D_A} (env_V^6, sto^6) \\ env_P^3, env_V^6 \vdash \langle S, sto^7\rangle \rightarrow_S sto^8 \end{array}}{\langle \mathbf{world}(n_1 n_2 n_3)\{D_{CM} D_{TM} D_{PM} D_R, D_T D_{AT} \mathbf{main}\{D_{TEAM} D_V D_A S\}\}, sto\rangle \rightarrow_w sto^8}$$

where

$env_p, env_V, sto \vdash n_1 \rightarrow_{ae} z_1$, $env_p, env_V, sto \vdash n_2 \rightarrow_{ae} z_2$, $env_p, env_V, sto \vdash n_3 \rightarrow_{ae} z_3$ and

$l = FOODBASE + sto(MAXFOOD) \cdot 2$

Table 3.42: The world declaration

The semantic rule [w-world] is large, but very straight-forward. A world definition is made up by the following elements

- The world parameters (size, ants and food)
- Common, teambrain and private ant memory declarations
- Rule declarations
- Ant type declarations
- Team declarations
- Main section

So to compute a world, we first store the world parameters in the predefined storage locations $WORLDSIZE$, $MAXANTS$ and $MAXFOOD$. Doing this we also move *next* to point at the next free location after the storage allocated to food. We then run through all declarations, and finally we execute the commands in the main section of the program.

3.5 Standard Environment

In our language we provide methods for getting a value directly from memory, and a method for setting a value directly to memory.

$$\textbf{getProperty}(ae) \quad \textbf{setProperty}(ae, v)$$

Using these methods requires the programmer to have a knowledge of how the data is stored, and that is of course a problem. However, it is also to great a task to define syntactical constructs and semantical rules for every desirable operation in AWL. This would simply result in a very large grammar, and some very complicated semantic rules. The solution to this is to introduce a standard environment, where high level functions could be implemented using the basic operations of AWL. An example of a function, that could very well be in a standard environment for AWL could be $getWorldSize()$, which would return the size of the world.

[getWorldSize]
procedure getWorldSize(){**return getproperty**(*0*); }

Another procedure that might be helpful when creating a ant world could be $walk(var\ d\ :\ direction)$, which would move an ant in a given direction, and perhaps also check to see if the moving ant could capture a base or kill an enemy ant.

It is clear that the usability of AWL would be greatly increased with a standard environment, however at its current state, there is no standard environment in AWL. To introduce one we should expand the procedure environment to contain normal procedure and functions, which should only be accessible from the main section and rules (ants should not have access).

An example of a another standard environment is the Java.lang package in JAVA, which implements many useful methods using the basic operations of JAVA.

3.6 Derivation Tree

In this section we will show how we can use the semantics described in the previous sections to describe an execution of a given program. We will give examples of how a derivation tree for a while command looks like, and one for an equals expression. The derivation trees shows the different stages a statement goes through before ending up in the end state.

While derivation tree.

$$\frac{(2) env_P, env_V \vdash \langle y = y - 1;, sto\rangle \to_S sto[l \mapsto (z_1)] \qquad (3) v_P, env_V \vdash \langle \textbf{while}(v > 0)\{y = y - 1;\}, sto'\rangle \to_S sto'}{(1) env_P, env_V \vdash \langle \textbf{while}(v > 0)\{y = y - 1;\}, sto\rangle \to_S sto'}$$

(1)=Where $env_P, env_V, sto \vdash (y > 0) \to tt$, $l = env_V\ (y)$, and
$env_P, env_V, sto \vdash y \to 1$
(2)=Where $env_P, env_V, sto \vdash (y - 1) \to z_1$
(3)=Where $env_P, env_V, sto \vdash (y > 0) \to ff$, and where
$env_P, env_V, sto \vdash y \to 0$

Here is a derivation tree for while. We see that in the first loop that $y > 0$ evaluates to true, and therefore we need to execute the commands found within the while body. The command to be executed is an assign command that subtracts 1 from the variable y. This results in y being equal to 0. with the updated storage we call while again. This time $y > 0$ will evaluate to false, and the while loop will terminate. If y had been equals e.g. 10, the tree would have been much more comprehensive and we would have had to loop through the while statement a lot more times.

Equals arithmetic expression derivation tree.

$$\frac{env_P, env_V, sto \vdash 9 \rightarrow_{ae} z_1 \quad \dfrac{env_P, env_V, sto \vdash 8 \rightarrow z_3 \quad env_P, env_V, sto \vdash 5 \rightarrow z_4}{env_P, env_V, sto \vdash 8 * 5 \rightarrow (z_3 \cdot z_4)}}{env_P, env_V, sto \vdash 9 == 8 * 5 \rightarrow_{be} ff}$$

Where $z_1 \neq (z_3 \cdot z_4)$

We here show the derivation tree for an equals expression. We first evaluate the leftmost argument to z_1, we then proceed to evaluate the rightmost argument. This argument is a composite arithmetic expression. Because of this we evaluate this first and get $(z_3 \cdot z_4)$. we now compare these and get *ff* because $z_1 \neq (z_3 \cdot z_4)$. Logically the more complex the expression is the higher the derivation tree will span. In this example we have a numeral and a multiplication expression as the logical parts of the boolean expression. The reason that these are the logical expressions is that this is so described in our grammar[3]

3.7 Summary

In this chapter we have defined the operational semantics of AWL. To do so, we have defined syntactical categories and specified an abstract grammar, on which we have based our transition systems. We have defined the environments and storage that AWL uses, and specified functions to aid us in describing the semantics. The semantics of all syntactical constructs have been defined in the transition relation of the transition systems.

Looking back at this chapter, a reasonable question would be if it wouldn't be a lot easier just to describe the semantics with words. Clearly it would make the semantics easier to understand when first reading them. However the semantics are constructed for implementation of the programming language, and it is very likely that semantics specified only with words would cause confusion and misunderstandings. By using a known notation and defining exactly how each syntactical construct changes the environment, we avoid confusion.

In the next chapter we will define another operational semantics for the abstract machine AWLAM.

[3]The BNF described in chapter 2

Chapter 4

AWLAM

In this chapter we will define the abstract machine AWLAM, which is shorthand notation for "Ant War Language Abstract Machine". We will define how the abstract machine is designed and works, and also which instructions it provides. The instructions will be described by an abstract syntax, and furthermore we will give an operational semantics for it. We will use the abbreviation AM and the composite word "abstract machine" interchangeably. When we refer to AM it will be the AWLAM unless we have specified otherwise. Also we will use the normal terms associated with stack operations, i.e. push and pop, when a value is added to the top of a stack, or when a value is removed from the top of a stack.

As such the abstract machine should be seen as an "abstract AM". It means that the AM defined in this chapter does not necessarily resemble how it actually will be implemented, but may be seen as an intermediate result. The reason for this is that we wish to bridge the semantic gap in a more gentle manner, and it makes the process of proving our result a lot nicer. The actual difference is not very big, and will be explained in the next section.

4.1 Definition of AWLAM

We will now give a definition of the AM. The AM is made up from a set of registers, a code store, a data store, a code stack, and an evaluation stack. The registers point to various areas of the code and data store during the execution of an AM program. The evaluation stack is used for direction, boolean and arithmetic calculations, the code stack for keeping track of which instructions have to be executed and the two stores will, as their names imply, be used for storage of program data and the program code itself. Figure 4.1 on page 51 shows how the memory and the registers are laid out.

There are nine registers which are: "Current team" (CT),"Current team memory" (CTM),"Current ant" (CA), "Current ant memory" (CAM), "Start team" (ST), "Start data" (SD), "Next" ($NEXT$), "Local scope" (LS), "Common memory" (CM) and "Program counter" (PC). The CT register refers to the start address of the current team, which means the team that is in scope at the moment, and the CA register refers to the start address of the current ant, on the current team. The CTM and CAM are used for referring to the ant-memory locations of the current team and the current ant. The effect is that you always know which ant is currently being executed, and which team it belongs to, and therefore you know what part of the memory should be in scope. The LS register refers to the value of local scope, and keeps track of what scope we are currently in, and the CM is used to keep track of the common memory. The PC register is used to refer to the current instruction on the code store. It is though, not used as one might expect. As such the actual execution of code is controlled with the code stack, and the program counter is used to keep track of what is currently on the code stack. The location of these registers at the start of the program is illustrated in figure 4.1

The code store will contain the program, which is going to be executed, expressed in abstract machine code. The data store contains all the data which is stored during the program's life cycle. When we refer

to locations in the stores, we will do so relatively from the start of the store. It means that the first line of a store will be addressed "0", and the line after "1". The same thing holds for the registers.

The evaluation stack is used for calculations during the execution of a program, and can contain integers, booleans and directions. The code stack will contain the code which is going to be executed next. At the start of a program it will be a copy of the code store, but as the program executes, instructions will be popped from the stack. If at some point in the code a jump is made to another part of the code, the content of the stack will be popped. The code at the place to where we jumped, and through to the end of the program, will be pushed onto the stack. Again instructions will be popped from the stack as they are executed. This will continue until the program terminates or another jump is reached.

As such, the code stack might not be the best solution to control execution of code, and may be seen as an abstraction. It would be more reasonable to implement a "program counter" register, and use it to point at the next instruction to be executed from the code store. The pros and cons for using a program counter is that it boosts execution speed and efficiency, but adds to the complexity of the machine. There is only a minor difference in the behavior between using a register solution and using the stack solution. As implied earlier we will use the register solution.

For the sake of simplicity we will assume that all variables and instructions use one memory location each in the memory of the AM. The theory and notation in this chapter is based on the book [1].

Figure 4.1 shows how the memory is organized, and where the registers are pointing at the start of a program. The layout is the same as that of AWL, the only difference being that in AWLAM we have registers.

4.1.1 Notation and Definitions

Before we go on, it might be a good idea to introduce some definitions, and to say a little about the notation used in the next section. Configurations of the AM are on the form:

$$\langle r,\ c,\ e,\ m\rangle \in \mathbf{Reg} \times \mathbf{Code} \times \mathbf{Stack} \times \mathbf{Memory}$$

r is the function mapping registers to numbers defined by:

$$r \in \mathbf{Reg} = \mathbf{Register} \hookrightarrow \mathbb{Z}$$

and the set of registers is:

$$g \in \mathbf{Register} = \{PC,\ CT,\ CTM,\ CA,\ CAM,\ ST,\ SD,\ NEXT,\ LS\}$$

code is the sequence of code to be executed and consists of

$$code \in \mathbf{Code}$$

a set of AM instructions, which is defined by the abstract syntax in table 4.1. k is the number of total instructions in the code. The code stack is defined by:

$$c \in \mathbf{cStack} = (code)$$

and e is the evaluation stack defined by:

[1] [2, chapter 3]

SD	0	world size
	1	max ant number
	2	number of foods
	3	number of declared teams
	4	current team
	5	current ant
	6	number of common decl.
	7	number of team decl.
	8	number of private decl.
	9	food - 0 -X
	10	food - 0 - Y
	11	food - 1 -X
	12	food - 1 - Y
	13	food - I -X
	14	food - I -Y
CM	15	common value - 0
	16	common value -1
	17	common value -i

Register	Name
SD	= Start data
CM	= Common memory
SD	= Start data
ST	= Start team
CTM	= Current team memory
CA	= Current ant
CAM	= Current ant memory

ST, CT	18	Team - 0 - no
	19	Team - 0 - nextant
	20	Team - 0 - Base X
	21	Team -0 - Base Y
CTM	22	Team -0- teamvar -0
	23	Team -0- teamvar -1
	24	Team -0- teamvar -j
CA	25	Ant - 0 - X Coord
	26	Ant - 0 - Y Coord
CAM	27	Ant - 0 - private value - 0
	28	Ant - 0 - private value -1
	29	Ant - 0 - private value - k
	30	Ant - 1 - X Coord
	31	Ant - 1- Y Coord
	32	Ant - 1 - private value - 0
	33	Ant - 1 - private value -1
	34	Ant - 1 - private value - k
	35	Ant - i - X Coord
	36	Ant - i - Y Coord
	37	Ant - i - private value - 0
	38	Ant - i - private value -1
	39	Ant - i - private value - k
	40	Team - 1 - no
	41	Team - 1 - nextant
	42	Team - 1 - Base X
	43	Team -1 - Base Y
	44	Team -1- teamvar -0
	45	Team -1- teamvar -1
	46	Team -1- teamvar -j

Figure 4.1: Memory of awlam.

$$e \in \mathbf{eStack} = (\mathbb{Z} \cup \mathbf{Bool} \cup \mathbf{Dir})^*$$

where

$$b \in \mathbf{Bool} = \{tt,\ ff\} \text{ and } \quad d \in \mathbf{Dir} = \{ll,\ rr,\ uu,\ dd,\ cc\}.$$

m is the memory formally defined by:

$$m \in \mathbf{Memory} = (\mathbb{Z} \cup \mathbf{Bool} \cup \mathbf{Dir})^*.$$

The transition relation for AWLAM is on the form:

$$\langle r,\ c,\ e,\ m\rangle \triangleright \langle r',\ c',\ e',\ m'\rangle$$

where the triangle specifies the transition itself, and means that it is done in one step. Finally $z \in \mathbb{Z}$.

When we write $\langle r, \mathbf{ADD} : \mathbf{TRUE} : c, z_1 : z_2 : e, m\rangle$, it means that the instructions on the code stack is the **ADD** command, and that the one coming right after is the **TRUE** command, i.e. when the **ADD** command has been executed, the next instruction on the stack will be **TRUE**. The c means that there might be more code on stack, but we will only specify the code that we need at this certain time.

When there is a colon between two elements it serves as a separator. In the example mentioned before two elements are on the stack, namely z_1 and z_2 which are used as operands for the **ADD** command.

Furthermore, we shall use the notation $r(g)$ to denote the value of register g. When we refer to it in connection with the evaluation stack, we will use the register name to mean the actual value of the register. E.g. when we write $next : e$ we actually mean $r(next) : e$.

With these definitions, we are ready to take a look at the operational semantics.

4.1.2 Instruction Set of AWLAM

The instruction set of AWLAM is seen in table 4.1. It is expressed in BNF and tells us that *code* can be either a single instruction, a sequence of instructions, or no instruction at all.

code	::=	ϵ | *inst* : *code*
inst	::=	**ADD** | **SUB** | **MULT** | **DIV** | **PUSH** n | **POP** | **TRUE** | **FALSE**
	|	**LEFT** | **RIGHT** | **UP** | **DOWN** | **CENTER** | **EQ** | **LE** | **NEG**
	|	**AND** | **OR** | **JUMP** n | **JUMPF** n | **LOADS** $[g]$ | **LOAD** n $[g]$
	|	**SAVES** $[g]$ | **SAVE** n $[g]$ | **LABEL n** | **NEXT**
	|	**CALL** n_1, n_2 | **CALLAT** n | **RETURN** | **SAVEREG** g
	|	**NOOP** | **SWAP** | **RAN** | **DUP**

Table 4.1: Abstract syntax for AWLAM

The instructions themselves will be explained in the next section, where we also explain the operational semantics for AWLAM.

4.2 Operational Semantics of AWLAM

In table 4.2 through 4.8 we show the operational semantics for AWLAM. We will start out by explaining the arithmetic rules in table 4.2.

[ADD-AM]
$\langle r, \textbf{ADD} : c, z_1 : z_2 : e, sto\rangle \triangleright \langle r, c, (z_1 + z_2) : e, sto\rangle$
[SUB-AM]
$\langle r, \textbf{SUB} : c, z_1 : z_2 : e, sto\rangle \triangleright \langle r, c, (z - z_2) : e, sto\rangle$
[MULT-AM]
$\langle r, \textbf{MULT} : c, z_1 : z_2 : e, sto\rangle \triangleright \langle r, c, (z_1 \cdot z_2) : e, sto\rangle$
[DIV-AM]
$\langle r, \textbf{DIV} : c, z_1 : z_2 : e, sto\rangle \triangleright \left\langle r, c, (\frac{Z_1}{Z_2}) : e, sto\right\rangle$

Table 4.2: Transition rules for arithmetic instructions.

The top of the code stack contains the **ADD** command, and two values z_1 and z_2 are found on the top of the evaluation stack. After the instruction has been executed the sum of z_1 and $z_2$will lie on top of the stack. The notation says that $(z_1 + z_2)$ is on the evaluation stack which should be read as the actual result of this operation. The rest of the arithmetic operations work in a similar way. In the table 4.3 we see the instructions for pushing and popping a value to the evaluation stack.

[PUSH n-AM]
$\langle r, \textbf{PUSH}\ n : c, e, sto\rangle \triangleright \langle r, c, \mathcal{N}[\![n]\!] : e, sto\rangle$
[POP-AM]
$\langle r, \textbf{POP} : c, v : e, sto\rangle \triangleright \langle r, c, e, sto\rangle$
[TRUE-AM]
$\langle r, \textbf{TRUE} : c, e, sto\rangle \triangleright \langle r, c, tt : e, sto\rangle$
[FALSE-AM]
$\langle r, \textbf{FALSE} : c, e, sto\rangle \triangleright \langle r, c, ff : e, sto\rangle$
[LEFT-AM]
$\langle r, \textbf{LEFT} : c, e, sto\rangle \triangleright \langle r, c, ll : e, sto\rangle$
[RIGHT-AM]
$\langle r, \textbf{RIGHT} : c, e, sto\rangle \triangleright \langle r, c, rr : e, sto\rangle$
[UP-AM]
$\langle r, \textbf{UP} : c, e, sto\rangle \triangleright \langle r, c, uu : e, sto\rangle$
[DOWN-AM]
$\langle r, \textbf{DOWN} : c, e, sto\rangle \triangleright \langle r, c, dd : e, sto\rangle$
[CENTER-AM]
$\langle r, \textbf{CENTER} : c, e, sto\rangle \triangleright \langle r, c, cc : e, sto\rangle$

Table 4.3: Instructions for pushing values onto the stack.

The first instruction **PUSH** n pushes a numeral n onto the stack, or actually it is the value of the numeral n denoted by $\mathcal{N}[\![n]\!]$ that is pushed. **POP** works in the opposite way, which means that it removes a value from the evaluation stack. The rest works in a way similar to **PUSH** n, except that they push the value of the instruction name, which means a *tt* for **TRUE**, *ff* for **FALSE**, *ll* for **LEFT**, etc. The table 4.4 shows instructions for boolean calculations.

[EQ1-AM]
$\langle r, \mathbf{EQ} : c, z_1 : z_2 : e, sto\rangle \triangleright \langle r, c, (z_1 = z_2) : e, sto\rangle$
[EQ2-AM]
$\langle r, \mathbf{EQ} : c, b_1 : b_2 : e, sto\rangle \triangleright \langle r, c, (b_1 = b_2) : e, sto\rangle$
[EQ3-AM]
$\langle r, \mathbf{EQ} : c, d_1 : d_2 : e, sto\rangle \triangleright \langle r, c, (d_1 = d_2) : e, sto\rangle$
[LE-AM]
$\langle r, \mathbf{LE} : c, z_1 : z_2 : e, sto\rangle \triangleright \langle r, c, (z_1 \leq z_2) : e, sto\rangle$
[NEG-AM]
$\langle r, \mathbf{NEG} : c, b : e, sto\rangle \triangleright \langle r, c, \neg b : e, sto\rangle$
[AND-AM]
$\langle r, \mathbf{AND} : c, b_1 : b_2 : e, sto\rangle \triangleright \langle r, c, (b_1 \wedge b_2) : e, sto\rangle$
[OR-AM]
$\langle r, \mathbf{OR} : c, b_1 : b_2 : e, sto\rangle \triangleright \langle r, c, (b_1 \vee b_2) : e, sto\rangle$

Table 4.4: Instructions for boolean operations.

The **EQ** instructions will pop two operands from the stack, and evaluate whether they are equal or not, and then push the boolean result onto the stack. The reason that there are three different rules is that we have three simple types, namely arithmetic, boolean and direction. The three are, except for their type, the alike. The **LE** instruction works in the same way as **EQ**, with the only difference that it is the "less than or equal" operation. The **NEG** (boolean not) gives the opposite value of a boolean value on the evaluation stack, pops the original value and pushes the new value.

AND and **OR** both works the same way. With two boolean values placed on the stack, they pop the two values, and use the boolean operator on them. They then push the result back onto the evaluation stack. The result depends on the values on the evaluation stack, and is calculated using normal boolean algebra rules. The part in table 4.5 is the instructions used for jumping to some specific part of the code. In the table k is the length of det total code.

[JUMP n-AM]	
$< r, \mathbf{JUMP}\ n : c, e, sto > \triangleright$	where $z = label\,(\mathcal{N}[\![n]\!])$
$< r[PC \mapsto z], code[z] : code[z+1], ..., code[k], e, sto >$	
[JUMPF1-AM]	
$< r, \mathbf{JUMPF}\ n : c, b : e, sto > \triangleright$	if $b = ff$ and
$< r[PC \mapsto z], code[z] : code[z+1], ..., code[k], e, sto >$	where $z = label\,(\mathcal{N}[\![n]\!])$
[JUMPF2-AM]	
$< r, \mathbf{JUMPF}\ n : c, b : e, sto > \triangleright < r, c, e, sto >$	if $b = tt$
[LABEL-AM]	
$< r, \mathbf{LABEL}\ n : c, e, sto > \triangleright < r, c, e, sto >$	

Table 4.5: Instructions for jumping in the code.

JUMP n is used to jump to a specific label in the code. What happens when a jump is made, is that all code on the code stack will be popped, and the code from the place to where we jumped, through to the end of the program, will be pushed onto the stack. **JUMPF** n does the same thing, except that it pops a boolean value from the stack first and evaluates it. If it evaluates to false, the same thing will happen as for the **JUMP** n, and if it evaluates to true the instruction just after the **JUMPF** n will be executed. The **LABEL** instruction is just to specify a label in the code, and if one is encountered, the instruction after the label will be executed next. The instructions in table 4.6 are used for accessing the registers.

[LOADS [g]-AM]	
$\langle r, \mathbf{LOADS}\,[g] : c, z_1 : e, sto\rangle \triangleright \langle r, c, z_2 : e, sto\rangle$	where $z_2 = m[r(g) + z_1]$
[SAVES [g]-AM]	
$\langle r, \mathbf{SAVES}\,[g] : c, z : v : e, sto\rangle \triangleright \langle r, c, e, sto\,[z \mapsto v]\rangle$	where $z = m[r(g) + z]$
[LOAD n [g]-AM]	
$\langle r, \mathbf{LOAD}\ n\,[g] : c, e, sto\rangle \triangleright \langle r, c, v : e, sto\rangle$	where $v = m[r\,(g) + \mathcal{N}[\![n]\!]]$
[SAVE n [g]-AM]	
$\langle r, \mathbf{SAVE}\ n\,[g] : c, v : e, sto\rangle \triangleright \langle r, c, e, sto\,[z \mapsto v]\rangle$	where $z = r\,(g) + \mathcal{N}[\![n]\!]$
[SAVEREG [g]-AM]	
$\langle r, \mathbf{SAVEREG}\,[g] : c, z : e, sto\rangle \triangleright \langle r\,[g \mapsto z]\,, c, e, sto\rangle$	
[NEXT-AM]	
$\langle r, \mathbf{NEXT} : c, e, sto\rangle \triangleright \langle r[NEXT \mapsto NEXT + 1], c, e, sto\rangle$	

Table 4.6: Instructions for loading and saving values to registers.

LOADS $[g]$ is used for loading a value from a memory location. The memory location is popped from the evaluation stack, and afterwards the retrieved value is pushed onto the evaluation stack. **SAVES** $[g]$ is used for saving the second value lying on the evaluation stack to the memory location lying on top of the stack. **LOAD** n $[g]$ is used for loading a memory location n which is relative to a register address g and then pushing it on top of the evaluation stack. So if $n = 3$, then it is the 3rd memory location after register g's memory location. **SAVE** n $[g]$ works in a similar way, only difference is that it pops a value

from the stack and saves it to a memory location n relative to the register g. The **NEXT** instruction is used for increasing the memory location, to which the register $NEXT$ points, by one. **SAVEREG** $[g]$ has the register g point at the location value that is on top of the stack

[CALL-AM]
$\langle r, \textbf{CALL}\ n_1, n_2 : c, v_1, \ldots, v_{n_2} : e, sto\rangle \triangleright$
$\langle r[LS \mapsto NEXT + 1], \textbf{JUMP}\ n_1 : c, v_1, \ldots, v_{n_2} : PC + 1 : NEXT : LS : e, sto\rangle$
where $n = label\,(\mathcal{N}[\![n_1]\!])$

[CALLAT-AM]
$\langle r, \textbf{CALLAT}\ n : c, e, sto\rangle \triangleright$
$\langle r[LS \mapsto NEXT][CT \mapsto z_3][CTM \mapsto z_4][CA \mapsto z_5][CAM \mapsto z_6],$
$\textbf{JUMP}\ n : c, PC + 1 : NEXT : LS : e, sto\rangle$
where
$z_1 = sto\,(4)$ (current team) , $z_2 = sto\,(5)$ (current ant)
$z_3 = ST + z_1 \cdot (4 + sto(7) + (2 + sto(8)) \cdot sto(1))$ (base address of current team)
$z_4 = z_3 + 4$ (base address of current team memory)
$z_5 = z_4 + sto(7) + (2 + sto(8)) \cdot z_2$ (base address of current ant)
$z_6 = z_5 + 2$ (base address of current ant private memory)

[RETURN-AM]
$\langle r, \textbf{RETURN} : c, z_1 : z_2 : z_3 : e, sto\rangle \triangleright$
$\langle r[PC \mapsto z_1][NEXT \mapsto z_2][LS \mapsto z_3], code[z_1] : code[z_1 + 1], ..., code[k], e, sto\rangle$

Table 4.7: Instructions calling code, and returning values.

CALL n_1, n_2 is used for calling a specific label in the code. It takes two parameters, n_1 and n_2. The first parameter is the label that you want to jump to, and the second one is the number of arguments you have on the stack. The reason you specify this in the call, is that it allows you to know the exact number of parameters on the stack. It works as follows: You have n_2, (which is $v_1$to v_n) arguments on the stack, and then use the **CALL** n_1, n_2 instruction. The result of this is that the LS register is set to point at the next free memory location, the PC register is set to point to $PC + 1$ and a **JUMP** n_1 is placed on the code stack. The PC register, the old values of register LS, and register $NEXT$ are pushed onto the evaluation stack so that after the subroutine call has finished, it is possible to return to the state that existed before we entered the subroutine call. The PC register is saved so we know what is the next instruction after we return from the subroutine. The LS register is saved so we will know what scope we were in before, and the $NEXT$ register is saved so we will know what was the next free memory location before the call. This has the effect that we actually will overwrite all local scope data from the subroutine after it has finished, which makes perfectly sense, since we do not want to save it. The return address is saved so we know which instruction is the next to be executed right after the subroutine returns, i.e what code to put on the code stack. One thing to notice is that the stack has been rearranged so that the arguments for the subroutine is now on top of the stack.

The **CALLAT** n instruction is a bit more complicated since it updates a lot more registers. It takes one parameter, which is the number of the anttype that should be called. It uses two values from storage, the current team number and the current ant number, which are used for updating registers. It updates $NEXT$, CT, CTM, CA and the CAM register, so that the right ant will be edited inside the ant type.

It pushes the same values on both the evaluation stack and the code stack as the **CALL**n_1, n_2instruction, so that it will know where to return to.

RETURN goes back to the state before a call was made. Is has the value of the return address (PC) together with the original $NEXT$ and LS registers on the evaluation stack, ($z_1, z_2$and z_3). After the return the $NEXT$ and LS registers will point at the values at which they did before. If there is any return value (v) it will be placed in the location 0 relative to register LS. Also the code stack will be popped for all code, and the code from the return address (PC), lying on the evaluation stack, and on, will be pushed on the code stack.

[NOOP-AM]
$\langle r, \textbf{NOOP} : c, e, sto\rangle \triangleright \langle r, c, e, sto\rangle$
[SWAP-AM]
$\langle r, \textbf{SWAP} : c, v_1 : v_2 : e, sto\rangle \triangleright \langle r, c, v_2 : v_1 : e, sto\rangle$
[RAN-AM]
$\langle r, \textbf{RAN} : c, z_1 : e, sto\rangle \triangleright \langle r, c, z_2 : e, sto\rangle$ where $0 \leq z_2 < z_1$
[DUP-AM]
$\langle r, \textbf{DUP} : c, v : e, sto\rangle \triangleright \langle r, c, v : v : e, sto\rangle$

Table 4.8: Various instructions

The **NOOP** command does nothing. It is an abbreviation for "No Operation". **SWAP** takes two values from the top of the stack, and swaps them so that the one on top will be switched with the one lying right after it. **RAN** returns a value between 0 and $z_1 - 1$, where z_1 lies on top of the evaluation stack, and pushes it onto the stack. The **DUP** instruction takes a value from the stack and pushes it back to the stack twice.

4.3 Program Example

Table 4.9 shows a small AWLAM program.

1.	**PUSH** *1*	5.	**EQ**	9.	**SUB**
2.	**LABEL** n_1	6.	**NEG**	10.	**JUMP** n_1
3.	**DUP**	7.	**JUMPF** n_2	11.	**LABEL** n_2
4.	**PUSH** *0*	8.	**PUSH** *1*	12.	**POP**

Where $n_1 = newLabel_{n_1}$ and $n_2 = newLabel_{n_2}$

Table 4.9: Example AWLAM instruction sequence.

In the AWLAM program in table 4.9, the value 1 is pushed onto the stack. This value is then duplicated so that we have 2 of the same value in the two top places of the stack. 0 is then pushed on top of the stack. We now proceeds to compare the two top values of the stack(0 and 1), the truth value of this is pushed on top of the stack. We then pop this value, negate it, and push the new truth value. After this if the value is *ff* the program jumps to the label n_2. Because the value is *tt* the program simply continues to the next instruction. Now 1 is pushed on top of the stack, and then pop the two topmost values off

the stack, subtracts the first value from the second value, and push this new value onto the stack. The program now encounters **JUMP** n_1. This means that the next instruction is the one directly following **LABEL** n_1 which is **DUP**. The program now goes through the above described phases until **JUMPF** n_2 is encountered. Because the value on top of the stack is *ff*, the program now jumps to the instruction directly following **LABEL** n_2. This instruction is **POP** which as the name implies pops the topmost value of the stack. After this there are no more instructions and the program is complete

To illustrate how the transitions of AWLAM progress, we will make a computation sequence of our small example. Initially all the program code is on the code stack, and the computation starts with an empty evaluation stack. Since it would take up to much space to write the entire code stack at each configuration, we just write $inst : c$ to illustrate, that after the topmost instruction the rest of the stack follows. Also each instruction is prefixed with it's line number to make the computation sequence more understandable.

	Registers	**Code stack**	**Evaluation stack**	**Storage**
	$\langle r,$	1. **PUSH** 2 : c,	ϵ,	$sto\rangle$
$\triangleright$	$\langle r,$	2. **LABEL** n_1 : c,	1,	$sto\rangle$
$\triangleright$	$\langle r,$	3. **DUP** : c,	1,	$sto\rangle$
$\triangleright$	$\langle r,$	4. **PUSH** 0 : c,	1 : 1,	$sto\rangle$
$\triangleright$	$\langle r,$	5. **EQ** : c,	0 : 1 : 1,	$sto\rangle$
$\triangleright$	$\langle r,$	6. **NEG** : c,	*ff* : 1,	$sto\rangle$
$\triangleright$	$\langle r,$	7. **JUMPF** n_2 : c,	*tt* : 1,	$sto\rangle$
$\triangleright$	$\langle r,$	8. **PUSH** 1 : c,	1,	$sto\rangle$
$\triangleright$	$\langle r,$	9. **SUB** : c,	1 : 1,	$sto\rangle$
$\triangleright$	$\langle r,$	10. **JUMP** $\mathbf{n_1}$: c,	0,	$sto\rangle$
$\triangleright$	$\langle r,$	3. **DUP** : c,	0,	$sto\rangle$
$\triangleright$	$\langle r,$	4. **PUSH** 0 : c,	0 : 0,	$sto\rangle$
$\triangleright$	$\langle r,$	5. **EQ** : c,	0 : 0 : 0,	$sto\rangle$
$\triangleright$	$\langle r,$	6. **NEG** : c,	*tt* : 0,	$sto\rangle$
$\triangleright$	$\langle r,$	7. **JUMPF** n_2 : c,	*ff* : 0,	$sto\rangle$
$\triangleright$	$\langle r,$	12. **POP** : c,	0,	$sto\rangle$
$\triangleright$	$\langle r,$	ϵ,	ϵ,	$sto\rangle$

Table 4.10: Computation sequence of the program in table 4.9

The computation in table 4.10 is an example of a terminating computation, because it is obviously not possible to make any transitions from the final configuration, since there are no more instructions. Furthermore the computation sequence ends in a terminal configuration, which means that the code component is empty.

If we appended the instruction *ADD* to the sequence, then the sequence would still terminate, but it would end in a stuck configuration, since *ADD* needs two numbers on the stack to make a transition. In contrast to a terminating computation is a computation which does not terminate. Such a computation is called a looping computation sequence. We will need these concepts when proving the correctness of the translation from AWL to AWLAM, so we will make a formal definition.

4.4 Summary

In this chapter we have given a definition of the AM. A central question when designing an AM is at what level should it be. Should it be as low level as possible, or should it be allowed to put in some abstract mechanisms, which will make it more high level. We have tried to keep the AM as low level as possible but have included a few mechanisms of abstraction. A central mechanism that we have added is relative addressing. We refer to memory locations relatively from registers which makes it a lot easier to work with the AM. Also we use a code stack to cycle through the code, and a program counter when executing

a jump. Using a code stack is not the most efficient way to design an AM, but it gives a model which is easier to illustrate, and the difference from making a code stack versus for example using a program counter register only, is not really that big. The main issue here is that the AM defined at first may not be the final AM. As such it is possible to master the complexity in a number of steps, whatever seems reasonable, and the AM defined here should be seen as an intermediate result, and not as a representation of how the AM will actually be implemented.

Another central concept is how the instruction set is laid out. How many instructions should there be. Should there be any composite instructions or helping functions to ease the job of translating. We do have some mechanism helping us, but have tried to limit these.

Since all evaluation is done on a stack, it may be seen as a stack machine. But we also do use registers to refer memory locations in an easier way, and to keep track of certain values and scope.

With the definition of AWL and the definition of the abstract machine, we are now ready to see how the translation of AWL code into AM instructions is done. This is the topic of the next chapter.

Chapter 5

Code Generation

In this chapter we will define translation functions which translates AWL to AWLAM. The translation functions are divided into categories, which are explained as they appear. To create a proper translation, it has been necessary to make protocols describing how e.g. a procedure is called. It should also be noted that the translation has not been optimized for performance in any way.

5.1 Protocols

In this section we will in general terms describe what happens when a procedure call and a call to an ant type is encountered.

When a procedure call is encountered the following happens.

Procedure Call

1. Place the value of $[NEXT]$on top of the stack
2. Place the value of$[LS]$ on top of the stack.
3. Place the return address on top of the stack
4. The value of $[LS]$ is set to the value of $[NEXT]$.
5. Place the arguments value on the stack.
6. Give the control to the procedure.

Inside the Procedure

1. Save the arguments on top of the stack.
2. perform any declarations and commands found in the body of the procedure.
3. Place the return value if any in the base of $[LS]$ $(0[LS])$
4. Pop the return address from the stack and jump to this.

After Exiting the Procedure

1. Pop the topmost value of the stack, and set $[LS]$ to this value.

2. Pop the topmost value of the stack, and set $[NEXT]$ to this value.
3. If there is a return value, this is found at the base of $[NEXT]$ $(0[NEXT])$

Before entering a procedure the old state of the program is saved along with where we want the the return value to be, if any. The procedure then enters the procedure. After declaring the variables found, and executing the commands that is inside the procedure body, the return address is saved in the base of $[LS]$, and the return value is popped from the stack, and then jumps to this location. When the program has left the procedure, $[NEXT]$ and $[LS]$ is restored to its old values.

When a call to an ant type is encountered the following happens

Call an Ant Type

1. Update CT to point to the address space associated with the current team.
2. Update CA to point to the address space associated with the current ant.
3. Place the value of $[NEXT]$on top of the stack
4. Place the value of$[LS]$ on top of the stack.
5. Place the return address on the stack
6. Jump to the ant type code

Inside Ant Type

1. Save the return address from the top of the stack.
2. Execute the declarations and commands found within the ant type body. Team mem is loaded relative to CT, and ant mem is loaded relatively to CA.
3. Pop the return address from the stack and jump to this.

After Exiting the Ant Type

1. Pop the topmost value of the stack, and set $[LS]$ to this value.
2. Pop the topmost value of the stack, and set $[NEXT]$ to this value.

Before entering an ant type, the old state of the program is saved The ant type call then enters the ant type instructions. After executing the instructions, the return value is saved in the base of $[LS]$, and the return address is popped from the stack, and then jumps to this location. When the program has left the procedure, $[NEXT]$ and $[LS]$ is restored to its old values.

5.2 Functions

Since we do not have variables in *AWLAM* we need a method to remember which storage location a given variable is bound to. We therefore define the function *mloc*.

$$mloc : \mathbf{ProcName} \times \mathbf{Var} \hookrightarrow Z$$

where the set **ProcName** is defined as **ProcName** = **RuleName** ∪ **TurnName** ∪ **AntTypeName**. We will use the meta variable p to reference elements in **ProcName**.

We do not have procedures either, so we need a way to remember which label prepends the translation of a given procedure. For that purpose we define *ploc*. Note that labels are in fact just natural numbers.

$$ploc : \mathbf{ProcName} \hookrightarrow Z$$

Finally we define the function *tloc* which maps team variable names to the first storage location allocated to the given team.

$$tloc : \mathbf{Var} \hookrightarrow \mathbf{Loc}$$

Updating the Functions

Instead of updating the functions with the correct values during the actual translation, we will specify how they can be updated before this process. Doing that will allow us to assume that they are defined correctly during the translation. Figure 5.1 shows the general idea on how to update *mloc* and *ploc*.

During translation we run through the AWL source program. This run-through will be done once before the actual translation only to map variables, procedures and teams to certain values. We can map each variable to a relative storage location inside a procedure, and we can map each procedure to the number it was declared as.

When we encounter a team declaration, it is a simple task of calculating the first storage location that will be allocated to that team. Similar calculations are described in the operational semantics of AWL. We can then update *tloc* with the correct location.

5.3 Code Generation

In this section we will define the translation functions.

5.3.1 Arithmetic Expressions

For the arithmetic expressions we have the total function:

$$CA : \mathbf{AExpr} \rightarrow (\mathbf{ProcName} \hookrightarrow \mathbf{Code})$$

which states that given a arithmetic expression and a procedure name, we will get translated code. We need the procedure name, so we can determine the relative storage locations of variables.

In the code generation for the arithmetic operations, we have swapped the arguments, so that they will be evaluated in the correct order. We use *mloc* to evaluate a variable x as the contents of the storage location $mloc(p, x)$ relatively to the local scope base LS inside the given procedure p . Evaluating a variable means pushing it onto the evaluation stack.

When calling a procedure we first evaluate the actual parameters, using the code translation function $\mathcal{CP}_{\mathcal{A}}[\![P_A]\!]$ defined in section 5.3.12. We then insert a **CALL** instruction, which jumps to the label mapped in *ploc* for the given procedure name. According to the defined protocols, we can now fetch the return value from storage location 0 relatively to the register **NEXT**.

The instructions **cmem**, **tmem** and **pmem** loads the contents of their mapped storage location relatively to their base registers **CM**, **CTM** and **CAM** onto the stack.

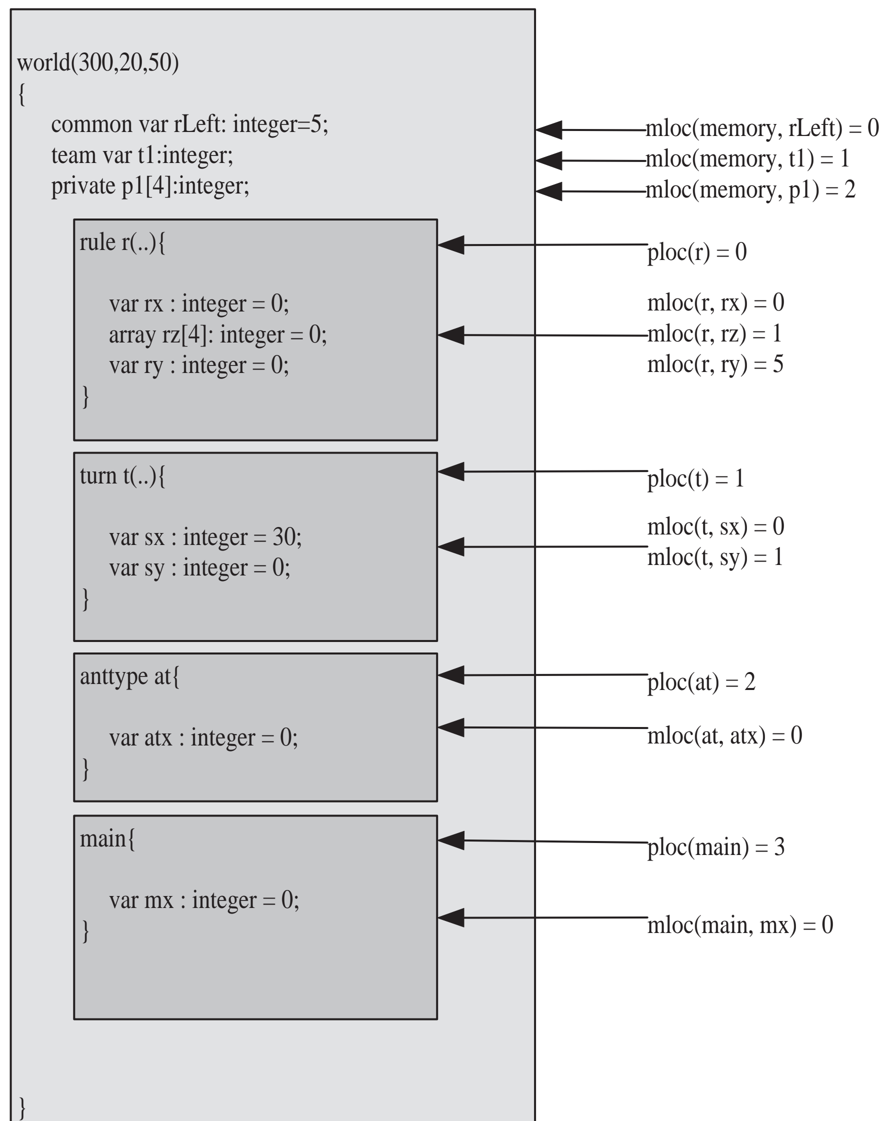


Figure 5.1: Definition of mloc, ploc and tloc

$\mathcal{CA}[\![n]\!]\,p$	=	**PUSH** n	
$\mathcal{CA}[\![x]\!]\,p$	=	**LOAD** n $[LS]$	where $n = mloc\,(p, x)$
$\mathcal{CA}[\![x\,[ae]]\!]\,p$	=	$\mathcal{CA}[\![ae]\!]$: **PUSH** n : **ADD** : **LOADS** $[LS]$	where $n = mloc\,(p, x)$
$\mathcal{CA}[\![ae_1{+}ae_2]\!]\,p$	=	$\mathcal{CA}[\![ae_2]\!]$: $\mathcal{CA}[\![ae_1]\!]$: **ADD**	
$\mathcal{CA}[\![ae_1{-}ae_2]\!]\,p$	=	$\mathcal{CA}[\![ae_2]\!]$: $\mathcal{CA}[\![ae_1]\!]$: **SUB**	
$\mathcal{CA}[\![ae_1{*}ae_2]\!]\,p$	=	$\mathcal{CA}[\![ae_2]\!]$: $\mathcal{CA}[\![ae_1]\!]$: **MULT**	
$\mathcal{CA}[\![ae_1/ae_2]\!]\,p$	=	$\mathcal{CA}[\![ae_2]\!]$: $\mathcal{CA}[\![ae_1]\!]$: **DIV**	
$\mathcal{CA}[\![(ae)]\!]\,p$	=	$\mathcal{CA}[\![ae]\!]$	
$\mathcal{CA}[\![r\,(P_A)]\!]\,p$	=	$\mathcal{CP}_{\mathcal{A}}[\![P_A]\!]$: **CALL** n_1, n_2 : **LOAD** 0 $[NEXT]$	where $n_1 = ploc(p)$ and $n_2 = parameter\ count$
$\mathcal{CA}[\![$**random**$(ae)]\!]\,p$		$\mathcal{CA}[\![ae]\!]$: **RAN**	
$\mathcal{CA}[\![$**cmem** $x;]\!]\,p$	=	**LOAD** n $[CM]$	where $n = mloc\,(memory, x)$
$\mathcal{CA}[\![$**tmem** $x;]\!]\,p$	=	**LOAD** n $[CTM]$	where $n = mloc\,(memory, x)$
$\mathcal{CA}[\![$**pmem** $x;]\!]\,p$	=	**LOAD** n $[CAM]$	where $n = mloc\,(memory, x)$
$\mathcal{CA}[\![$**getProperty**$(ae);]\!]\,p$	=	$\mathcal{CA}[\![ae]\!]$: **LOADS** $[SD]$	

Table 5.1: Translation of **AExp**

5.3.2 Boolean Expressions

For boolean expressions we have the total function:

$$\mathcal{CB} : \mathbf{BExpr} \rightarrow (\mathbf{ProcName} \hookrightarrow \mathbf{Code})$$

The translation of boolean expressions are very similar to the translation of arithmetic expression. Since we only have instructions for the relational operations "less than or equals" (**LE**) and "equals" (**EQ**), we use a combination of these and **NEG** to implement the other relational operations. Table 5.2 illustrates the translation of boolean expression.

$\mathcal{CB}\,[\![true]\!]\,p$	$=$	$\mathbf{TRUE}$	
$\mathcal{CB}\,[\![false]\!]\,p$	$=$	$\mathbf{FALSE}$	
$\mathcal{CB}\,[\![x]\!]\,p$	$=$	$\mathbf{LOAD}\,n\,[NEXT]$	where $n = mloc\,(p, x)$
$\mathcal{CB}\,[\![x\,[ae]\,;]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae]\!] : \mathbf{PUSH}\,n : \mathbf{ADD} : \mathbf{LOADS}\ [LS]$	where $n = mloc\,(p, x)$
$\mathcal{CB}\,[\![ae_1 == ae_2]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{EQ}$	
$\mathcal{CB}\,[\![be_1 == be_2]\!]\,p$	$=$	$CB\,[\![be_2]\!] : CB\,[\![be_1]\!] : \mathbf{EQ}$	
$\mathcal{CB}\,[\![de_1 == de_2]\!]\,p$	$=$	$\mathcal{CD}\,[\![de_2]\!] : \mathcal{CD}\,[\![de_1]\!] : \mathbf{EQ}$	
$\mathcal{CB}\,[\![ae_1 != ae_2]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{EQ} : \mathbf{NEG}$	
$\mathcal{CB}\,[\![be_1 != be_2]\!]\,p$	$=$	$\mathcal{CB}\,[\![be_2]\!] : \mathcal{CB}\,[\![be_1]\!] : \mathbf{EQ} : \mathbf{NEG}$	
$\mathcal{CB}\,[\![de_1 != de_2]\!]\,p$	$=$	$\mathcal{CD}\,[\![de_2]\!] : \mathcal{CD}\,[\![de_1]\!] : \mathbf{EQ} : \mathbf{NEG}$	
$\mathcal{CB}\,[\![ae_1 > ae_2]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{LE} : \mathbf{NEG}$	
$\mathcal{CB}\,[\![ae_1 < ae_2]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{EQ} : \mathbf{NEG} :$	
		$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{LE} : \mathbf{AND}$	
$\mathcal{CB}\,[\![ae_1 >= ae_2]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{EQ}$	
		$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{LE} : \mathbf{NEG} : \mathbf{AND}$	
$\mathcal{CB}\,[\![ae_1 <= ae_2]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{LE}$	
$\mathcal{CB}\,[\![be_1\,\mathbf{and}\,be_2]\!]\,p$	$=$	$\mathcal{CB}\,[\![be_2]\!] : \mathcal{CB}\,[\![be_1]\!] : \mathbf{AND}$	
$\mathcal{CB}\,[\![be_1\,\mathbf{or}\,be_2]\!]\,p$	$=$	$\mathcal{CB}\,[\![be_2]\!] : \mathcal{CB}\,[\![be_1]\!] : \mathbf{OR}$	
$\mathcal{CB}\,[\![(be)]\!]\,p$	$=$	$\mathcal{CB}\,[\![be]\!]$	
$\mathcal{CB}\,[\![!be]\!]\,p$	$=$	$\mathcal{CB}\,[\![be]\!] : \mathbf{NEG}$	
$\mathcal{CB}\,[\![r\,(P_A)]\!]\,p$	$=$	$\mathcal{CP}_{\mathcal{A}}\,[\![P_A]\!] : \mathbf{CALL}\,n_1, n_2 : \mathbf{LOAD}\,0\,[NEXT]$	where $n_1 = ploc(p)$, and $n_2 = parameter\ count$
$\mathcal{CB}\,[\![\mathbf{cmem}\,x]\!]\,p$	$=$	$\mathbf{LOAD}\,n\ [CM]$	where $n = mloc\,(memory, x)$
$\mathcal{CB}\,[\![\mathbf{tmem}\,x]\!]\,p$	$=$	$\mathbf{LOAD}\,n\ [CTM]$	where $n = mloc\,(memory, x)$
$\mathcal{CB}\,[\![\mathbf{pmem}\,x]\!]\,p$	$=$	$\mathbf{LOAD}\,n\ [CAM]$	where $n = mloc\,(memory, x)$
$\mathcal{CB}\,[\![\mathbf{getProperty}(ae)]\!]\,p$	$=$	$\mathcal{CA}\,[\![ae]\!] : \mathbf{LOADS}\ [SD]$	

Table 5.2: Translation of **BExp**

5.3.3 Direction Expressions

For the direction expressions we have the total function:

$$\mathcal{CD} : \mathbf{DExpr} \rightarrow (\mathbf{ProcName} \hookrightarrow \mathbf{Code})$$

The translation of direction expressions is similar to the translation of arithmetic and boolean expressions, and will stand uncommented. The translation is shown in table 5.3.

$\mathcal{CD_V}[\![var\ x : type = exp; D_V]\!]\,p$	=	$\mathcal{CE}[\![exp]\!]$: **SAVE** $0\,[NEXT]$: **NEXT** : $\mathcal{CD_V}[\![D_V]\!]$
$\mathcal{CD_V}[\![\epsilon]\!]\,p$	=	**NOOP**

Table 5.4: Translation of **DecVar**

$\mathcal{CD}[\![center]\!]\,p$	=	**CENTER**	
$\mathcal{CD}[\![up]\!]\,p$	=	**UP**	
$\mathcal{CD}[\![down]\!]\,p$	=	**DOWN**	
$\mathcal{CD}[\![right]\!]\,p$	=	**RIGHT**	
$\mathcal{CD}[\![left]\!]\,p$	=	**LEFT**	
$\mathcal{CD}[\![x]\!]\,,p$	=	**LOAD** $n\,[LS]$	where $n = mloc\,(p,x)$
$\mathcal{CD}[\![(de)]\!]\,p$	=	$\mathcal{CD}[\![de]\!]$	
$\mathcal{CD}[\![r\,(P_A)]\!]\,,p$	=	$\mathcal{CP_A}[\![P_A]\!]$: **CALL** n_1, n_2 : **LOAD** $0\,[NEXT]$	where $n_1 = ploc(p)$, and $n_2 = parameter\ count$
$\mathcal{CD}[\![$**cmem** $x;]\!]\,p$	=	**LOAD** $n\ [CM]$	where $n = mloc\,(memory, x)$
$\mathcal{CD}[\![$**tmem** $x;]\!]\,p$	=	**LOAD** $n\ [CTM]$	where $n = mloc\,(memory, x)$
$\mathcal{CD}[\![$**pmem** $x;]\!]\,p$	=	**LOAD** $n\ [CAM]$	where $n = mloc\,(memory, x)$
$\mathcal{CD}[\![x\,[ae]\,;]\!]\,,p$	=	$\mathcal{CA}[\![ae]\!]$: **PUSH** n : **ADD** : **LOADS** $[LS]$	where $n = mloc\,(p,x)$
$\mathcal{CD}[\![$**getProperty**$(ae);]\!]\,p$	=	$\mathcal{CA}[\![ae]\!]$: **LOADS** $[SD]$	

Table 5.3: Translation of **DExp**

5.3.4 Variable Declarations

To translate variable declarations we define two function

$$\mathcal{CD_V} : \textbf{DecVar} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

As with the semantic rules of a variable declaration, the code translation is recursive. Since we already have the mapping of each variables relative storage location, we only need to save the value of the variable in the next free storage location, and then update the $NEXT$ register to point at the next free location. Table 5.4 defines the function $\mathcal{CD_V}$.

5.3.5 Array Declarations

The translation function for array declarations is specified as

$$\mathcal{CD_A} : \textbf{DecArr} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

and defined in table 5.5 . The translated instructions create a looping instruction sequence, which walks through the storage locations of the array, and assigns the given value. $\mathcal{CD_A}$ is defined recursively.

5.3.6 Ant Type Declarations

We translate ant type declarations using the function

$\mathcal{CD_A}\,[\![array\ x\,[n] : type = exp; D_A]\!]\,p$	=	**PUSH** n : **LABEL** n_1 : **DUP** : **PUSH** 0 : **EQ** : **NEG** : **JUMPF** n_2 : $\mathcal{CE}\,[\![exp]\!]$: **SAVE** 0 $[NEXT]$: **NEXT** : **PUSH** 1 : **SWAP** : **SUB** : **JUMP** $\mathbf{n_1}$: **LABEL** n_2 : **POP** : $\mathcal{CD_A}\,[\![D_A]\!]$ where $n_1 = newlabel_1$ and $n_2 = newlabel_2$
$\mathcal{CD_A}\,[\![\epsilon]\!]\,p$	=	**NOOP**

Table 5.5: Translation of **DecArr**

$\mathcal{CD_{AT}}\,[\![\textbf{anttype}\ at\ \{D_V\,D_A\,S\}D_{AT}]\!]\,p$	=	**JUMP** n_2 : **LABEL** n_1 : $\mathcal{CD_V}\,[\![D_V]\!]$: $\mathcal{CD_A}\,[\![D_A]\!]$: $\mathcal{CS}\,[\![S]\!]$: **RETURN** : **LABEL** n_2 where $n_1 = newlabel_1$, $n_2 = newlabel_2$and $ploc(p) = n_1$
$\mathcal{CD_{AT}}\,[\![\epsilon]\!]\,p$	=	**NOOP**

Table 5.6: Translation of **DecAT**

$$\mathcal{CD_{AT}} : \textbf{DecAT} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

defined in table 5.6. We see that the first instruction in the translated sequence is a **JUMP** instruction, which jumps to the end of the sequence. This ensures that the declarations and commands inside the ant type is not computed during the declaration of the ant type. $\mathcal{CD_{AT}}$ uses the code translation functions for variable and array declarations to declare local data, and the code translation function for commands (defined in section 5.3.14) to execute its code. As defined in the protocols in section 5.1 the **RETURN** instruction return control to wherever the ant type was called from.

5.3.7 Rule Declarations

The translation of rule declarations is expressed by the function

$$\mathcal{CD_R} : \textbf{DecRule} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

defined in table 5.7. The translation is very similar to the translation of ant type declarations, however since rules can take parameters, we also need to evaluate those. For this purpose we use the translation functions $\mathcal{CP_F}$ and $\mathcal{CP_A}$ defined in section 5.3.12. We see that the translation of a rule with or without a return type results in the same sequence of AWLAM instructions.

5.3.8 Turn Declarations

We define the translation function

$$\mathcal{CD_T} : \textbf{DecTurn} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

to translate turn declarations. The function is further defined in table 5.8, and is identical to the translation of rule declarations.

$\mathcal{CD_R}$ $[\![\textbf{rule}\, r\,(P_F) : type\{D_V\, D_A S\} D_R]\!]\, p$	$=$	$\textbf{JUMP}\, n_2 : \textbf{LABEL}\, n_1 : \mathcal{CP_F}\,[\![P_F]\!] : \mathcal{CD_V}\,[\![D_V]\!] :$ $\mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \textbf{RETURN} : \textbf{LABEL}\, n_2$
		where $n_1 = newlabel_1$, $n_2 = newlabel_2$ and $ploc(p) = n_1$
$\mathcal{CD_R}$ $[\![\textbf{rule}\, r\,(P_F)\ \{D_V\, D_A S\} D_R]\!]\, p$	$=$	$\textbf{JUMP}\, n_2 : \textbf{LABEL}\, n_1 : \mathcal{CP_F}\,[\![P_F]\!] : \mathcal{CD_V}\,[\![D_V]\!] :$ $\mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \textbf{RETURN} : \textbf{LABEL}\, n_2$
		where $n_1 = newlabel_1$, $n_2 = newlabel_2$and $ploc(p) = n_1$
$\mathcal{CD_R}\,[\![\epsilon]\!]\, p$	$=$	**NOOP**

Table 5.7: Translation of **DecRule**

$\mathcal{CD_T}\,[\![\textbf{turn}\, t\,(P_F)\,\{D_V\, D_A S\} D_T]\!]\, p$	$=$	$\textbf{JUMP}\, n_2 : \textbf{LABEL}\, n_1 : \mathcal{CP_F}\,[\![P_F]\!] : \mathcal{CD_V}\,[\![D_V]\!] :$ $\mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \textbf{RETURN} : \textbf{LABEL}\, n_2$ where $n_1 = newlabel_1$, $n_2 = newlabel_2$and $ploc(p) = n_1$
$\mathcal{CD_T}\,[\![\epsilon]\!]\, p$	$=$	**NOOP**

Table 5.8: Translation of **DecTurn**

5.3.9 Team Declarations

When a team is declared, we need to update certain storage locations (see the definition of the transitions system for **TeamDec** for details). We also need to calculate the next free storage location, which means jumping over all locations allocated to a team. We can make this calculation at translation time (as opposed to doing it at runtime), since all the numbers needed in the calculation is known - e.g. the maximum ant count is programmed as an integer literal, so we can read the number directly. Had it been an expression the calculation could not have been done at translation time, since we would not know which number the expression would evaluate to.

To translate the team declarations we have the function

$$\mathcal{CD_{TEAM}} : \textbf{DecTeam} \to (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

defined by table 5.9.

5.3.10 Common Memory Declarations

Common memory variables are declared almost identical to normal variables. The only difference is that besides making the declaration we also update the storage location containing the number of common declarations made. We have the function

$$\mathcal{CD_{MC}} : \textbf{DecMC} \to (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

which is defined in table 5.10

$CD_{TEAM}\,[\![\textbf{createTeam}(x); D_{TEAM}]\!]\,p$	$=$	$\textbf{LOAD}\ \mathit{3}\ [SD] : \textbf{PUSH}\ ts : \textbf{SAVES}\ [SD] :$ $\textbf{LOAD}\ \mathit{0}\ [SD] : \textbf{RAN} : \textbf{PUSH}\ \mathit{2} : \textbf{PUSH}\ ts : \textbf{ADD} : \textbf{SAVE}\ [SD] :$ $\textbf{LOAD}\ \mathit{0}\ [SD] : \textbf{RAN} : \textbf{PUSH}\ \mathit{3} : \textbf{PUSH}\ ts : \textbf{ADD} : \textbf{SAVE}\ [SD] :$ $\textbf{LOAD}\ \mathit{3}\ [SD] : \textbf{PUSH}\ \mathit{1} : \textbf{ADD} : \textbf{SAVE}\ \mathit{3}\ [SD] :$ $\textbf{PUSH}\ n : \textbf{SAVEREG}\ [NEXT] : CD_{TEAM}\,[\![D_{TEAM}]\!]$ where $ts = tloc(x)$ and $n = ts + 4 + \mathit{teambrain\ declaration\ count} +$ $(2 + \mathit{private\ declaration\ count}) \cdot \mathit{ant\ count}$
$CD_{TEAM}\,[\![\epsilon]\!]\,p$	$=$	**NOOP**

Table 5.9: Translation of **DecTeam**

$\mathcal{CD}_{\mathcal{MC}}\,[\![\textbf{common var}\ x : type = exp; CD_{MC}]\!]\,p$	$=$	$\mathcal{CE}\,[\![exp]\!] : \textbf{SAVE}\ 0\ [NEXT] : NEXT$ $\textbf{LOAD}\ 6\ [SD] : \textbf{PUSH}\ 1 : \textbf{ADD} : \textbf{SAVE}\ 6\ [SD] : \mathcal{CD}_{\mathcal{MC}}\,[\![D_{MC}]\!]$ where $n = mloc\,(memory, x)$
$\mathcal{CD}_{\mathcal{MC}}\,[\![\epsilon]\!]\,p$	$=$	**NOOP**

Table 5.10: Translation of **DecMC**

5.3.11 Teambrain and Private Memory Declarations

When declaring a teambrain or a private memory variable there are no assignment included. Since we have already determined the relative storage location of the variables, we only need to update the locations dedicated to the number of teambrain and private memory allocations made.

We have the functions

$$\mathcal{CD}_{\mathcal{MT}} : \textbf{DecMT} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

and

$$\mathcal{CD}_{\mathcal{MP}} : \textbf{DecMP} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

defined in the tables 5.11 and 5.12.

$\mathcal{CD}_{\mathcal{MT}}\,[\![\textbf{teambrain var}\ x : type; CD_{MT}]\!]\,p$	$=$	$\textbf{LOAD}\ 7\ [SD] : \textbf{PUSH 1} : \textbf{ADD} : \textbf{SAVE}\ 7\ [SD] : \mathcal{CD}_{\mathcal{MT}}\,[\![CD_{MT}]\!]$
$\mathcal{CD}_{\mathcal{MT}}\,[\![\epsilon]\!]\,p$	$=$	**NOOP**

Table 5.11: Translation of **DecMT**

$\mathcal{CD}_{\mathcal{MP}}$ [[**private var** $x : type; CD_{MP}$]] p	$=$	**LOAD** 8 $[SD]$: **PUSH** 1 : **ADD** : **SAVE** 8 $[SD]$: $\mathcal{CD}_{\mathcal{MP}}$ [[CD_{MP}]] p
$\mathcal{CD}_{\mathcal{MP}}$ [[ϵ]] p	$=$	ϵ

Table 5.12: Translation of **DecMP**

5.3.12 Formal and Actual Parameters

Together formal and actual parameters declarations perform the function of a variable declaration. The actual parameters are placed on the stack, and the translation of the formal parameters stores the actual parameters in the allocated storage locations.

We have the functions

$$\mathcal{CP}_{\mathcal{F}} : \textbf{FParm} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

and

$$\mathcal{CP}_{\mathcal{A}} : \textbf{AParm} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

defined in table 5.3.12.

$\mathcal{CP}_{\mathcal{F}}$ [[**var** $x : type; P_F$]] p	$=$	**SAVE** 0 $[NEXT]$: **NEXT** : $\mathcal{CP}_{\mathcal{F}}$ [[P_F]]
$\mathcal{CP}_{\mathcal{F}}$ [[ϵ]] p	$=$	**NOOP**
$\mathcal{CP}_{\mathcal{A}}$ [[$ae; P_A$]] p	$=$	$\mathcal{CP}_{\mathcal{A}}$ [[P_A]] : $\mathcal{CA}$ [[ae]]
$\mathcal{CP}_{\mathcal{A}}$ [[ϵ]] p	$=$	**NOOP**

Table 5.13: Translation for formal and actual parameters

5.3.13 World

The translation of the world construct is actually the translation of the entire program. All other translation functions are called from $\mathcal{CW}$ expressed as

$$\mathcal{CW} : \textbf{World} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

and defined in table 5.14. We see that we make a calculation at translation time. We need to update the register $NEXT$ to point at the address following the food allocations. We can make this calculation now, because we know both the number of dedicated storage locations and the maximum pieces of food allowed.

$\mathcal{CW}\,[\![\textbf{world}\,(n_1, n_2, n_3)\,\{D_{MC} D_{MT} D_{MP}$ $D_R D_T D_{AT}\,\textbf{main}\,\{D_{TEAM} D_V D_A S\}\}]\!]\,p$	$=$	$\textbf{PUSH}\; n_1 : \textbf{SAVE}\; 0\,[SD] : \textbf{PUSH}\; n_2 : \textbf{SAVE}\; 1\,[SD] :$ $\textbf{PUSH}\; n_3 : \textbf{SAVE}\; 2\,[SD] : \textbf{PUSH}\; n_{next} : \textbf{SAVEREG}\,[NEXT]$ $\mathcal{CD}_{\mathcal{MC}}\,[\![D_{MC}]\!] : \mathcal{CD}_{\mathcal{MT}}\,[\![D_{MT}]\!] : \mathcal{CD}_{\mathcal{MP}}\,[\![D_{MP}]\!] : \mathcal{CD}_{\mathcal{TEAM}}\,[\![D_{TEAM}]\!] :$ $\mathcal{CD}_{\mathcal{R}}\,[\![D_R]\!] : \mathcal{CD}_{\mathcal{T}}\,[\![D_T]\!] : \mathcal{CD}_{\mathcal{AT}}\,[\![D_{AT}]\!] : \textbf{LABEL}\; 0 :$ $\mathcal{CD}_{\mathcal{V}}\,[\![D_V]\!] : \mathcal{CD}_{\mathcal{A}}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!]$ where $n_{next} = 9 + food\ count \cdot 2$

Table 5.14: Translation for world declaration

5.3.14 Commands

To translate commands we have the function

$$\mathcal{CW} : \textbf{World} \rightarrow (\textbf{ProcName} \hookrightarrow \textbf{Code})$$

defined in table 5.16.

The translation of the AWL commands are for the most part straight forward. There are however a couple of constructs, which need explaining. The translation of **process** needs to update the storage location dedicated to the current team and current ant. To do so it evaluates its parameters to the stack, and then saves the values and calls the given ant type.

The **return** command is translated to instructions, which saves the parameters at location 0 relatively to the register LS - as specified in our protocols. The **skip** command translates to a **NOOP** (no operation) instruction, since it changes nothing.

$\mathcal{CS}\llbracket r\,(P_A)\rrbracket\,p$	$=$	$\mathcal{CP_A}\llbracket P_A\rrbracket : \mathbf{CALL}\,n_1, n_2$ where $n_1 = ploc(r)$ and $n_2 = parameter\ count$
$\mathcal{CS}\llbracket x = exp;\rrbracket\,p$	$=$	$\mathcal{CE}\llbracket exp\rrbracket : \mathbf{SAVE}\,n\ [LS]$ where $n = mloc(p, x)$
$\mathcal{CS}\llbracket x\,[ae] = exp;\rrbracket\,p$	$=$	$\mathcal{CE}\llbracket exp\rrbracket : \mathcal{CA}\llbracket ae\rrbracket : \mathbf{PUSH}\,n : \mathbf{ADD} : \mathbf{SAVES}\ [LS]$ where $n = mloc\,(p, x)$
$\mathcal{CS}\llbracket \mathbf{endturn}\,t\,(P_A)\rrbracket\,p$	$=$	$\mathcal{CP_A}\llbracket P_A\rrbracket : \mathbf{CALL}\,n_1, n_2$ where $n_1 = ploc(t)$ and $n_2 = parameter\ count$
$\mathcal{C}S\llbracket \mathbf{process}\,(ae_1, ae_2, at)\rrbracket\,p$	$=$	$\mathcal{CA}\llbracket ae_1\rrbracket : \mathbf{SAVE}\,4\ [CT] : \mathcal{CA}\llbracket ae_2\rrbracket :$ $\mathbf{SAVE}\,5\ [CA] :: \mathbf{CALLAT}\,n :$ where $n = ploc(at)$
$\mathcal{CS}\llbracket \mathbf{while}(be)\{S\}\rrbracket\,p$	$=$	$\mathbf{LABEL}\,n_1 : \mathcal{CB}\llbracket be\rrbracket : \mathbf{JUMPF}\,n_2 :$ $\mathcal{CS}\llbracket S\rrbracket : \mathbf{JUMP}\,n_1 : \mathbf{LABEL}\,n_2$ where $n_1 = newlabel_1$and $n_2 = newlabel_2$
$\mathcal{CS}\llbracket \mathbf{if}(be)\{S_1\}\mathbf{else}\{S_2\}\rrbracket\,p$	$=$	$\mathcal{CB}\,[be] : \mathbf{JUMPF}\,n_1 : \mathcal{CS}\llbracket S_1\rrbracket :$ $\mathbf{JUMP}\,n_2 : \mathbf{LABEL}\,n_1 : \mathcal{CS}\llbracket S_2\rrbracket : \mathbf{LABEL}\,n_2$ where $n_1 = newlabel_1$ and $n_2 = newlabel_2$
$\mathcal{CS}\llbracket cmem\ x = exp;\rrbracket\,p$	$=$	$\mathcal{CE}\llbracket exp\rrbracket : \mathbf{SAVE}\,n\ [CM]$ Where $n = mloc\,(memory, x)$
$\mathcal{CS}\llbracket tmem\ x = exp;\rrbracket\,p$	$=$	$\mathcal{CE}\llbracket exp\rrbracket : \mathbf{SAVE}\,n\ [CTM]$ Where $n = mloc\,(memory, x)$
$\mathcal{CS}\llbracket pmem\ x = exp;\rrbracket\,p$	$=$	$\mathcal{CE}\llbracket exp\rrbracket : \mathbf{SAVE}\,n\ [CAM]$ Where $n = mloc\,(memory, x)$
$\mathcal{CS}\llbracket \mathbf{setProperty}(ae, exp);\rrbracket\,p$	$=$	$\mathcal{CE}\llbracket exp\rrbracket : \mathcal{CA}\llbracket ae\rrbracket : \mathbf{SAVES}\ [SD]$
$\mathcal{CS}\llbracket S_1 S_2\rrbracket\,p$	$=$	$\mathcal{CS}\llbracket S_1\rrbracket : \mathcal{CS}\llbracket S_2\rrbracket$
$\mathcal{CS}\llbracket \mathbf{return}\,exp;\rrbracket\,p$	$=$	$\mathcal{CE}\llbracket exp\rrbracket : \mathbf{SAVE}\,0\ [LS]$
$\mathcal{CS}\llbracket \mathbf{skip};\rrbracket\,p$	$=$	$\mathbf{NOOP} :$

Table 5.16: Translation of commands

5.4 Summary

We have now defined how to translate an AWL program to a sequence of AWLAM instructions through a list of translation functions. We have made such functions for each syntactical construct of AWL. The question that now remains is how to prove that the translation is in fact correct. Fortunately that is the topic of the next chapter.

Chapter 6

Provable Correct Implementation

In the last chapter we defined the abstract machine AWLAM, and constructed code generating functions, which translated AWL commands into a sequence of AWLAM instructions. In this chapter we will show that the translation is in fact correct, and we will define what correct means in this context. This chapter will only show a hand-full of proofs - the remaining can be found in Appendix A.

To avoid confusion we will refer to AWLAM as AM.

6.1 Correctness

We define the translation of an AWL program into AM code to be correct if (and only if) the execution of the AM code on the abstract machine will give the same result as specified by the operational semantics for AWL.

Since AWL and AM has the same type of storage, yielding the same result means ending up with identical storage states.

In the following sections we will prove that the translation functions from the previous chapters are correct. We will divide the proofs into

- proving the correct implementation of declarations,
- proving the correct implementation of expressions and
- proving the correct implementation of commands.

However we first need to describe the techniques, which we will use to make the proof.

6.2 Proof Techniques

We will conduct proofs by the two different proof techniques:

- induction on the shape of derivation trees, and
- induction on the length of computation sequences.

Below follow a short description of each of the two techniques.

Induction on the shape of derivation trees [1]

Proofs by induction on the shape of derivation trees are conducted on the following manner.

[1] [2, p. 28]

- We prove that the property, which we are trying to prove, holds for all simple derivation trees by showing that it holds for the axioms of the transition system.
- We then prove that the property holds for all composite derivation trees. This is done by assuming for each semantic rule that the property holds for the premises of the rule, and then proving that it also holds for the conclusion provided that the side conditions of the rule are satisfied.

Induction on the length of computation sequences [2]

Proofs by induction on the length of computation sequences are conducted in the following manner.

- We prove that the property holds for all computation sequences of length 0
- We then prove that the property holds for all other derivation sequences by first assuming that it holds for all sequences at length most k, and then showing that it then also holds for sequences of of length $k + 1$.

6.3 Meaning of Commands

For AWL we define the meaning of commands S as a partial function from **Store** to **Store**. [3]

$$S_{AWL} : \mathbf{Com} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

which means that for each command S, we have a partial function $S_{AWL}[\![S]\!] \in \mathbf{Store} \hookrightarrow \mathbf{Store}$. This function is defined as

$$S_{AWL}[\![S]\!]\,sto = \begin{cases} sto' & \text{if } env_V, env_P \vdash \langle S, sto\rangle \to sto' \\ undefined & \text{otherwise} \end{cases}$$

We also define the meaning of a sequence of instructions on AM as a partial function from **Store** to **Store**.

$$\mathcal{M} : \mathbf{Code} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

and more specific

$$\mathcal{M}[\![c]\!]\,sto = \begin{cases} sto' & \text{if } \langle r, c, \epsilon, sto\rangle \rhd^* \langle r', \epsilon, e, sto'\rangle \\ undefined & \text{otherwise} \end{cases}$$

So using these functions, we can determine how AWL commands or AM instructions will change the storage.

Using the function $\mathcal{M}$ we can now also specify the meaning of a command S by translating it into AM instructions and then executing the instructions on the abstract machine. We define the function $S_{AM}[\![S]\!] : \mathbf{Com} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$ by

$$S_{AM}[\![S]\!] = (\mathcal{M} \circ \mathcal{CS})[\![S]\!] = \mathcal{M}(\mathcal{CS}[\![S]\!])$$

[2] [2, p. 37]
[3] [2, p. 31]

6.4 Notation

Since we do not want to make the proofs of correctness to hard to read, we will not write e.g. $env_p, env_V, sto \vdash ae \rightarrow_{ae} z$ when we need to imply that a arithmetic expression evaluates to the number z in the semantics of AWL. Instead we will just assume that the reader understands that this is the case and use z. We will of course only do this when there can be no doubt. If we do need to write the complete transition rule, we will for the most part omit the first part and just write e.g. $ae \rightarrow_{ae} z$.

We will make the following shortcuts:

z where $\mathcal{N}[\![n]\!] = z$ and where $env_p, env_V, sto \vdash ae \rightarrow_{ae} z$

b where $env_p, env_V, sto \vdash be \rightarrow_{be} b$

d where $env_p, env_V, sto \vdash de \rightarrow_{de} d$

v where $env_p, env_V, sto \vdash exp \rightarrow_{exp} v$

This means that if we encounter e.g. the command **PUSH** n we can write $\langle r, \textbf{PUSH}\ n, \epsilon, sto\rangle \triangleright \langle r, \epsilon, z, sto\rangle$ without any further explanation.

6.5 Variable Declarations

Before proving the correctness of variable declarations, we must define which properties that must hold. The intuitive correctness is that the storage states are identical in the two semantics after declaring a variable. We would also like the register **NEXT** to point to the same storage location as the pointer *next* in the semantics of AWL. We define the following lemma to express this.

Lemma 6.5.1 *For all variable declarations we have that*

$$if\ \langle D_V, env_V, sto\rangle \rightarrow (env'_V, sto')\ then\ \langle r, \mathcal{CD_V}[\![D_V]\!]\,p, \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, \epsilon, sto'\rangle$$

where $env'_V(next) = r'(NEXT)$

So each variable must be stored at the same storage location in the two semantics. Also the pointer next must point to the same location as the register NEXT after declaration.

Proof: We will make the proof by induction on the shape of the derivation tree.

The case: [Dv-variable-declaration-empty]

We assume that $\langle \varepsilon, env_V, sto\rangle \rightarrow_{D_V} (env_V, sto)$. Using the translation function we get that $\mathcal{CD_V}[\![\epsilon]\!]\,p =$ **NOOP**. Using the semantics of **NOOP** we get that

$$\langle r, \textbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r, \epsilon, \epsilon, sto\rangle$$

which completes the proof of this case.

The case: [Dv-variable-declaration]

We assume that $\langle \textbf{var}\ x{:}type{=}exp; D_V, env_V, sto\rangle \rightarrow (env'_V, sto')$ holds because

$$\langle D_V, env_V\,[x \mapsto (type, l)]\,[next \mapsto new\,(l)]\,, sto\,[l \mapsto v]\rangle \rightarrow_{D_V} (env'_V, sto')\text{(which is the premise)}$$

because $l = env_V\,(next)$.

Using the code translation function we get that

$$\mathcal{CD_V}\,[\![\mathbf{var}\ x : type{=}exp;D_V]\!]\,p = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ 0[NEXT] : \mathbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p$$

We can now make the following computation sequence.

$$\langle r, \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ 0[NEXT] : \mathbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto\rangle \rhd^*$$
$$\langle r', \mathbf{SAVE}\ 0[NEXT] : \mathbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p, v, sto\rangle \rhd$$
$$\langle r'', \mathbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto''\rangle \rhd$$
$$\langle r''', \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto''\rangle$$

We see that $sto'' = [l \mapsto v]$, so it follows that the variables are stored at the correct locations.

Applying the induction hypothesis to the premise we get that

$$\langle r'', \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto''\rangle \rhd^* \langle r', \epsilon, \epsilon, sto'\rangle$$

which completes the computation which ends in the required state. It follows from the computation sequence that $env'_V(next) = r'(NEXT)$. This completes the proof of lemma A.1.1.

6.6 Array Declarations

We have allready defined a lemma expressing the correctness of variable declarations. The correctness of array declarations are naturally almost identical. We therefore define the following lemma[4].

Lemma 6.6.1 *For all array declarations we have that*

$$\textit{if}\ \langle D_A, env_V, sto\rangle \rightarrow (env'_V, sto')\ \textit{then}\ \langle r, \mathcal{CD_A}\,[\![D_A]\!]\,p, \epsilon, sto\rangle \rhd \langle r', \epsilon, \epsilon, sto'\rangle$$

where $r(NEXT) = env_V(next)$

So each array must be stored at the same storage locations in the two semantics. Also the pointer next must point to the same location as the register NEXT after declaration.

Proof: We will use induction on the shape of the derivation tree to prove lemma A.1.2.

The case: [Da-declaration-empty]

We assume that $\langle \epsilon, env_V, sto\rangle \rightarrow_{D_A} (env_V, sto)$. Using the translation function we get that $\mathcal{CD_A}\,[\![\epsilon]\!]\,p = \mathbf{NOOP}$, and with the semantics of **NOOP** we have that

$$\langle r, \mathbf{NOOP}, \epsilon, sto\rangle \rhd \langle r, \epsilon, \epsilon, sto\rangle$$

[4] [2, p. 73]

which completes the proof of this case.

The case: [Da-declaration]

We assume that

$$\langle \textbf{array}\ x[n]\ {:}type{=}exp;\ D_A, env_V, sto\rangle \rightarrow_{D_A} (env'_V, sto')$$

because

$$\langle D_A, env_V[x \mapsto (type, l, z)][next \mapsto new\ (l, z))], sto[l_i \mapsto v]\rangle \rightarrow_{D_A} (env'_V, sto')$$

where $i \in [0..z-1]$ and $l = env_V\ (next)$ and $z > 0$.

Using the code translation function we get that

$$\mathcal{CDA}\,[\![array\ x[n] : type = exp\ D_A]\!]\,p =$$

$$PUSH\ n_3 : LABEL\ n_1 : DUP : PUSH\ 0 : EQ : NEG : JUMPF\ n_2 : \mathcal{CE}\,[\![exp]\!] : SAVE\ 0[NEXT] :$$

$$NEXT : PUSH\ 1 : SWAP : SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CDA}\,[\![D_A]\!]\,p$$

We can now make the following computation sequence:

$$\left\langle r, \begin{array}{l} PUSH\ n_3 : LABEL\ n_1 : DUP : PUSH\ 0 : EQ : NEG : JUMPF\ n_2 : \mathcal{CE}\,[\![exp]\!] : SAVE\ 0[NEXT] \\ : NEXT : PUSH\ 1 : SWAP : SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD_A}\,[\![D_A]\!]\,p \end{array}, \epsilon, sto \right\rangle \triangleright^7$$

$$\left\langle r, \begin{array}{l} \mathcal{CE}\,[\![exp]\!] : SAVE\ 0[NEXT] : NEXT : PUSH\ 1 : SWAP : \\ SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD_A}\,[\![D_A]\!]\,p \end{array}, e, sto \right\rangle \triangleright^*$$

$$\langle r', \ PUSH\ 1 : SWAP : SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD_A}\,[\![D_A]\!]\,p\ \ , e, sto'\rangle \triangleright^*$$

$$\langle r', \ JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD_A}\,[\![D_A]\!]\,p\ \ , e', sto'\rangle \triangleright^*$$

$$\langle r'', \ \mathcal{CD_A}\,[\![D_A]\!]\,p\ \ , \epsilon, sto''\rangle \triangleright^*$$

$$\langle r', \epsilon, \epsilon, sto'\rangle$$

We get the first part of the computation using the semantics of AM, and we see that $sto'' = sto[l_i \mapsto v]$ where $i \in [0..\mathcal{N}[\![n]\!] - 1]$ as required. We get the last part by applying the induction hypothesis to the premise. This completes the proof of lemma A.1.2.

6.7 Arithmetic Expressions

Since the proofs of the three expression types in AWL are practically identical, we will only show the proof of arithmetic expressions - or at least some of it. The proofs of boolean and direction expressions can be found in Appendix A, as can the remaining part of the proof of arithmetic expressions..

The intuitive correctness of an arithmetic expression is that it evaluates to the correct number. Since we are using an evaluation stack in AM this means that the correct value of the expression must be pushed onto the stack. We will define the following lemma to express this.

Lemma 6.7.1 *For all arithmetic expression*

s ae we have that

$$\langle r, \mathcal{CA}[\![ae]\!], \epsilon, sto\rangle \triangleright \langle r, \epsilon, z, sto\rangle$$

where $env_P, env_V, sto \vdash ae \rightarrow_{ae} z$*.*

Furthermore, all intermediate configurations of this computation sequence will have a non-empty evaluating stack.

Proof: The proof of lemma A.3.1 is done by structual induction on ae.

The case: [ae-lit]

Using the code generation function $\mathcal{CA}$, we have that $\mathcal{CA}[\![n]\!]\, p = \textbf{PUSH}\ n$. From the semantics of AM we have that

$$\langle r, \textbf{PUSH}\ n, \epsilon, sto\rangle \triangleright \langle r', \epsilon, z, sto\rangle$$

and since $n \rightarrow z$ in the operational semantics for AWL, we have completed the proof for [ae-lit].

The case: [ae-var]

We have that $\mathcal{CA}[\![x]\!]\, p = \textbf{LOAD}\ n\,[LS]$, where LS is the register, which points to the local base address of the current routine p, and where $n = mloc\,(p, x)$ (the relative address of x inside p).

Using the semantics of AM we have that

$$\langle r, \textbf{LOAD}\ n\,[LS], \epsilon, sto\rangle \triangleright \langle r', \epsilon, sto\,(r\,(LS) + z)\,, sto\rangle$$

In the operational semantics of AWL we have that $x \rightarrow sto\,(env_V\,(x))$. Using the definition of LS and $mloc$ we see that $r(LS) + z = env_V(x)$, which completes the proof of this case.

The case: [ae-getProperty]

Using the code translation function we have

$$\mathcal{CA}[\![\textbf{getProperty}(ae);]\!] = \mathcal{CA}[\![ae]\!] : \textbf{LOADS}\ [SD]$$

and we there have the computation sequence

$$\langle r, \mathcal{CA}[\![ae]\!] : \textbf{LOADS}\ [SD]\,, \epsilon, sto\rangle \triangleright^*$$

$$\langle r', \textbf{LOADS}\ [SD]\,, z_1, sto\rangle \triangleright$$

$$\langle r'', \epsilon, z_2, sto\rangle$$

To make the first computation we apply the induction hypothesis to ae, and to make the second we use the semantics of **LOADS**. We see that $z_2 = sto(z_1)$, and using the rule [ae-getProperty] we see that this is the required result.

The case: [ae-mult]

We have that

$$\mathcal{CA}[\![ae_1 \cdot ae_2]\!] = \mathcal{CA}[\![ae_2]\!] : \mathcal{CA}[\![ae_1]\!] : \textbf{MULT}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}\,[\![ae_1]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, z_1, sto\rangle \qquad \text{and}$$

$$\langle r', \mathcal{CA}\,[\![ae_2]\!], \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{MULT}, \epsilon, sto\rangle \rhd^* \langle r, \mathcal{CA}\,[\![ae_1]\!] : \mathbf{MULT}, z_2, sto\rangle \rhd^*$$
$$\langle r, \mathbf{MULT}, z_1 : z_2, sto\rangle$$

We now apply the transition rule for **MULT**, and get

$$\langle r, \mathbf{MULT}, z_1 : z_2, sto\rangle \rhd^* \langle r, \epsilon, (z_1 \cdot z_2), sto\rangle$$

Since $ae_1 \cdot ae_2 \rightarrow_{ae} (z_1 \cdot z_2)$ in the semantics of AWL, the proof is complete.

6.8 Commands

In section 6.3 we defined the meaning of commands and instructions. We will use these definitions to make a theorem that expresses the correctness of the translation of commands. The theorem expresses, that if a execution of S terminates in a state in the semantics of AWL, then it will also terminate in the semantics of the abstract machine AM with the resulting states being equal. This also applies the other way around. The theorem also expresses that if the execution of S from one state loops in one of the semantics then it will also loop in the other.

Theorem 6.8.1 *For every statement S of AWL we have that $S_{AWL}\,[\![S]\!] = S_{AM}\,[\![S]\!]$*[5]

The theorem is proved in two stages expressed by lemma A.4.2 and lemma A.4.3.

Lemma 6.8.2 *For every statement S of AWL and stores sto and sto', we have that*

$$\textit{if } \langle S, sto\rangle \rightarrow sto' \textit{ then } \langle r, \mathcal{CS}\,[\![S]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, \epsilon, sto'\rangle$$

If the execution of S from the store sto terminates in the big step semantics for AWL, then the execution of the translated code from the store sto will also terminate in the semantics for AWLAM and the resulting stores will be equal.[6]

Proof: The proof of lemma A.4.2 is completed by induction on the shape of the derivation tree for $\langle S, sto\rangle \rightarrow sto'$. So we will prove the lemma for each command in AWL.

The case: [s - assign]

We asume that $\langle x = exp, sto\rangle \rightarrow sto'$ where $sto' = sto[l \mapsto v]$, $l = env_V(x)$ and $exp \rightarrow v$.

Using $\mathcal{CS}$ we get that

[5] [2, p. 74]
[6] [2, p. 75]

$$\mathcal{CS}[\![x = exp]\!]\,p = \mathcal{CE}[\![exp]\!] : \textbf{SAVE}\; n\; [LS]$$

where $n = mloc(p, x)$. From the expression lemmas we have

$$\langle r,\ \ \mathcal{CE}[\![exp]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r', \epsilon, v, sto\rangle$$

and from the semantic rules of AM we get

$$\langle r', \textbf{SAVE}\; n\; [LS], v, sto\rangle \rhd \langle r'', \epsilon, \epsilon, sto\,[(r(LS) + z]) \mapsto v\rangle$$

Since we have that $env_V(x) = r(LS) + z$ using the definition of LS, this completes the proof.

The case: [s - comp]

Using the semantics of AWL we have that $\langle S_1 S_2, sto\rangle \to sto'$ because $\langle S_1, sto\rangle \to sto''$ and $\langle S_2, sto''\rangle \to sto'$. Using $\mathcal{CS}$ we get that

$$\mathcal{CS}[\![S_1\, S_2]\!] = \mathcal{CS}[\![S_1]\!] : \mathcal{CS}[\![S_2]\!]$$

We apply the induction hypothesis to the premises and get that

$$\langle r,\ \ \mathcal{CS}[\![S_1]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, \epsilon, sto''\rangle \text{and}$$

$$\langle r'',\ \ \mathcal{CS}[\![S_2]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r', \epsilon, \epsilon, sto'\rangle$$

Since we can extend the code component we get that

$$\langle r,\ \ \ \mathcal{CS}[\![S_1]\!] :\ \mathcal{CS}[\![S_2]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r'',\ \ \mathcal{CS}[\![S_2]\!]\ \ , \epsilon, sto''\rangle \rhd^* \langle r', \epsilon, \epsilon, sto'\rangle$$

which completes the proof.

The rest of the cases can be found in Appendix A. We will now proceed to prove the following lemma[7].

Lemma 6.8.3 *For every command S of AWL and stores sto and sto$'$, we have that*

$$\text{if } \langle r, \mathcal{CS}[\![S]\!], \epsilon, sto\rangle \rhd^k \langle r', \epsilon, e, sto'\rangle \text{ then } \langle S, sto\rangle \to sto'$$

So if the execution of the code for S from a storage s terminates, then the AWL semantics of S from s will terminate in a state being equal to the storage of the terminal configuration.

Proof: We will prove lemma A.4.3 by induction on the length k of the computation sequence on AM. If $k = 0$ then the result holds because $\mathcal{CS}[\![S]\!] = \epsilon$ is impossible. So we assume that it holds for $k \le k_0$ and will prove that it holds for $k = k_0 + 1$. We make a case study on the command S.

The case: $x = exp;$

We have that $\mathcal{CS}[\![x = exp;]\!] = \mathcal{CE}[\![exp]\!] : \textbf{SAVE}\; n\; [LS]$, so we assume that

$$\langle r, \mathcal{CE}[\![exp]\!] : SAVE\; n\; [LS], \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

[7] [2, p. 77]

Since we can split the instruction sequence into two we have that

$$\langle r, \mathcal{CE}\,[\![exp]\!]\,, \epsilon, sto\rangle \triangleright^{k_1} \langle r'', \epsilon, e'', sto''\rangle \text{and}$$

$$\langle r, SAVE\; n\;[LS], e'', sto''\rangle \triangleright^{k_2} \langle r', \epsilon, e, sto'\rangle$$

where $k_1 + k_2 = k_0 + 1$. From the expression lemmas we get that $sto'' = sto$ and $e'' = v$ where $exp \rightarrow v$. Using the semantics of **SAVE** we see that $sto' = sto[(r(LS) + z) \mapsto v]$. It follows from [s - assign] that $\langle \mathbf{x} = \mathbf{exp};, sto\rangle \rightarrow sto'$, which completes the proof.

The case: $if(be)\{S_1\}else\{S_2\}\; true$

We have that

$$\mathcal{CS}\,[\![if(b)\{S_1\}else\{S_2\}]\!] =$$

$$\mathcal{CB}\,[be] : \mathbf{JUMPF}\; n_1 : \mathcal{CS}\,[\![S_1]\!] : \mathbf{JUMP}\; n_2 : \mathbf{LABEL}\; n_1 : \mathcal{CS}\,[\![S_2]\!] : \mathbf{LABEL}\; n_2$$

We assume that

$$\langle r, \mathcal{CB}\,[be] : \mathbf{JUMPF}\; n_1 : \mathcal{CS}\,[\![S_1]\!] : \mathbf{JUMP}\; n_2 : \mathbf{LABEL}\; n_1 : \mathcal{CS}\,[\![S_2]\!] : \mathbf{LABEL}\; n_2, \epsilon, sto\rangle \triangleright^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

Since we can split up the code component we get

$$\begin{array}{lll}
\langle r, \mathcal{CB}\,[\![be]\!], \epsilon, sto\rangle & \triangleright^{k_1} & \langle r'''', \epsilon, e''', sto''''\rangle \\
\langle r'''', \mathbf{JUMPF}\; n_1, e''', sto''''\rangle & \triangleright^{k_2} & \langle r''', \epsilon, e'', sto'''\rangle \\
\langle r''', \mathcal{CS}\,[\![S_1]\!], e'', sto'''\rangle & \triangleright^{k_3} & \langle r'', \epsilon, e', sto''\rangle \\
\langle r'', \mathbf{JUMP}\; n_2, e', sto''\rangle & \triangleright^{k_4} & \langle r', \epsilon, e, sto'\rangle
\end{array}$$

where $k_1 + k_2 + k_3 + k_4 = k_0 + 1$ and $k_2, k_4 = 1$.

Since $\mathcal{CB}\,[\![be]\!]$ and **JUMPF** does not change the storage, we have that $sto'''' = sto'''$ and $sto'' = sto'$. Likewise $\mathcal{CS}\,[\![S_1]\!]$ and **JUMP** does not change the evaluation stack so we have that $e'' = e' = e = \epsilon$. We assume that $e''' = tt$.

Since $k_3 \leq k_0$ we can apply the induction hypothesis to this computation and then we have that $\langle S_1, sto\rangle \rightarrow sto'$

The rule [S-if-true] gives the required $\langle if(be)\{S_1\}else\{S_2\}, sto\rangle \rightarrow sto'$. The proof of $if(be)\{S_1\}else\{S_2\}false$ is analogous.

The remaining proofs of this lemma can be found in Appendix A.

6.9 Summary

In this chapter we have (almost) proved that the translation functions defined in the previous chapter are correct. We can not claim to have proven the total correctness of the translation, since we have not proved the correctness of rule and ant type declarations due to a lack of time. It's obious that with the machine architecture of AWLAM, we are still far from any normal hardware implemented machine (such

as the pentium). However we are now one step closer, and with the proof made in this chapter, one could carry on towards an even lower level.

When making a proof like this, we make a computation sequence for each sequence of translated code. During this process, one is certain to find errors or misunderstandings in the translated code. This naturally makes it a very good exercise to do when wanting to translate a programming language into another language.

In the next chapter we will implement the results of our theoretical work.

Bibliography

[1] Hans Hüttel. *Pilen ved træets rod.* Aalborg University, 2003.

[2] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications.* Wiley, revised internet edition edition, July 1999.

[3] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java: Compilers and Interpreters.* Prentice Hall, 2000.

Chapter 7

Implementation

In the following section we will describe how AWL is implemented from the high-level language to the running program. First off we will describe the process from the AWL document to the AWLAM document. In this section we have scanning, parsing, identification, type checking and finally code generation. After this we will describe how the abstract machine works and how we will get from the AWLAM document to a running program using an interpreter.

7.1 Scanning and Parsing

For this task we have chosen to use a tool that does the task for us. We have chosen to use SableCC[1], because it both scans and parses the code, unlike e.g. JLex that only scans the code or JCup that only parses. Furthermore SableCC produces an abstract syntax tree that we will use in the later phases of compiling. The scanner and parser that SableCC generates are based on a kind of Extended Backus Naur Form (EBNF). This document can be found in the appendix.

In lexical analysis, or scanning as it is also called, the input program is scanned and divided into tokens that the parser can use. A token is described by its kind and its spelling. This means that in this part of the compiler, identifiers, keywords and other single parts of the program are recognized and put into a token stream. This token stream is then used by the parser, in that it is examined in order to see whether the statements made in the original program match those described in the grammar.

The purpose of parsing is to determine whether a stream of tokens is valid in accordance to the language – and if this is the case, to group the tokens into larger pieces, such as Commands or Expressions.

An important concept in parsing is unambiguity, meaning that a specific sentence has one and only one parse tree. The reason for this is that a sentence with more than one parse tree can lead to different end results. Just consider the simple mathematical sentence $2 + 2 \cdot 5$. To us humans it is easy to see that the result is 12, namely by unconsciously adding parentheses in order to determine the precedence: $2 + (2 \cdot 5)$. To the program, however, if nothing else is specified, the result might as well be 20: $(2 + 2) \cdot 5$. Luckily, if we have created our EBNF grammar correctly this should not be a problem.

As mentioned earlier in this section, SableCC makes an abstract syntax tree for the program. Furthermore it produces a tree-walker which is based on an extended visitor pattern.

7.2 Identification and Type Checking

Using the before mentioned abstract syntax tree, we can now perform identification and type checking also known as contextual analysis.

[1] `http://www.sablecc.org/`

Expression type	Input	Output
or-Expression	*Boolean* × *Boolean*	→ *Boolean*
and-Expression	*Boolean* × *Boolean*	→ *Boolean*
equality-Expression	*Direction* × *Direction*	→ *Boolean*
	Integer × *Integer*	→ *Boolean*
	Boolean × *Boolean*	→ *Boolean*
relational-Expression	*Integer* × *Integer*	→ *Boolean*
add-Expression	*Integer* × *Integer*	→ *Integer*
mult-Expression	*Integer* × *Integer*	→ *Integer*
unary-Expression	*Boolean*	→ *Boolean*
	Direction	→ *Direction*
	Integer	→ *Integer*

Figure 7.1: Expression hierarchy

> "The first task of the contextual analyzer is to relate each applied occurrence of an identifier in the source program to the corresponding declaration."[2]

To make sure that a program does not violate any contextual rules, one first has to look at identification.

What happens in the identification phase is that when an identifier is encountered, the identification process checks to see whether this identifier has been declared earlier in the program. If it has not, then the identifier ought to be about being declared (*varident*:*integer*) or else the program is ill formed and an error will be generated.

One should notice when reading the above, that in order to positively know whether a reached identifier has previously been encountered, you would have to search though all of the program examined so far. In our case this is not so, however. Instead we will use an identification table, in which all the identifiers are stored along with their type and other relevant information. Using this method, when encountering an identifier (assuming that this is not the declaration), the table is simply checked for previous occurrences of the identifier.

> "The second task of the contextual analyzer is to ensure that the source program contains no type errors."[3]

In type checking we need to make certain that all expressions yield the expected type. An example of this is the rule *VarInit*, which might look like this: *var id* : *integer* $= 5 + 5$;. Here we need to make sure that the expression $5 + 5$ yields an *integer*. This is one of the obvious rules – one not so obvious is when, say, a non turn based rule is involved. Here we need to check whether the argument types sent along with the rule call match those expected. Furthermore we need to check if the correct type is returned by the return command in the rule, and check that it is not placed in such a way that it will result in unreachable statements.

In figure 7.1 we show the expression hierarchy for AWL. Here we see that if we have the expression $5 + 3 * 1$ we must first evaluate the $3 * 1$ part, and then evaluate the result of it along with the 5+ part. This is implemented in type check in a way so that when an Expression is encountered, a method *evalExpression*(*Expression*) is called. The method first checks to see if the expression has any or-Expressions in it – and if not, it continues down the list. If it does contain an or-Expression, the checked expression is divide into two parts; the left and right side of the 'or' and uses *evalExpression* on each part. When both parts have done evaluating they return their respective types to the previous method. The method can then evaluate the two expressions according to the above rules. What we end up with

[2] [3, page 136]
[3] [3, page 150]

is either a *SimpleType* or an error is discovered during the process. In the latter case an exception is thrown.

By this we can conclude that if the identification and type checking do not throw any exceptions, the program is well formed, and we can proceed to the next step of compiling which is the actual code generation.

7.3 Code Generation

In code generation we will continue using the abstract syntax tree generated by SableCC. We will use the rules for code generation described in chapter 5, and apply these when walking trough the syntax tree. A more elaborate description of the functions in the code generation can be found in the previous mentioned chapter.

The result of code generation is an intermediate code document used by the abstract machine AWLAM which is described next.

7.4 AWLAM Implementation

The implementation of the AWL abstract machine follows the operational semantics of AWLAM is close as possible. The only notable difference is the fact that the implementation works with a PC register (program counter) and the code is not placed on a stack, but in (in this case, simulated) memory. This is, however, the way a hardware machine would have been implemented, and it is merely a slight step down the abstraction ladder from the operational semantics. The reason for this minor abstraction in the operational semantics is explained in section 3.

The abstract machine is implemented in Java, but it could just as well have been any other programming language. It might have been more profitable to code an interpreter in an assembly language, though, but since the purpose of this one was really not execution speed, and since the interpreter is in the periphery of the project's subject, we have chosen to do it in a high-level language. The specific language Java was chosen because of it being the language of most experience to the implementer.

7.4.1 The Evaluation Stack

The standard environment of Java supports an implementation of a stack, which we have placed a wrapper around and used here. The wrapper serves only as an interface receiving and returning the basic type int instead of Objects as is the case with the standard Java implementation since ints are what we operate on in the rest of the implementation.

This makes it very easy to implement the AWLAM instructions that operate on the evaluation stack, since it is just a question of invoking the push and/or pop methods whenever the operational semantics dictates it.

7.4.2 Registers

There is a fixed amount of registers to be implemented into the AM. Therefore it was obvious to use an array which, in Java terms, has a fixed amount of indexes. For referring to the specific indexes of the array, i.e. the specific registers of the AM, we use so-called ***fields***, or ***final variables*** (constants). One per index in the array, each with a name of a register and with a (fixed) value of an index in the array. This makes us able to refer to e.g. the PC register by **reg[PC]**, provided the array is called reg.

```
} else if (instr[0].equals("JUMPF")) {
    String n = instr[1];
    int z = label(N(n));
    int b = stack.pop();
    reg[PC] = (b == FALSE ? z : reg[PC] + 1);
} else if
```

Figure 7.2: Variables equal to those in the operational semantics.

```
} else if(instr[0].equals("JUMPF")) {
    reg[PC] = (stack.pop() == FALSE ? label(N(instr[1]) : reg[PC] + 1);
} else if
```

Figure 7.3: Higher level of optimization.

7.4.3 Memory

The memory, too, is an array. Or in fact, two arrays; one for code, and one for data. This is not the normal way of doing so, and it prevents methods such as self-modifying code. However, we don't need such things in AWLAM and that combined with the fact that splitting memory into two makes it possible for us to keep code in a string array (as we interpret assembler-like code and not binary machine-like code) and data in an int array, made us choose to do so. We still use an overall memory size, though, which is by the way adjustable from the command line, and assign only as much code memory as needed while the rest goes to data memory.

As said, all data items are strictly integer values, represented by Java's int type. In AWL, however, we have other basic types, namely booleans and directions[4]. They both have a very limited amount of possible values, so we solve this by simply assigning an int value to each boolean and direction value. To represent these values, we use constants (again as Java *fields*) in order to easily be able to work with them. This way we can push e.g. a "true" value by **stack.push(TRUE);** given "stack" is the name of the evaluation stack, and "push" is the name of the method that pushes new values into the top of a given stack. Both is the case in our situation.

7.4.4 Interpretation

The interpretation itself is basically a while construct, running as long as the program counter points within the code memory. For each loop, the code line pointed out by PC is evaluated, and action is taken accordingly. This action includes updating the PC register, whether this means simply increasing it by one, as in most instructions, or changing it to a totally different value, as is the case with e.g. the JUMP instruction.

Most of the actions performed when an instruction is recognized could have been formatted quite differently, and possibly more efficiently. However, we wanted to make a clearer connection to the operational semantics, and therefore we often save a given value in a variable (named as in the operational semantics), just to use it for the last time during the program in the very next Java code line. See an example of this in figures 7.2 and 7.3.

In the operational semantics we have functions for different purposes. An example of this is the $\mathcal{N}[n]$ function which gives the value of a numeral n. Equally we get the "numerals" as strings so this is a suitable reason for constructing an *N(n)* method, converting a string n to an int. The *label(l)*, *m()*, and *r()* methods are of the same principle, only that their functions are respectively to return the memory location pointed to by a label, to return the value of a given memory location, and to return the value

[4] See e.g. section 2.8 about these.

of a register. All methods are implemented in a very simply way, and from a technical point of view they could probably have been omitted – however they serve a purpose of easy understanding as well as reference to the operational semantics.

7.5 Screenshots

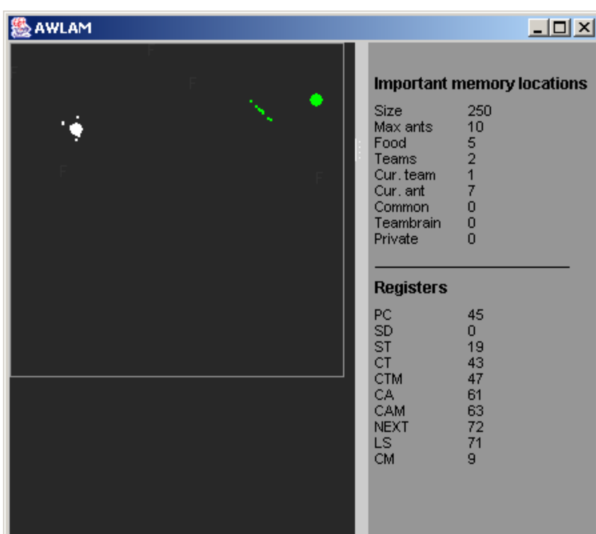
C:\Documents and Settings\Administrator\My Documents\helloWorld.awl

File Run

Compile Run Debug

```
1 world(250,10,5){
2
3   rule Rwalk(var team : integer; var ant : integer; var d : direction;){
4      var antSize : integer = 2 + getProperty(8);
5      var teamSize : integer = 4 + getProperty(7) + (getProperty(1) * antSize);
6      var lteam : integer = 9 + getProperty(6) + (getProperty(2)*2) + (team * teamSize
7      var lant : integer = lteam + getProperty(7) + 4 + (ant * antSize);
8      var x : integer = getProperty(lant);
9      var y : integer = getProperty(lant + 1);
10
11     if (d == left){ x = x - 1; } else{
12     if (d == right){ x = x + 1; } else{
13     if (d == up){ y = y - 1; } else{
14     if (d == down){ y = y + 1; } else{ skip; }}}}
15
16     if (x == getProperty(0)){ x = 0; } else{ skip; }
17     if (x == -1){ x = getProperty(0)-1; }  else{ skip; }
18
19     if (y == getProperty(0)){ y = 0; } else{ skip; }
20     if (y == -1){ y = getProperty(0)-1; }  else{ skip; }
21
22     setProperty(lant,x);
```

Parsing program...
Addressing...
Generation AWLAM code...
Writing code to file...
Done!



7.6 Summary

In this chapter we have had a look at some aspects of the implementation in the various stages from scanning and parsing, through identification and type checking to code generation, and at last the interpreter.

The scanning and parsing part is in our case handled by SableCC. Fed with something very close to an EBNF of the AWL language, the tool provides us with a scanner and parser able to verify a given piece of AWL code by producing an abstract syntax tree and traversing through it using an extended visitor's pattern.

Identifiers are kept track of by an identification table so that we will not have to look through the whole program every time an identifier is encountered. Next phase is type checking, where we make sure that for e.g. every integer variable declaration, also the value initializing it must be an integer.

Code generation is based on the rules set up earlier in this report and produces program code, interpretable by the AWLAM interpreter.

The interpreter itself emulates a machine, in that it contains registers, memory and an evaluation stack. It runs through the AWLAM code from one end to the other, reacting on the instructions it sees, whether the instructions tell the interpreter to calculate something and proceed or jump to another place in the code.

Chapter 8

Conclusion

In the preceding chapters we have described the development of the programming language AWL. We will now compare what was required of the programming language with what has happened in the report. We will go through these requirements one by one and describe how we have dealt with the problem[1].

AWL has to contain high level language constructs, such as in C: We have implemented AWL to contain high-level constructs such as while and if statements, along with variable and array declarations.

AWL has to provide specific constructs for the programmer, so that rules such as $walk(LEFT)$**are easy to create:** AWL provides ways for the programmer to declare a rule, and then a way to call this rule from a command or an expression. Furthermore we have provided a construct that allows the programmer direct access to the memory so that creating an ant of moving an ant becomes possible. This makes the language very flexible for the programmer using it.

AWL has to provide a construct to allow the world programmer to move the focus from one ant and team to another ant and team: AWL contains a command that allows an ant to be processed i.e. moved or what ever the ant creator wants his or her ant to do.

AWL has to provide some "ant memory" which differ in scope: In AWL each ant will have its own memory, that no other ant can access. Furthermore AWL provides a team memory that all ants from a team can access and modify, and a common memory that all ants regardless of team can access.

AWL has to provide a construct for creating teams: We have added a construct that can add a team to a game by a simple declaration.

AWL has to compile to an abstract machine, which we will call AWLAM(AWL Abstract Machine): The abstract machine AWLAM has been defined and implemented. Also an interpreter to run the code generated by AWLAM has been defined and implemented. This interpreter shows a graphical representation of the game that it interprets.

Below we will describe how the product of the report adheres to the goals of the report. We will, like with the requirements, describe each goal one by one.

Define the grammar of the high-level language AWL, using Backus Naur Form(BNF): In chapter 2 of the report the grammar of AWL is described. This grammar is using the BNF notation form, and describes the constructs of the language.

Define an operational big step semantic for AWL: In chapter 3 The big step semantic of AWL is shown. Here, the full description of how the semantics works is given. An abstract syntax that the

[1]The requirements can also be found in chapter 1

semantics is based on has been defined. There is a thorough description of each syntactic category from this abstracts syntax, and of the rules from each of these categories.

Define the abstract machine AWLAM and have the AWL compile to this: In chapter 4 the abstract machine(AWLAM) has been defined. We here define the abstract syntax that AWLAM adheres to, and give a description of each instruction found in the syntax.

Prove that the translated code is actually equivalent with the original AWL code: In order to prove the translated code is equivalent with the original AWL code, we have in chapter 5 described how the various semantical rules from AWL translates to AWLAM. We then, in chapter 6, proceed by proving that the implementation is correct, by comparing the translated code with the semantics defined in chapter 3. We here use different forms of induction as a proof method.

Our main goal in this project was to define the operational semantics of AWL, and then prove that a translation of AWL code into some target language would actually be correct. To do that it has been necessary to define a lot of other things as well, which was not a part of the goal as such, but which enables and helps us to reach the desired result. Before defining an operational semantics it was a necessity to have a syntax which was well defined, and actually showed the details of all constructs. As such it would have been enough to just have an abstract syntax, but that might have become rather complex since the definition of the syntax also gave us insight into what the problem area was actually about.

As such an operational semantics serves as a clear and precise notation that shows how the language actually behaves when being used. However there is no actual standard notation, which makes it difficult to describe the semantics in a way that is easy to read for everyone. We have aimed at defining a notation that both satisfies the need for a precise definition, but also a notation that should be somehow easy to read, compared to the relative complexity of the matter.

In the definition of the abstract machine the main issue was to make a machine that was simple and easy to understand, but yet at a higher level than for example the Pentium platform is today. A lot of issues arise when designing a piece of software at this level, and we have tried to make it as abstract as possible, without actually going high level. It would of course have been possible to use an existing abstract machine, but we felt that it would give us a better feeling with the machine to actually develop it ourselves, and also it enabled us to leave out aspects, which are indeed important from a general point of view, but which were not central to our project.

The definition of an operational semantics for the abstract machine was of course central to the task of proving the correctness of the translation process, and we have partially proved the equivalence of the two operational semantics using induction, giving us a mathematical proof of the correctness.

Bibliography

[1] Hans Hüttel. *Pilen ved træets rod.* Aalborg University, 2003.

[2] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications.* Wiley, revised internet edition edition, July 1999.

[3] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java: Compilers and Interpreters.* Prentice Hall, 2000.

Index

abstract machine, 86
abstract syntax tree, 84, 86
AWLAM, 86
  program example, 57

BNF, 12, 14, 52

code generation, 60, 86
code store, 49
commands, 40
computation sequence
  looping, 58
  terminating, 58
contextual analysis, 84

data store, 49
declarations
  ant type, 16, 38
  array, 17, 36
  memory, 14
    common, 38
    private, 40
    team, 39
  rule, 15, 37
  team, 17
  turn, 38
  variable, 17, 35
derivation tree, 47

EBNF, 84
environment, 13
  procedure, 26
  variable, 26
environment-store model, 23
evaluation stack, 50, 86
expressions
  arithmetic, 17, 28
  boolean, 17, 31
  direction, 33
  relational, 17

functions, 13

grammar, 14

identification, 84
implementation, 84, 86
instruction set, 52
instructions, 52
interpretation, 87

lexical analysis, 84
literals, 18
location, 13

memory, 87
Myrekrig, 11

notation, 13

operational semantics, 28, 52
  AWLAM, 53

parameters
  actual, 16, 44
  formal, 16, 44
parser, 84
partial functions, 13
precedence, 17
program counter, 50

registers, 49, 86

SableCC, 84
scanner, 84
semantics
  big step, 21
  natural, 21
  operational, 21
standard environment, 47
syntactic categories, 21, 22
  constructs, 23
syntax
  abstract, 22
  concrete, 22

token, 84
transition systems, 21, 28
translation functions, 62
type checking, 84

variable bindings
  dynamic, 26
  static, 26

world construct, 46

Appendix A

Provable Correct implementation

In this appendix we have listed all the proofs of translation correctness.

A.1 Variable Declarations

A.1.1 Variables

In this section we will prove that the translation of variable declarations is correct. We define a lemma to express the correctness.

Lemma A.1.1 *For all variable declarations we have that*

$$\textit{if } \langle D_V, env_V, sto\rangle \rightarrow (env'_V, sto') \textit{ then } \langle r, \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, \epsilon, sto'\rangle$$

where $env'_V(next) = r'(NEXT)$

So each variable must be stored at the same storage location in the two semantics. Also the pointer next must point to the same location as the register NEXT after declaration.

Proof: We will make the proof by induction on the shape of the derivation tree.

The case: [Dv-variable-declaration-empty]

We assume that $\langle \varepsilon, env_V, sto\rangle \rightarrow_{D_V} (env_V, sto)$. Using the translation function we get that $\mathcal{CD_V}\,[\![\epsilon]\!]\,p =$ **NOOP**. Using the semantics of **NOOP** we get that

$$\langle r, \textbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r, \epsilon, \epsilon, sto\rangle$$

which completes the proof of this case.

The case: [Dv-variable-declaration]

We assume that $\langle \textbf{var}\ x{:}type{=}exp; D_V, env_V, sto\rangle \rightarrow (env'_V, sto')$ holds because

$$\langle D_V, env_V\,[x \mapsto (type, l)]\,[next \mapsto new\,(l)]\,, sto\,[l \mapsto v]\rangle \rightarrow_{D_V} (env'_V, sto')\text{(which is the premise)}$$

because $l = env_V(next)$.

Using the code translation function we get that

$$\mathcal{CD_V}\,[\![\textbf{var}\ x : type{=}exp;D_V]\!]\,p = \mathcal{CE}\,[\![exp]\!] : \textbf{SAVE}\ 0[NEXT] : \textbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p$$

We can now make the following computation sequence.

$$\begin{aligned}
&\langle r, \mathcal{CE}\,[\![exp]\!] : \textbf{SAVE}\ 0[NEXT] : \textbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto\rangle \rhd^*\\
&\langle r, \textbf{SAVE}\ 0[NEXT] : \textbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p, v, sto\rangle \rhd\\
&\langle r, \textbf{NEXT} : \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto''\rangle \rhd\\
&\langle r'', \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto''\rangle
\end{aligned}$$

We see that $sto'' = [l \mapsto v]$, so it follows that the variables are stored at the correct locations.

Applying the induction hypothesis to the premise we get that

$$\langle r'', \mathcal{CD_V}\,[\![D_V]\!]\,p, \epsilon, sto''\rangle \rhd^* \langle r', \epsilon, \epsilon, sto'\rangle$$

which completes the computation which ends in the required state. It follows from the computation sequence that $env'_V(next) = r'(NEXT)$. This completes the proof of lemma A.1.1.

A.1.2 Arrays

The following lemma expresses the correctness of array declarations.

Lemma A.1.2 *For all array declarations we have that*

$$\textit{if}\ \langle D_A, env_V, sto\rangle \rightarrow (env'_V, sto')\ \textit{then}\ \langle r, \mathcal{CD_A}\,[\![D_A]\!]\,p, \epsilon, sto\rangle \rhd \langle r', \epsilon, \epsilon, sto'\rangle$$

where $r(NEXT) = env_V(next)$

So each array must be stored at the same storage locations in the two semantics. Also the pointer next must point to the same location as the register NEXT after declaration.

Proof: We will use induction on the shape of the derivation tree to prove lemma A.1.2.

The case: [Da-declaration-empty]

We assume that $\langle \epsilon, env_V, sto\rangle \rightarrow_{D_A} (env_V, sto)$. Using the translation function we get that $\mathcal{CD_A}\,[\![\epsilon]\!]\,p = \textbf{NOOP}$, and with the semantics of **NOOP**we have that

$$\langle r, \textbf{NOOP}, \epsilon, sto\rangle \rhd \langle r, \epsilon, \epsilon, sto\rangle$$

which completes the proof of this case.

The case: [Da-declaration]

We asume that

$$\langle \textbf{array}\ x[n] : type{=}exp; D_A, env_V, sto\rangle \rightarrow_{D_A} (env'_V, sto')$$

because

$$\langle D_A, env_V[x \mapsto (type, l, z)][next \mapsto new\ (l, z))], sto[l_i \mapsto v]\rangle \rightarrow_{D_A} (env'_V, sto')$$

where
$i \in [0..z-1]$ and $l = env_V\ (next)$ and $z > 0$.
Using the code translation function we get that

$$\mathcal{CDA}\,[\![array\ x[n] : type = exp\ D_A]\!]\, p =$$

$$PUSH\ n_3 : LABEL\ n_1 : DUP : PUSH\ 0 : EQ : NEG : JUMPF\ n_2 : \mathcal{CE}\,[\![exp]\!] : SAVE\ 0[NEXT] :$$

$$NEXT : PUSH\ 1 : SWAP : SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CDA}\,[\![D_A]\!]\, p$$

We can now make the following computation sequence:

$$\left\langle r, \begin{array}{l} PUSH\ n_3 : LABEL\ n_1 : DUP : PUSH\ 0 : EQ : NEG : JUMPF\ n_2 : \mathcal{CE}\,[\![exp]\!] : SAVE\ 0[NEXT] \\ : NEXT : PUSH\ 1 : SWAP : SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD}_{\mathcal{A}}\,[\![D_A]\!]\, p \end{array}, \epsilon, sto \right\rangle \rhd^7$$

$$\left\langle r', \begin{array}{l} \mathcal{CE}\,[\![exp]\!] : SAVE\ 0[NEXT] : NEXT : PUSH\ 1 : SWAP : \\ SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD}_{\mathcal{A}}\,[\![D_A]\!]\, p \end{array}, e, sto \right\rangle \rhd^*$$

$$\langle r'', \ PUSH\ 1 : SWAP : SUB : JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD}_{\mathcal{A}}\,[\![D_A]\!]\, p\ \ , e, sto'\rangle \rhd^*$$

$$\langle r''', \ JUMP\ n_1 : LABEL\ n_2 : POP : \mathcal{CD}_{\mathcal{A}}\,[\![D_A]\!]\, p\ \ , e', sto'\rangle \rhd^*$$

$$\langle r'''', \ \mathcal{CD}_{\mathcal{A}}\,[\![D_A]\!]\, p\ \ , \epsilon, sto''\rangle \rhd^*$$

$$\langle r', \epsilon, \epsilon, sto'\rangle$$

We get the first part of the computation using the semantics of AM, and we see that $sto'' = sto[l_i \mapsto v]$ where $i \in [0..\mathcal{N}[\![n]\!] - 1]$ as required. We get the last part by applying the induction hypothesis to the premise. This completes the proof of lemma A.1.2.

A.1.3 Common Memory

The following lemma expresses the correctness of common memory variable declarations.

Lemma A.1.3 *For all common memory variable declarations we have that*

$$\textit{if}\ \langle D_{MC}, env_V, sto\rangle \rightarrow (env'_V, sto')\ \textit{then}\ \langle r, \mathcal{CD}_{\mathcal{MC}}\,[\![D_{MC}]\!]\, p, \epsilon, sto\rangle \rhd \langle r', \epsilon, \epsilon, sto'\rangle$$

where $r'(NEXT) = env'_V(next)$.
So each common variable must be stored at the same storage location in the two semantics. Also the pointer next must point to the same location as the register NEXT after declaration.

Proof: We will use induction on the shape of the derivation tree to prove lemma A.1.3.

The case: [Dmc-common-empty]

We assume that $\langle \varepsilon, env_V, sto\rangle \rightarrow_{D_{MC}} (env_V, sto)$. Using the translation function we get that $\mathcal{CD}_{\mathcal{MC}}\,[\![\epsilon]\!]\, p = \textbf{NOOP}$.
Using the semantics of **NOOP** we get that

$$\langle r, \mathbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r, \epsilon, \epsilon, sto\rangle$$

which completes the proof of this case.

The case: [Dmc-common]

We assume that $\langle \mathbf{common\,var}\ x : type{=}exp; D_{MC}, env_V, sto\rangle \rightarrow (env'_V, sto')$ holds because

$$\langle D_{MC}, env_V\,[x \mapsto (type, z)]\,[next \mapsto new(l)]\,, sto[l \mapsto v][COMMONDECLS \mapsto z + 1]\rangle$$

Using the code translation function we get that

$$\mathcal{CD}_{\mathcal{MC}}\,[\![\mathbf{common\,var}\ x : type = exp; CD_{MC}]\!]\,p =$$

$$\mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ 0\ \ [NEXT] : NEXT : \mathbf{LOAD}\ 6\ [SD] : \mathbf{PUSH}\ 1 : \mathbf{ADD} : \mathbf{SAVE}\ 6\ \ [SD] : \mathcal{CD}_{\mathcal{MC}}\,[\![D_{MC}]\!]$$

We can now make the following computation sequence.

$$\langle r, \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ 0\ \ [NEXT] : NEXT : \mathbf{LOAD}\ 6\ [SD] : \mathbf{PUSH}\ 1\ : \mathbf{ADD} : \mathbf{SAVE}\ 6\ \ [SD] : \mathcal{CD}_{\mathcal{MC}}\,[\![D_{MC}]\!], \epsilon, sto\rangle \triangleright^*$$

$$\langle r'', \mathbf{LOAD}\ 6\ [SD] : \mathbf{PUSH}\ 1\ : \mathbf{ADD} : \mathbf{SAVE}\ 6\ \ [SD] : \mathcal{CD}_{\mathcal{MC}}\,[\![D_{MC}]\!], \epsilon, sto'''\rangle \triangleright$$

$$\langle r'', \mathcal{CD}_{\mathcal{MC}}\,[\![D_{MC}]\!], \epsilon, sto''\rangle \triangleright$$

$$\langle r', \epsilon, \epsilon, sto'\rangle$$

We get the two first parts of the computation by using the semantics of AM. We see that $sto'' = sto[l \mapsto v]$ as required. The last computation is made by applying the induction hypothesis to the premise. It follows from the computation sequence that $r'(NEXT) = env'_V(next)$ which completes the proof.

A.1.4 Teambrain Memory

The following lemma expresses the correctness of teambrain memory variable declarations.

Lemma A.1.4 *For all teambrain memory variable declarations we have that*

$$\textit{if } \langle D_{MT}, env_V, sto\rangle \rightarrow (env'_V, sto')\ \textit{ then } \langle r, \mathcal{CD}_{\mathcal{MT}}\,[\![D_{MT}]\!]\,p, \epsilon, sto\rangle \triangleright \langle r', \epsilon, \epsilon, sto'\rangle$$

So that state of the storage must be identical after computation in the two semantics.

Proof: We will conduct the proof of lemma A.1.4 by induction on the shape of the derivation tree.

The case: [Dmt-team-empty]

We assume that $\langle \varepsilon, env_V, sto\rangle \rightarrow_{D_{MT}} (env_V, sto)$. Using the translation function we get that $\mathcal{CD}_{\mathcal{MT}}\,[\![\epsilon]\!]\,p = \mathbf{NOOP}$. Using the semantics of **NOOP** we get that

$$\langle r, \mathbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r, \epsilon, \epsilon, sto\rangle$$

which completes the proof of this case.

The case: [Dmt-team]

We assume that $\langle \mathbf{teambrain\,var}\ x\ :\ type\,; D_{MT}, env_V, sto\rangle \rightarrow (, env'_V, sto')$ because

$$\langle D_{MT}, env_V\,[x \mapsto (type, z)]\,, sto[TEAMDECLS \mapsto z+1]\rangle \to (env'_V, sto')$$

Using the code translation function we get that

$$\mathcal{CD}_{\mathcal{MT}}\,[\![\textbf{teambrain var}\ x : type;\, CD_{MT}]\!]\,p =$$
$$\textbf{LOAD}\ 7\ [SD] : \textbf{PUSH 1} : \textbf{ADD} : \textbf{SAVE}\ 7\ [SD] : \mathcal{CD}_{\mathcal{MT}}\,[\![CD_{MT}]\!]$$

We can now make the following computation sequence.

$$\langle r, \textbf{LOAD}\ 7\ [SD] : \textbf{PUSH 1} : \textbf{ADD} : \textbf{SAVE}\ 7\ [SD] : \mathcal{CD}_{\mathcal{MT}}\,[\![CD_{MT}]\!], \epsilon, sto\rangle \triangleright^4$$
$$\langle r'', \mathcal{CD}_{\mathcal{MT}}\,[\![CD_{MT}]\!], \epsilon, sto''\rangle \triangleright^*$$
$$\langle r', \epsilon, \epsilon, sto'\rangle$$

We get the first parts of the computation by using the semantics of AM. We see that $sto'' = sto[7 \mapsto z+1]$ as required ($TEAMDECLS = 7$). The last computation is made by applying the induction hypothesis to the premise, which completes the proof.

A.1.5 Private Memory

The following lemma expresses the correctness of private memory variable declarations.

Lemma A.1.5 *For all private memory variable declarations we have that*

$$\textit{if}\ \langle D_{MT}, env_V, sto\rangle \to (env'_V, sto')\ \textit{then}\ \langle r, \mathcal{CD}_{\mathcal{MT}}\,[\![D_{MT}]\!]\,p, \epsilon, sto\rangle \triangleright \langle r', \epsilon, \epsilon, sto'\rangle$$

So that state of the storage must be identical after computation in the two semantics.

Proof: We will conduct the proof of lemma A.1.5 by induction on the shape of the derivation tree.

The case: [Dmt-private-empty]

We assume that $\langle \varepsilon, env_V, sto\rangle \to_{D_{MP}} (env_V, sto)$. Using the translation function we get that $\mathcal{CD}_{\mathcal{MP}}\,[\![\epsilon]\!]\,p = \textbf{NOOP}$. Using the semantics of **NOOP** we get that

$$\langle r, \textbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r, \epsilon, \epsilon, sto\rangle$$

which completes the proof of this case.

The case: [Dmt-private]

We assume that $\langle \textbf{private var}\ x\ :\ type\ ; D_{MP}, env_V, sto\rangle \to (, env'_V, sto')$ because

$$\langle D_{MT}, env_V\,[x \mapsto (z, type)]\,, sto[PRIVATEDECLS \mapsto z+1]\rangle \to (env'_V, sto')$$

Using the code translation function we get that

$$\mathcal{CD}_{\mathcal{MP}}\,[\![\textbf{private var}\ x : type;\, CD_{MP}]\!]\,p =$$
$$\textbf{LOAD}\ 8\ [SD] : \textbf{PUSH}\ 1 : \textbf{ADD} : \textbf{SAVE}\ 8\ [SD] : \mathcal{CD}_{\mathcal{MP}}\,[\![CD_{MP}]\!]\,p$$

We can now make the following computation sequence.

$$\langle r, \mathbf{LOAD}\ \mathit{8}\ [SD] : \mathbf{PUSH}\ \mathit{1} : \mathbf{ADD} : \mathbf{SAVE}\ \mathit{8}\ [SD] : \mathcal{CD}_{\mathcal{MP}}\ [\![CD_{MP}]\!]\ p, \epsilon, sto\rangle \triangleright^4$$

$$\langle r'', \mathcal{CD}_{\mathcal{MP}}\ [\![CD_{MP}]\!], \epsilon, sto''\rangle \triangleright^*$$

$$\langle r', \epsilon, \epsilon, sto'\rangle$$

We get the first parts of the computation by using the semantics of AM. We see that $sto'' = sto[8 \mapsto z+1]$ as required ($PRIVATEDECLS = 8$). The last computation is made by applying the induction hypothesis to the premise, which completes the proof.

A.2 Parameters

The correctness of formal parameters is expressed by the following lemma.

A.2.1 Formal

Lemma A.2.1 *For all formal parameters we have that*

$$\textit{if } \langle P_F, env_V\rangle \rightarrow (env'_V) \textit{ then } \langle r, \mathcal{CP}_{\mathcal{F}}\ [\![P_F]\!]\ p, e, sto\rangle \triangleright \langle r', \epsilon, \epsilon, sto'\rangle$$

where the stack e contains the actual parameter values, and where sto′ has the parameters stored at the locations following $r(NEXT)$.

Proof: To make this proof we will assume that there are the same amount of actual parameters as there are formal parameters. We will prove lemma A.2.1 be induction on the shape of the derivation tree.

The case: [Pf-formal parameters-empty]

We assume that $\langle \varepsilon, env_V\rangle \rightarrow_{P_F} (env_V)$. Using the translation function we get that $\mathcal{CP}_{\mathcal{F}}\ [\![\epsilon]\!]\ p = \mathbf{NOOP}$. Using the semantics of **NOOP** we get that

$$\langle r, \mathbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r, \epsilon, \epsilon, sto\rangle$$

Since there are no formal parameters then there are no actual parameters either. This completes the proof of this case.

The case: [Pf-formal parameters]

We assume that $\langle var\ x\ :\ type; P_F, env_V\rangle \rightarrow_S env'_V$ holds because

$$\langle P_F, env_V\ [x \mapsto l]\ [next \mapsto new\ (l)]\rangle \rightarrow env'_V.\text{(which is the premise)}$$

because $l = env_V\ (next)$.

Using the code translation function we get that

$$\mathcal{CP}_{\mathcal{F}}\ [\![\mathbf{var}\ x : type; P_F]\!]\ p = \mathbf{SAVE}\ \mathit{0}\ [NEXT] : \mathbf{NEXT} : \mathcal{CP}_{\mathcal{F}}\ [\![P_F]\!]$$

We can now make the following computation sequence.

$$\langle r, \mathbf{SAVE}\ \mathit{0}\ [NEXT] : \mathbf{NEXT} : \mathcal{CP_F}\ [\![P_F]\!], e, sto\rangle \triangleright^*$$
$$\langle r'', \mathcal{CP_V}\ [\![P_F]\!]\ p, e', sto''\rangle \triangleright^*$$
$$\langle r', \epsilon, \epsilon, sto'\rangle$$

We see that $sto'' = [l \mapsto v]$, and it follows from the computation that the parameters are stored at the correct locations. The last computation is made by applying the induction hypothesis to the premise. This completes the proof.

A.2.2 Actual

The correctness of actual parameters is expressed by the following lemma.

Lemma A.2.2 *For all actual parameters we have that*

$$\textit{if}\ \langle P_A, env_V, sto\rangle \rightarrow (env'_V, sto')\ \textit{then}\ \langle r, \mathcal{CP_A}\ [\![P_F]\!]\ p, \epsilon, sto\rangle \triangleright \langle r', \epsilon, e, sto\rangle$$

where the stack e contains the actual parameter values.

Proof: We will prove lemma A.2.2 be induction on the shape of the derivation tree.

The case: [Pf-actual parameters-empty]

We assume that $\langle \varepsilon, env_V, sto\rangle \rightarrow_{P_F} (env_V, sto)$. Using the translation function we get that $\mathcal{CP_A}\ [\![\epsilon]\!]\ p =$ **NOOP**. Using the semantics of **NOOP** we get that

$$\langle r, \mathbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r, \epsilon, \epsilon, sto\rangle$$

Since there is no actual parameter then $e = \epsilon$. This completes the proof of this case.

The case: [Pf-actual parameters]

We assume that $\langle exp; P_A, env_V, sto\rangle \rightarrow_{P_A} (env'_V, sto')$ holds because

$$\langle P_A, env_V\ [next \mapsto new\ (l)]\ , sto\ [l \mapsto v]\rangle \rightarrow (env'_V, sto')$$

because $l = env_V\ (next)$.

Using the code translation function we get that

$$\mathcal{CP_A}\ [\![ae; P_A]\!]\ p =$$
$$\mathcal{CP_A}\ [\![P_A]\!] : \mathcal{CA}\ [\![ae]\!]$$

We can now make the following computation sequence.

$$\langle r, \mathcal{CP_A}\ [\![P_A]\!] : \mathcal{CA}\ [\![ae]\!], \epsilon, sto\rangle \triangleright^*$$
$$\langle r'', \mathcal{CA}\ [\![ae]\!], e', sto\rangle \triangleright^*$$
$$\langle r', \epsilon, e, sto\rangle$$

The first computation is done by applying the induction hypothesis on the premise and by using that we can extend the code base. It follows from the computation that the actual parameters will be placed on the stack.

A.3 Expressions

A.3.1 Arithmetic

The correctness of the implementation of the arithmetic expressions in AWL is expressed by lemma A.3.1.

Lemma A.3.1 *For all arithmetic expressions ae we have that*

$$\langle r, \mathcal{CA}\llbracket ae \rrbracket, \epsilon, sto\rangle \rhd \langle r, \epsilon, z, sto\rangle$$

where $env_P, env_V, sto \vdash ae \rightarrow_{ae} z$.

Furthermore, all intermediate configurations of this computation sequence will have a non-empty evaluation stack.

Proof: The proof of lemma A.3.1 is done by structual induction on ae.

The case: [ae-lit]

Using the code generation function $\mathcal{CA}$,we have that $\mathcal{CA}\llbracket n \rrbracket\, p = \mathbf{PUSH}\ n$. From the semantics of AM we have that

$$\langle r, \mathbf{PUSH}\ n, \epsilon, sto\rangle \rhd \langle r', \epsilon, z, sto\rangle$$

and since $n \rightarrow z$ in the operational semantics for AWL, we have completed the proof for [ae-lit].

The case: [ae-var]

We have that $\mathcal{CA}\llbracket x \rrbracket\, p = \mathbf{LOAD}\ n\,[LS]$, where LS is the register, which points to the local base address of the current routine p, and where $n = mloc\,(p, x)$ (the relative address of x inside p).

Using the semantics of AM we have that

$$\langle r, \mathbf{LOAD}\ n\,[LS], \epsilon, sto\rangle \rhd \langle r', \epsilon, sto\,(r\ (LS) + z)\,, sto\rangle$$

In the operational semantics of AWL we have that $x \rightarrow sto\,(env_V\,(x))$. Using the definition of LS and $mloc$ we see that $r(LS) + z = env_V(x)$, which completes the proof of this case.

The case: [ae-array]

We have that

$$\mathcal{CA}\llbracket x\,[ae] \rrbracket\, p = \mathcal{CA}\llbracket ae \rrbracket : \mathbf{PUSH}\ n : \mathbf{ADD} : \mathbf{LOADS}\ [LS]$$

where LS is the register, which points to the local base adress of the current routine p, and where $n = mloc(p, x)$ is the relative adress of the first element of the array variable x inside p.

We can make the computation sequence

$$\langle r, \mathcal{CA}\llbracket ae \rrbracket : \mathbf{PUSH}\ n : \mathbf{ADD} : \mathbf{LOADS}\ [LS]\,, \epsilon, sto\rangle \rhd^* \langle r', \epsilon, sto\,(r\ (LS) + z_1 + z_2)\,, sto\rangle$$

where $z_1 = \mathcal{N}[\![n]\!]$ and $ae \rightarrow_{ae} z_2$

The semantics of AWL states that $x[ae] \rightarrow_{ae} sto(env_V(x) + z_2)$. Using the definition of LS and $mloc$ we see that $r(LS) + z = env_V(x)$, which completes the proof of this case.

The case: [ae-common memory variable]

Using the translation function we have that

$$\mathcal{CA}[\![\mathbf{cmem}\ x]\!] = \mathbf{LOAD}\ n\ [CM]$$

where CM is the register pointing to the first common memory location and $n = mloc(memory, x)$, where n is the relative address of x in the memory scope. When applying the semantics of AM we get

$$\langle r, \mathbf{LOAD}\ n\ [CM], \epsilon, sto\rangle \triangleright \langle r', \epsilon, sto\,(r\,(CM) + z)\,, sto\rangle$$

In the operational semantics of AWL we have that $(\mathbf{cmem} x) \rightarrow_{ae} z_1$ where $env_V(x) = (integer, z_1)$. and $z_1 = (COMMONBASE + z_1)$.

Using the definition of CM and $mloc$ we see that $r(CM) + z = z_1$, which completes the proof of this case.

The case: [ae-teambrain memory variable]

We have that $\mathcal{CA}[\![\mathbf{tmem}\, x]\!]\, p = \mathbf{LOAD}\, n\ [CTM]$ where CTM is the register pointing to first teambrain memory location for the current team and $n = mloc(memory, x)$, where n is the relative address of x in the memory variable scope. Using the translation function we get that

$$\langle r, \mathbf{LOAD}\ n\ [CTM], \epsilon, sto\rangle \triangleright \langle r', \epsilon, sto\,(r\,(CTM) + z)\,, sto\rangle$$

Since $\mathbf{tmem}\ x \rightarrow_{ae} z_1$ where $env_V(x) = (integer, z_2)$ and

$$z_1 = sto\,(teamLoc\,(sto\,(CURRENTTEAM)) + TEAMALLOC + z_2)$$

we need to show that

$$r\,(CTM) + z = teamLoc\,(sto\,(CURRENTTEAM)) + TEAMALLOC + z_2$$

The definition of $teamLoc$ specifies that it will return the base storage location of a given team. Using that, the definition of CTM and $z = z_2$, we can see that the above statement holds.

The case: [ae-private memory variable]

We have that

$$\mathcal{CA}[\![\mathbf{pmem}\ x;]\!]\, p = \mathbf{LOAD}\ n\ [CAM]$$

where CAM is the register pointing to first private memory location for the current ant on the current team and $n = mloc(memory, x)$, where n is the relative address of x in the memory variable scope. We then have

$$\langle r, \mathbf{LOAD}\ n\ [CAM], \epsilon, sto\rangle \triangleright \langle r', \epsilon, sto\,(r\,(CAM) + z)\,, sto\rangle$$

Since **pmem** $x \rightarrow_{ae} z_1$ where $env_V(x) = (integer, z_2)$ and

$$z_1 = sto\,(antLoc\,(sto\,(CURRENTTEAM)\,, sto\,(CURRENTANT)) + ANTALLOC + z_2)$$

we need to show that

$$r\,(CAM) + z = antLoc\,(sto\,(CURRENTTEAM)\,, sto\,(CURRENTANT)) + ANTALLOC + z_2$$

The definition of *antLoc* specifies that it will return the base storage location of a given team and ant. Using that, the definition of CAM and $z = z_2$, we can see that the above statement holds.

The case: [ae-random]

Using the code translation function we get that

$$\mathcal{CA}\,[\![\mathbf{random}(ae)]\!]\,p = \mathcal{CA}\,[\![ae]\!] : \mathbf{RAN}$$

This results in the computation sequence

$$\langle r, \mathcal{CA}\,[\![ae]\!] : \mathbf{RAN}, \epsilon, sto\rangle \triangleright \langle r', \mathbf{RAN}, z_1, sto\rangle \triangleright \langle r, \epsilon, z_2, sto\rangle$$
$$\text{where } 0 \leq z_2 < z_1$$

The first computation is made by applying the induction hypothesis to ae and the second by using the semantics of **RAN**. It follows from the rule [ae-random] that this completes the proof.

The case: [ae-getProperty]

Using the code translation function we have

$$\mathcal{CA}\,[\![\mathbf{getProperty}(ae);]\!] = \mathcal{CA}\,[\![ae]\!] : \mathbf{LOADS}\ [SD]$$

and we there have the computation sequence

$$\langle r, \mathcal{CA}\,[\![ae]\!] : \mathbf{LOADS}\ [SD]\,, \epsilon, sto\rangle \triangleright^*$$
$$\langle r', \mathbf{LOADS}\ [SD]\,, z_1\,, sto\rangle \triangleright$$
$$\langle r'', \epsilon, z_2\,, sto\rangle$$

To make the first computation we apply the induction hypothesis to ae, and to make the second we use the semantics of **LOADS**. We see that $z_2 = sto(z_1)$, and using the rule [ae-getProperty] we see that this is the required result.

The case: [ae-par]

We have that

$$\mathcal{CA}\,[\![(ae)]\!] = \mathcal{CA}\,[\![ae]\!].$$

Applying the induction hypothesis to ae we get that

$$\langle r, \mathcal{CA}\,[\![ae]\!]\,, \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, z, sto\rangle$$

Since $(ae) \rightarrow_{ae} z$ the proof is complete.

The case: [ae-add]

We have that

$$\mathcal{CA}\,[\![ae_1 + ae_2]\!] = \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{ADD}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}\,[\![ae_1]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, z_1, sto\rangle \qquad \text{and}$$
$$\langle r', \mathcal{CA}\,[\![ae_2]\!], \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{ADD}, \epsilon, sto\rangle \rhd^* \langle r, \mathcal{CA}\,[\![ae_1]\!] : \mathbf{ADD}, z_2, sto\rangle \rhd^*$$
$$\langle r, \mathbf{ADD}, z_1 : z_2, sto\rangle$$

We now apply the transition rule for **ADD**, and get

$$\langle r, \mathbf{ADD}, z_1 : z_2, sto\rangle \rhd^* \langle r, \epsilon, (z_1 + z_2), sto\rangle$$

Since $ae_1 + ae_2 \rightarrow_{ae} (z_1 + z_2)$ in the semantics of AWL, the proof is complete.

The case: [ae-sub]

We have that

$$\mathcal{CA}\,[\![ae_1 - ae_2]\!] = \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{SUB}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}\,[\![ae_1]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, z_1, sto\rangle \qquad \text{and}$$
$$\langle r', \mathcal{CA}\,[\![ae_2]\!], \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{SUB}, \epsilon, sto\rangle \rhd^* \langle r, \mathcal{CA}\,[\![ae_1]\!] : \mathbf{SUB}, z_2, sto\rangle \rhd^*$$
$$\langle r, \mathbf{SUB}, z_1 : z_2, sto\rangle$$

We now apply the transition rule for **SUB**, and get

$$\langle r, \mathbf{SUB}, z_1 : z_2, sto\rangle \rhd^* \langle r, \epsilon, (z_1 - z_2), sto\rangle$$

Since $ae_1 - ae_2 \rightarrow_{ae} (z_1 - z_2)$ in the semantics of AWL, the proof is complete.

The case: [ae-mult]

We have that

$$\mathcal{CA}\,[\![ae_1 \cdot ae_2]\!] = \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \textbf{MULT}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}\,[\![ae_1]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, z_1, sto\rangle \qquad \text{and}$$
$$\langle r', \mathcal{CA}\,[\![ae_2]\!], \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \textbf{MULT}, \epsilon, sto\rangle \rhd^* \langle r, \mathcal{CA}\,[\![ae_1]\!] : \textbf{MULT}, z_2, sto\rangle \rhd^*$$
$$\langle r, \textbf{MULT}, z_1 : z_2, sto\rangle$$

We now apply the transition rule for **MULT**, and get

$$\langle r, \textbf{MULT}, z_1 : z_2, sto\rangle \rhd^* \langle r, \epsilon, (z_1 \cdot z_2), sto\rangle$$

Since $ae_1 \cdot ae_2 \rightarrow_{ae} (z_1 \cdot z_2)$ in the semantics of AWL, the proof is complete.

The case: [ae-div]

We have that

$$\mathcal{CA}\,[\![ae_1 / ae_2]\!] = \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \textbf{DIV}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}\,[\![ae_1]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, z_1, sto\rangle \qquad \text{and}$$
$$\langle r', \mathcal{CA}\,[\![ae_2]\!], \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \textbf{DIV}, \epsilon, sto\rangle \rhd^* \langle r, \mathcal{CA}\,[\![ae_1]\!] : \textbf{DIV}, z_2, sto\rangle \rhd^*$$
$$\langle r, \textbf{DIV}, z_1 : z_2, sto\rangle$$

We now apply the transition rule for **DIV**, and get

$$\langle r, \textbf{DIV}, z_1 : z_2, sto\rangle \rhd^* \langle r, \epsilon, (z_1 / z_2), sto\rangle$$

Since $ae_1 \cdot ae_2 \rightarrow_{ae} (z_1 / z_2)$ in the semantics of AWL, the proof is complete.

The case: [ae-rulecall]

We have that

$$\mathcal{CA}\,[\![r\,(P_A)]\!] = \mathcal{CP}_{\mathcal{A}}\,[\![P_A]\!] : \textbf{CALL}\ n_1, n_2 : \textbf{LOAD}\ 0\ [NEXT]$$

This gives us the computation sequence

$$\langle r, \mathcal{CP}_{\mathcal{A}}[\![P_A]\!] : \textbf{CALL}\ n_1, n_2 : \textbf{LOAD}\ 0\ [NEXT], \epsilon, sto\rangle \rhd^*$$

$$\langle r, \textbf{CALL}\ n_1, n_2 : \textbf{LOAD}\ 0\ [NEXT], e, sto'\rangle \rhd^*$$

$$\langle r, \textbf{LOAD}\ 0\ [NEXT], \epsilon, sto'\rangle \rhd$$

$$\langle r, \epsilon, z, sto'\rangle$$

We get the first computation by using lemma A.2.2. The second computation is made by using the semantics of **CALL** and the procedure protocols defined in the last chapter, which states (among other things) that after returning from a procedure, the actual parameters have been removed from the stack. The protocols also states that the return value is stored at location 0 relative to the address stored in $NEXT$, and using **LOAD** we therefore get the final computation. We see that the storage has changed which goes against the lemma - however since we haven't updated $NEXT$ the updated storage location will be overwritten, making the change irrelevant. This concludes the proof of lemma A.3.1.

A.3.2 Boolean

The correctness of the implementation of the boolean expressions in AWL is expressed by the following lemma.

Lemma A.3.2 *For all boolean expressions be we have that*

$$\langle r, \mathcal{CB}[\![be]\!], \epsilon, sto\rangle \rhd \langle r, \epsilon, b, sto\rangle$$

$$\textit{where } env_V, sto \vdash be \rightarrow_{be} b$$

Furthermore, all intermediate configurations of this computation sequence will have a non-empty evaluation stack.

Proof: The proof of the lemma is done by structual induction on *be*.

The case: [be-lit]

Using the code generation function we have that $\mathcal{CB}[\![bl]\!] = \{TRUE, FALSE\}$.

From the semantics of AWLAM we have that

$$\langle r, \mathcal{CB}[\![bl]\!], \epsilon, sto\rangle \rhd \langle r, \epsilon, b, sto\rangle$$

and since $bl \rightarrow b$ in the operational semantics for AWL, we have completed the proof for [be-lit].

The case: [be-var]

We have that $\mathcal{CB}[\![x]\!]\,p = \textbf{LOAD}\ n\ [LS]$, meaning that this proof is analog to that of [ae-var].

The case: [be-array]

We have that $\mathcal{CB}[\![x\,[ae]]\!]\,p = \mathcal{CB}[\![ae]\!] : \textbf{PUSH}\ n : \textbf{ADD} : \textbf{LOADS}\ [LS]$, meaning that this proof is analog to that of [ae-array].

The case: [be-common memory variable]

We have that $\mathcal{CB}\,[\![\mathbf{cmem}\,x]\!] = \mathbf{LOAD}\ n\ [CM]$, meaning that this proof is analog to that of [be-common memory variable].

The case: [be-teambrain memory variable]

We have that $\mathcal{CB}\,[\![\mathbf{tmem}\,x]\!]\,p = \mathbf{LOAD}\ n\ [CTM]$, meaning that this proof is analog to that of [be-teambrain memory variable].

The case: [be-private memory variable]

We have that $\mathcal{CB}\,[\![\mathbf{pmem}\,x;]\!]\,p = \mathbf{LOAD}\ n\ [CAM]$, meaning that this proof is analog to that of [ae-private memory variable].

The case: [be-getProperty]

Using the code translation function we have $\mathcal{CB}\,[\![\mathbf{getProperty}(ae);]\!] = \mathcal{CB}\,[\![ae]\!] : \mathbf{LOADS}\ [SD]$, meaning that this proof is analog to that of [ae-getProperty].

The case: [be-par]

We have that $\mathcal{CB}\,[\![(ae)]\!] = \mathcal{CB}\,[\![ae]\!]$, meaning that this proof is analog to that of [ae-getProperty].

The case: [be-equals(ae)]

We have that

$$\mathcal{CB}\,[\![ae_1 == ae_2]\!]\,p = \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{EQ}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\begin{aligned} &\langle r, \mathcal{CA}\,[\![ae_1]\!], \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, z_1, sto\rangle \qquad \text{and} \\ &\langle r', \mathcal{CA}\,[\![ae_2]\!], \epsilon, sto\rangle \triangleright^* \langle r'', \epsilon, z_2, sto\rangle \end{aligned}$$

Since we can extend the code base, we have that

$$\begin{aligned} &\langle r, \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{EQ}, \epsilon, sto\rangle \triangleright^* \langle r, \mathcal{CA}\,[\![ae_1]\!] : \mathbf{EQ}, z_2, sto\rangle \triangleright^* \\ &\langle r, \mathbf{EQ}, z_1 : z_2, sto\rangle \end{aligned}$$

We now apply the transition rule for **EQ**, and get

$$\langle r, \mathbf{EQ}, z_1 : z_2, sto\rangle \triangleright^* \langle r, \epsilon, (z_1 = z_2), sto\rangle$$

Since $ae_1 \cdot ae_2 \rightarrow_{ae} (z_1 = z_2)$ in the semantics of AWL, the proof is complete.

The case: [be-equals(be)]

We have that $\mathcal{CB}\,[\![be_1 == be_2]\!]\,p = \mathcal{CB}\,[\![ae_2]\!] : \mathcal{CB}\,[\![ae_1]\!] : \mathbf{EQ}$, meaning that this proof is analog to that of [be-equals(ae)].

The case: [be-equals(de)]

We have that $\mathcal{CB}[\![de_1 == de_2]\!]\,p = \mathcal{CB}[\![de_2]\!] : \mathcal{CB}[\![de_1]\!] : \mathbf{EQ}$, meaning that this proof is analog to that of [be-equals(ae)].

The case: [be-not-equals(ae)]

We have that

$$\mathcal{CB}[\![ae_1 != ae_2]\!]\,p = \mathcal{CA}[\![ae_2]\!] : \mathcal{CA}[\![ae_1]\!] : \mathbf{EQ} : \mathbf{NEG}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}[\![ae_1]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, z_1, sto\rangle \quad \text{and}$$
$$\langle r', \mathcal{CA}[\![ae_2]\!], \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}[\![ae_2]\!] : \mathcal{CA}[\![ae_1]\!] : \mathbf{EQ} : \mathbf{NEG}, \epsilon, sto\rangle \rhd^* \langle r, \mathcal{CA}[\![ae_1]\!] : \mathbf{EQ} : \mathbf{NEG}, z_2, sto\rangle \rhd^* \langle r, \mathbf{EQ} : \mathbf{NEG}, z_1 : z_2, sto\rangle$$

We now apply the transition rules for **EQ** and **NEG**, and get

$$\langle r, \mathbf{EQ}, z_1 : z_2, sto\rangle \rhd^* \langle r, \mathbf{NEG}, (z_1 = z_2), sto\rangle \rhd^* \langle r, \epsilon, (z_1 \neq z_2), sto\rangle$$

which completes the proof.

The case: [be-not-equals(be)]

We have that $\mathcal{CB}[\![be_1 != be_2]\!]\,p = \mathcal{CA}[\![be_2]\!] : \mathcal{CA}[\![be_1]\!] : \mathbf{EQ} : \mathbf{NEG}$, meaning that this proof is analog to that of [be-not-equals(ae)].

The case: [be-not-equals(be)]

We have that $\mathcal{CB}[\![de_1 != de_2]\!]\,p = \mathcal{CA}[\![de_2]\!] : \mathcal{CA}[\![de_1]\!] : \mathbf{EQ} : \mathbf{NEG}$, meaning that this proof is analog to that of [be-not-equals(ae)].

The case: [be-greater-than]

We have that

$$\mathcal{CB}[\![ae_1 > ae_2]\!]\,p = \mathcal{CA}[\![ae_2]\!] : \mathcal{CA}[\![ae_1]\!] : \mathbf{LE} : \mathbf{NEG}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}[\![ae_1]\!], \epsilon, sto\rangle \rhd^* \langle r', \epsilon, z_1, sto\rangle \quad \text{and}$$
$$\langle r', \mathcal{CA}[\![ae_2]\!], \epsilon, sto\rangle \rhd^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}[\![ae_2]\!] : \mathcal{CA}[\![ae_1]\!] : \mathbf{LE} : \mathbf{NEG}, \epsilon, sto\rangle \rhd^* \langle r, \mathcal{CA}[\![ae_1]\!] : \mathbf{LE} : \mathbf{NEG}, z_2, sto\rangle \rhd^* \langle r, \mathbf{LE} : \mathbf{NEG}, z_1 : z_2, sto\rangle$$

We now apply the transition rules for **LE** and **NEG**, and get

$$\langle r, \mathbf{LE}, z_1 : z_2, sto\rangle \triangleright^* \langle r, \mathbf{NEG}, (z_1 > z_2), sto\rangle \triangleright^* \langle r, \epsilon, (z_1 \leq z_2), sto\rangle$$

which completes the proof.

The proofs of the constructs [be-lower-than], [be-greater-than-or-equals] is analogous.

The case: [be-and]

We have that

$$\mathcal{CB}\,[\![be_1 \,\mathbf{and}\; be_2]\!]\,p = \mathcal{CB}\,[\![be_2]\!] : \mathcal{CB}\,[\![be_1]\!] : \mathbf{AND}$$

Applying the induction hypothesis to ae_1 and ae_2 results in

$$\langle r, \mathcal{CA}\,[\![ae_1]\!], \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, z_1, sto\rangle \qquad \text{and}$$

$$\langle r', \mathcal{CA}\,[\![ae_2]\!], \epsilon, sto\rangle \triangleright^* \langle r'', \epsilon, z_2, sto\rangle$$

Since we can extend the code base, we have that

$$\langle r, \mathcal{CA}\,[\![ae_2]\!] : \mathcal{CA}\,[\![ae_1]\!] : \mathbf{AND}, \epsilon, sto\rangle \triangleright^* \langle r, \mathcal{CA}\,[\![ae_1]\!] : \mathbf{AND}, z_2, sto\rangle \triangleright^* \langle r, \mathbf{AND}, z_1 : z_2, sto\rangle$$

We now apply the transition rules for **AND** we get

$$\langle r, \mathbf{AND}, z_1 : z_2, sto\rangle \triangleright^* \langle r, \mathbf{AND}, (z_1 \wedge z_2), sto\rangle$$

which completes the proof.

The proof of [be-or] is analogous.

A.3.3 Direction

All proofs of direction expressions are analogous to the proofs of arithmetic expressions.

A.4 Commands

The following theorem expresses, that if a execution of S terminates in a state in the semantics of AWL, then it will also terminate in the semantics of the abstract machine AM with the resulting states being equal. This also applies the other way around. The theorem also expresses that if the execution of S from one state loops in one of the semantics then it will also loop in the other.

Theorem A.4.1 *For every statement S of AWL we have that $S_{AWL}\,[\![S]\!] = S_{AM}\,[\![S]\!]$*

The theorem is proved in two stages expressed by lemma A.4.2 and lemma A.4.3.

Lemma A.4.2 *For every statement S of AWL and stores sto and sto', we have that*

$$\textit{if } \langle S, sto\rangle \to sto' \textit{ then } \langle r, \mathcal{CS}\,[\![S]\!], \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, \epsilon, sto'\rangle$$

If the execution of S from the store sto terminates in the big step semantics for AWL, then the execution of the translated code from the store sto will also terminate in the semantics for AWLAM and the resulting stores will be equal.

Proof: The proof of lemma A.4.2 is completed by induction on the shape of the derivation tree for $\langle S, sto\rangle \to sto'$. So we will prove the lemma for each command in AWL.

The case: [s - assign]

We asume that $\langle x = exp, sto\rangle \to sto'$ where $sto' = sto[l \mapsto v]$, $l = env_V(x)$ and $exp \to v$.

Using $\mathcal{CS}$ we get that

$$\mathcal{CS}\,[\![x = exp]\!]\,p = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ n\ [LS]$$

where $n = mloc(p, x)$. From the expression lemmas we have

$$\langle r,\ \ \mathcal{CE}\,[\![exp]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r', \epsilon, v, sto\rangle$$

and from the semantic rules of AM we get

$$\langle r', \mathbf{SAVE}\ n\ [LS], v, sto\rangle \rhd \langle r'', \epsilon, \epsilon, sto\,[(r(LS) + z]) \mapsto v\rangle$$

Since we have that $env_V(x) = r(LS) + z$ this completes the proof.

The case: [S assign array]

We assume that

$$\langle \mathbf{x}[ae] = exp;, sto\rangle \to sto'$$

where $sto' = sto\,[(l + z_1) \mapsto v]$, $ae \to z_1$ and $env_V\,(x) = (type, l, z_2)$ and $0 \leq z_1 < z_2$

We have that

$$\mathcal{CS}\,[\![x\,[ae] = exp;]\!]\,p = \mathcal{CE}\,[\![exp]\!] : \mathcal{CA}\,[\![ae]\!] : \mathbf{PUSH}\ n : \mathbf{ADD} : \mathbf{SAVES}\ [LS]$$

From $\mathcal{CE}$ and $\mathcal{CA}$ we get

$$\langle r, \mathcal{CE}\,[\![exp]\!] : \mathcal{CA}\,[\![ae]\!]\,, \epsilon, sto\rangle \rhd^* \langle r'', \mathcal{CA}\,[\![ae]\!]\,, v, sto\rangle \rhd^* \langle r''', \epsilon, z_1 : v, sto\rangle$$

Applying **PUSH** we get

$$\langle r''', \mathbf{PUSH}\ n, z_1 : v, sto\rangle \rhd \langle r'''', \epsilon, z_3 : z_1 : v, sto\rangle$$

where $z_3 = mloc\,(p, x)$. We now apply **ADD**

$$\langle r'''', \mathbf{ADD}, z_3 : z_1 : v, sto\rangle \rhd \langle r''''', \epsilon, z_4 : v, sto\rangle$$

Finally applying **SAVES** $[LS]$ we get

$$\langle r''''', \textbf{SAVES}\ [LS], z_4 : v, sto\rangle \triangleright \langle r', \epsilon, \epsilon, sto'\rangle$$

Since $l = z_3$, $z_1$in the semantics of AWL equals z_1 in the semantics of AM, the proof is complete.

The case: [s - comp]

Using the semantics of AWL we have that $\langle S_1 S_2, sto\rangle \to sto'$ because $\langle S_1, sto\rangle \to sto''$ and $\langle S_2, sto''\rangle \to sto'$. Using $\mathcal{CS}$ we get that

$$\mathcal{CS}\,[\![S_1 S_2]\!] = \mathcal{CS}\,[\![S_1]\!] : \mathcal{CS}\,[\![S_2]\!]$$

We apply the induction hypothesis to the premises and get that

$$\langle r,\ \ \mathcal{CS}\,[\![S_1]\!]\ \ , \epsilon, sto\rangle \triangleright^* \langle r'', \epsilon, \epsilon, sto''\rangle \text{and}$$

$$\langle r'',\ \ \mathcal{CS}\,[\![S_2]\!]\ \ , \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, \epsilon, sto'\rangle$$

Since we can extend the code component we get that

$$\langle r,\ \ \mathcal{CS}\,[\![S_1]\!] :\ \mathcal{CS}\,[\![S_2]\!]\ \ , \epsilon, sto\rangle \triangleright^* \langle r'',\ \ \mathcal{CS}\,[\![S_2]\!]\ \ , \epsilon, sto''\rangle \triangleright^* \langle r', \epsilon, \epsilon, sto'\rangle$$

which completes the proof.

The case: [S-if-true]

We assume that $\langle if(be)\{S_1\}else\{S_2\}, sto\rangle \to_S sto'$ because $be \to_{be} b$, $\mathcal{B}\,[\![b]\!] = tt$ and $\langle S_1, sto\rangle \to_S sto'$.

From the implementation we get

$$\mathcal{CS}\,[\![if(be)\{S_1\}else\{S_2\}]\!] =$$

$$\mathcal{CB}\,[\![be]\!] : \textbf{JUMPF}\ n_1 : \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2$$

For boolean expressions we have that $\langle r,\ \ \mathcal{CB}\,[\![be]\!]\ \ , \epsilon\ \ , sto\rangle \triangleright \langle r',\ \ \epsilon\ \ , b, sto\rangle$

Applying this we get

$$\langle r,\ \ \mathcal{CB}\,[\![be]\!] : \textbf{JUMPF}\ n_1 : \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2\ \ , \epsilon\ \ , sto\rangle \triangleright^*$$
$$\langle r', \textbf{JUMPF}\ n_1 : \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2, b, sto\rangle$$

Using the rule for **JUMPF** n and assuming that $b = tt$ we have that

$$\langle r', \textbf{JUMPF}\ n_1 : \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2, b, sto\rangle \triangleright$$
$$\langle r'', \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2, \epsilon, sto\rangle$$

Using the rule for $\mathcal{CS}\,[\![S_1]\!]$ we get

$$\langle r'', \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2, \epsilon, sto\rangle \triangleright^*$$
$$\langle r''', \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2, \epsilon, sto'\rangle$$

Applying the rule for *JUMP* n we get

$$\langle r''', \mathbf{JUMP}\ n_2 : \mathbf{LABEL}\ n_1 : \mathcal{CS}[\![S_2]\!] : \mathbf{LABEL}\ n_2, \epsilon, sto'\rangle \triangleright$$

$$\langle r', \epsilon, \epsilon, sto'\rangle$$

Since $\langle S_1, sto\rangle \rightarrow_S sto'$ the proof is complete.

The case: [S-if-false]

This proof is analog to [S-if-true].

The case: [S-while-true]

We asume that $\langle while(be)\{S\}, sto\rangle \rightarrow sto''$ because $be \rightarrow_{be} tt$, $\langle S, sto\rangle \rightarrow sto''$ and $\langle while(be)\{S\}, sto''\rangle \rightarrow sto'$.

From our code translation functions we have that

$$\mathcal{CS}[\![while(be)\{S\}]\!] =$$

$$\mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2$$

The computation sequence of the translated code results in

$$\left\langle r_1,\ \ \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2\ \ , \epsilon, sto\right\rangle \triangleright$$

$$\left\langle r_2,\ \ \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2\ \ , \epsilon, sto\right\rangle \triangleright^*$$

$$\left\langle r_3,\ \ \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2\ \ , tt, sto\right\rangle \triangleright$$

$$\left\langle r_4,\ \ \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2\ \ , \epsilon, sto\right\rangle \triangleright$$

$$\left\langle r_5,\ \ \mathcal{CS}[\![S]\!] : \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2\ \ , \epsilon, sto\right\rangle$$

We get the last computation by using the semantics of **JUMP**. We now apply the induction hypothesis on the premises of the AWL semantics for [while-true]. So $\langle S, sto\rangle \rightarrow sto''$ and $\langle while(be)\{S\}, sto''\rangle \rightarrow sto'$ results in

$$\langle r_5, \mathcal{CS}[\![S]\!], \epsilon, sto\rangle \triangleright^* \langle r_6, \epsilon, \epsilon, sto''\rangle \qquad \text{and}$$

$$\langle r_6, \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2, \epsilon, sto''\rangle \triangleright \langle r_7, \epsilon, \epsilon, sto'\rangle$$

Since we can extend the code component we get that

$$\langle r_5,\ \ \mathcal{CS}[\![S]\!] : \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2\ \ , \epsilon, sto\rangle \triangleright^*$$

$$\langle r_6,\ \ \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2\ \ , \epsilon, sto''\rangle \triangleright^*$$

$$\langle r_7, \epsilon, \epsilon, sto'\rangle$$

This completes the proof for [S-while-true].

The case: [S-while-false]

This proof is analog to the proof of [S-while-true]

The case: [S-rulecall]

We assume $\langle r(P_A);, sto\rangle \rightarrow_S sto^4$ because

$$\langle P_F, env'_V\,[next \mapsto new\,(l)]\rangle \rightarrow env''_V$$

$$\langle P_A, env''_V\,[next \mapsto new\,(l)]\,, sto\rangle \rightarrow (env^3_V, sto')$$

$$\langle D_V, env^3_V, sto'\rangle \rightarrow_{D_V} (env^4{}_V, sto'')$$

$$\langle D_A, env^4_V, sto''\rangle \rightarrow_{D_A} (env^5_V, sto^3)$$

$$\langle S, sto^3\rangle \rightarrow_S sto^4$$

where $l = env_V\,(next)$ and $env_P\,(r) = (S, P_F, env'_V, D_V, D_A)$. From our code translation function we have that

$$\mathcal{CS}\,[\![r\,(P_A)]\!] = \mathcal{CP_A}\,[\![P_A]\!] : \textbf{CALL}\; ploc(r), parameter\ count$$

Using the semantics of **CALL** and the defined protocols we can rewrite our computation sequence to

$$\mathcal{CP_A}\,[\![P_A]\!] : \textbf{CALL}\; ploc(r), parameter\ count \triangleright =$$
$$\mathcal{CP_A}\,[\![P_A]\!] : \mathcal{CP_F}\,[\![P_F]\!] : \mathcal{CD_V}\,[\![D_V]\!] : \mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \textbf{RETURN}$$

We can now make the computation sequence

$$\langle r, \mathcal{CP_A}\,[\![P_A]\!] : \mathcal{CP_F}\,[\![P_F]\!] : \mathcal{CD_V}\,[\![D_V]\!] : \mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \textbf{RETURN}, \epsilon, sto\rangle \triangleright$$

$$\langle r', \mathcal{CS}\,[\![S]\!]\,, \epsilon, sto^3\rangle$$

$$\langle r'', \epsilon, \epsilon, sto^4\rangle$$

We make the first computation using the other proofs in this chapter. To make the last computation we apply the induction hypothesis to the premise $\langle S, sto^3\rangle \rightarrow_S sto^4$, which completes the proof.

The proof of [S-endturn] and [S-process] is analogous.

The case: [S - common memory assign]

We assume that $\langle \textbf{cmem}\, x{=}exp;, sto\rangle \rightarrow sto'$ where $sto' = sto[l \mapsto v]$, $l = env_V(x)$ and $exp \rightarrow v$.

Using $\mathcal{CS}$ we get that

$$\mathcal{CS}\,[\![cmem\ x = exp]\!] = \mathcal{CE}\,[\![exp]\!] : \textbf{SAVE}\; n\,[CM]$$

where $n = mloc(memory, x)$. From the expression lemmas we have

$$\langle r,\ \ \mathcal{CE}\,[\![exp]\!]\ \ , \epsilon, sto\rangle \triangleright^* \langle r', \epsilon, v, sto\rangle$$

and from the semantic rules of **SAVE** we get

$$\langle r', \textbf{SAVE}\; n\,[CM], v, sto\rangle \triangleright \langle r'', \epsilon, \epsilon, sto\,[(r(CM) + z]) \mapsto v\rangle$$

Using the definition of CM we see that $env_V(x) = r(CM) + z$.

The case: [S - team memory assign]

We assume that $\langle \textbf{tmem}\, x{=}exp;, sto\rangle \rightarrow sto'$ where $sto' = sto[l \mapsto v]$, $l = env_V(x)$ and $exp \rightarrow v$.

Using $\mathcal{CS}$ we get that

$$\mathcal{CS}\,[\![tmem\ x = exp]\!] = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ n\ [CTM]$$

where $n = mloc(memory, x)$. From the expression lemmas we have

$$\langle r,\ \ \mathcal{CE}\,[\![exp]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r', \epsilon, v, sto\rangle$$

and from the semantic rules of AWLAM we get

$$\langle r', \mathbf{SAVE}\ n\ [CTM], v, sto\rangle \rhd \langle r'', \epsilon, \epsilon, sto\,[(r(CTM) + z) \mapsto v]\rangle$$

Using the definition of CTM we see that $env_V(x) = r(CTM) + z$.

The case: [S - private memory assign]

We assume that $\langle \mathbf{pmem}\, x{=}exp;, sto\rangle \to sto'$ where $sto' = sto[l \mapsto v]$, $l = env_V(x)$ and $exp \to v$.

Using $\mathcal{CS}$ we get that

$$\mathcal{CS}\,[\![\mathbf{pmem}\ x = exp]\!] = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ n\ [CAM]$$

where $n = mloc(memory, x)$. From the expression lemmas we have

$$\langle r,\ \ \mathcal{CE}\,[\![exp]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r', \epsilon, v, sto\rangle$$

and from the semantic rules of AWLAM we get

$$\langle r,'\ \mathbf{SAVE}\ n\ [CAM], v, sto\rangle \rhd \langle r'', \epsilon, \epsilon, sto\,[(r(CAM) + z) \mapsto v]\rangle$$

Using the definition of CAM we see that $env_V(x) = r(CAM) + z$.

The case: [S-return]

We assume that $\langle \mathbf{return}\, exp; , sto\rangle \to sto'$ where $sto' = sto[l \mapsto v]$, $l = env_V(return)$ and $exp \to v$.

Using $\mathcal{CS}$ we get that

$$\mathcal{CS}\,[\![\mathbf{return}\ exp;]\!] = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ \mathit{0}\ \ [LS]$$

From the expression lemmas we have

$$\langle r,\ \ \mathcal{CE}\,[\![exp]\!]\ \ , \epsilon, sto\rangle \rhd^* \langle r', \epsilon, v, sto\rangle$$

and from the semantic rules of AWLAM we get

$$\langle r', \mathbf{SAVE}\ \mathit{0}\ [LS], v, sto\rangle \rhd \langle r'', \epsilon, \epsilon, sto\,[r(LS) \mapsto v]\rangle$$

Using the definition of LS and the defined protocols, we see that $env_V(return) = r(LS)$.

The case: [S-skip]

We assume that $\langle \mathbf{skip};, sto\rangle \to sto$. Using $\mathcal{CS}$ we get that

$$\mathcal{CS}\,[\![skip]\!]\,p = \mathbf{NOOP}$$

From the rule for **NOOP** we have that $\langle r, \mathbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r', \epsilon, \epsilon, sto\rangle$, which completes the proof.

The case: [S-setProperty]

We assume that $\langle \mathbf{setProperty}(ae,\, exp);, sto\rangle \rightarrow sto'$ where $sto' = sto[l \mapsto v]$, $exp \rightarrow v_1$, $ae \rightarrow z_1$ and $z_1 = l$.

We have that

$$\mathcal{CS}\,[\![\mathbf{setProperty}(ae, exp);]\!] = \mathcal{CE}\,[\![exp]\!] : \mathcal{CA}\,[\![ae]\!] : \mathbf{SAVES}\;[SD]$$

Applying $\mathcal{CE}$ and $\mathcal{CA}$ we get

$$\langle r, \mathcal{CE}\,[\![exp]\!] : \mathcal{CA}\,[\![ae]\!], \epsilon, sto\rangle \triangleright^{*} \langle r'', \mathcal{CA}\,[\![ae]\!], v_2, sto\rangle \triangleright^{*} \langle r''', \epsilon, z_2 : v_2, sto\rangle$$

Using the semantics of $SAVES$ we get

$$\langle r''', \mathbf{SAVES}\;[SD]\,, z_2 : v_2, sto\rangle \triangleright \langle r', \epsilon, \epsilon, sto'\rangle$$

Since $l = z_1 = z_2$ and $v_1 = v_2$ the proof is complete.

This concludes the proof of lemma A.4.2.

Lemma A.4.3 *For every command S of AWL and stores sto and sto', we have that*

$$\textit{if}\ \langle r, \mathcal{CS}\,[\![S]\!]\,, \epsilon, sto\rangle \triangleright^{k} \langle r', \epsilon, e, sto'\rangle\ \textit{then}\ \langle S, sto\rangle \rightarrow sto'$$

So if the execution of the code for S from a storage s terminates, then the AWL semantics of S from s will terminate in a state being equal to the storage of the terminal configuration.

Proof: We will prove lemma A.4.3 by induction on the length k of the computation sequence on AM. If $k = 0$ then the result holds because $\mathcal{CS}[\![S]\!] = \epsilon$ is impossible. So we assume that it holds for $k \leq k_0$ and will prove that it holds for $k = k_0 + 1$. We make a case study on the command S.

The case: $x = exp$;

We have that $\mathcal{CS}\,[\![x = exp;]\!] = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\; n\,[LS]$, so we assume that

$$\langle r, \mathcal{CE}\,[\![exp]\!] : SAVE\, n\; [LS], \epsilon, sto\rangle \triangleright^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

Since we can split the instruction sequence into two we have that

$$\langle r, \mathcal{CE}\,[\![exp]\!]\,, \epsilon, sto\rangle \triangleright^{k_1} \langle r'', \epsilon, e'', sto''\rangle \text{and}$$

$$\langle r, SAVE\, n\; [LS], e'', sto''\rangle \triangleright^{k_2} \langle r', \epsilon, e, sto'\rangle$$

where $k_1 + k_2 = k_0 + 1$. From the expression lemmas we get that $sto'' = sto$ and $e'' = v$ where $exp \to v$. Using the semantics of **SAVE** we see that $sto' = sto[(r(LS) + z) \mapsto v]$. It follows from [s - assign] that $\langle \mathbf{x} = \mathbf{exp};, sto\rangle \to sto'$, which completes the proof.

The case: $x\,[ae] = exp;$

We have that

$$\mathcal{CS}\,[\![x\,[ae] = exp;]\!]p = \mathcal{CE}\,[\![exp]\!] : \mathcal{CA}\,[\![ae]\!] : \mathbf{PUSH}\ n : \mathbf{ADD} : \mathbf{SAVES}\ [LS]$$

We can then make the computation sequence

$$\langle r, \mathcal{CE}\,[\![exp]\!] : \mathcal{CA}\,[\![ae]\!] : \mathbf{PUSH}\ n : \mathbf{ADD} : \mathbf{SAVES}\ [LS], \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

Since we can split the code component up we get

$$\begin{array}{lll}
\langle r, \mathcal{CE}\,[\![exp]\!], \epsilon, sto\rangle & \rhd^{k_1} & \langle r^5, \epsilon, e^4, sto^5\rangle \\
\langle r^5, \mathcal{CA}\,[\![ae]\!], e^4, sto^5\rangle & \rhd^{k_2} & \langle r^4, \epsilon, e^3, sto^4\rangle \\
\langle r^4, \mathbf{PUSH}\ n, e^3, sto^4\rangle & \rhd^{k_3} & \langle r^3, \epsilon, e^2, sto^3\rangle \\
\langle r^3, \mathbf{ADD}, e^2, sto^3\rangle & \rhd^{k_4} & \langle r^2, \epsilon, e^1, sto^2\rangle \\
\langle r^2, \mathbf{SAVES}\ [LS], e^1, sto^2\rangle & \rhd^{k_5} & \langle r', \epsilon, e, sto'\rangle
\end{array}$$

where $k_1 + k_2 + k_3 + k_4 + k_5 = k_0 + 1$

Since $\mathcal{CE}\,[\![exp]\!]$, $\mathcal{CA}\,[\![ae]\!]$, **PUSH** and **ADD** do not change the storage, we have that $sto = sto^5 = sto^4 = sto^3 = sto^2$. We also have that $e^4 = v$, $e^3 = z_1 : v$, $e^3 = z_2 : z_1 : v$, $e^1 = z_3 : v$ and $e = \epsilon$.

Since we have that $sto' = sto[(l + z_1) \mapsto v]$ where $env_V(x) = (type, l, z_3)$ and $0 \leq z_1 < z_3$ this completes the proof.

The case: $S_1 S_2$

We have that $\mathcal{CS}\,[\![S_1\,S_2]\!] = \mathcal{CS}\,[\![S_1]\!] : \mathcal{CS}\,[\![S_2]\!]$, so we assume that

$$\langle r, \mathcal{CS}\,[\![S_1]\!] : \mathcal{CS}\,[\![S_2]\!], \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

Since we can split the instruction sequence into two we have that

$$\langle r, \mathcal{CS}\,[\![S_1]\!], \epsilon, sto\rangle \rhd^{k_1} \langle r', \epsilon, e', sto'\rangle \text{ and}$$

$$\langle r'', \mathcal{CS}\,[\![S_2]\!], e', sto''\rangle \rhd^{k_2} \langle r', \epsilon, e, sto'\rangle$$

where $k_1 + k_2 = k_0 + 1$, $e' = \epsilon$ and $e = \epsilon$.

We can now apply the induction hypothesis to $\langle r, \mathcal{CS}\,[\![S_1]\!], \epsilon, sto\rangle \rhd^{k_1} \langle r', \epsilon, e', sto'\rangle$ because $k_1 \leq k_0$

$$\langle S_1, sto\rangle \to_S sto''$$

Because we have $\langle r'', \mathcal{CS}\,[\![S_2]\!], e', sto''\rangle \rhd^{k_2} \langle r', \epsilon, e, sto'\rangle$ and $k_2 \leq k_0$ we can now apply the induction hypothesis one more time and get.

$$\langle S_2, sto'' \rangle \rightarrow_S sto'$$

This gives us $\langle S_1 S_2, sto\rangle \rightarrow_S sto'$ as required. The proof is now complete.

The case: $if(be)\{S_1\}else\{S_2\}\ true$

We have that

$$\mathcal{CS}\,[\![if(b)\{S_1\}else\{S_2\}]\!] =$$
$$\mathcal{CB}\,[be] : \textbf{JUMPF}\ n_1 : \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2$$

We assume that

$$\langle r, \mathcal{CB}\,[be] : \textbf{JUMPF}\ n_1 : \mathcal{CS}\,[\![S_1]\!] : \textbf{JUMP}\ n_2 : \textbf{LABEL}\ n_1 : \mathcal{CS}\,[\![S_2]\!] : \textbf{LABEL}\ n_2, \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

Since we can split up the code component we get

$$\begin{array}{lll}
\langle r, \mathcal{CB}\,[\![be]\!], \epsilon, sto\rangle & \rhd^{k_1} & \langle r'''', \epsilon, e''', sto''''\rangle \\
\langle r'''', \textbf{JUMPF}\ n_1, e''', sto''''\rangle & \rhd^{k_2} & \langle r''', \epsilon, e'', sto'''\rangle \\
\langle r''', \mathcal{CS}\,[\![S_1]\!], e'', sto'''\rangle & \rhd^{k_3} & \langle r'', \epsilon, e', sto''\rangle \\
\langle r'', \textbf{JUMP}\ n_2, e', sto''\rangle & \rhd^{k_4} & \langle r', \epsilon, e, sto'\rangle
\end{array}$$

where $k_1 + k_2 + k_3 + k_4 = k_0 + 1$ and $k_2, k_4 = 1$.

Since $\mathcal{CB}\,[\![be]\!]$ and **JUMPF** does not change the storage, we have that $sto'''' = sto'''$ and $sto'' = sto'$. Likewise $\mathcal{CS}\,[\![S_1]\!]$ and **JUMP** does not change the evaluation stack so we have that $e'' = e' = e = \epsilon$. We assume that $e''' = tt$.

Since $k_3 \le k_0$ we can apply the induction hypothesis to this computation and then we have that $\langle S_1, sto\rangle \rightarrow sto'$

The rule [S-if-true] gives the required $\langle if(be)\{S_1\}else\{S_2\}, sto\rangle \rightarrow sto'$. The proof of $if(be)\{S_1\}else\{S_2\}false$ is analogous.

The case: $while(be)\{S\}\ true$

The code for the while loop is

$$\mathcal{CS}[\![while(be)\{S\}]\!] = \textbf{LABEL}\ n_1 : \mathcal{CB}\,[\![be]\!] : \textbf{JUMPF}\ n_2 : \mathcal{CS}\,[\![S]\!] : \textbf{JUMP}\ n_1 : \textbf{LABEL}\ n_2$$

and we therefore assume that

$$\langle r, \textbf{LABEL}\ n_1 : \mathcal{CB}\,[\![be]\!] : \textbf{JUMPF}\ n_2 : \mathcal{CS}\,[\![S]\!] : \textbf{JUMP}\ n_1 : \textbf{LABEL}\ n_2, \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, \epsilon, sto'\rangle$$

Using the definition of **JUMP** we can rewrite the computation sequence in the following maner:

$\langle r, \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2, \epsilon, sto\rangle \triangleright$

$\langle r'', \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2, \epsilon, sto\rangle \triangleright$

$\langle r''', \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2, tt, sto\rangle \triangleright$

$\langle r'''', \mathcal{CS}[\![S]\!] : \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2, \epsilon, sto\rangle \triangleright^{k_0-2}$

$\langle r', \epsilon, \epsilon, sto'\rangle$

We can now split up our code component, and we get

$$\langle r'''', \mathcal{CS}[\![S]\!], \epsilon, sto\rangle \triangleright^{k_1} \langle r''''', \epsilon, \epsilon, sto''\rangle \text{ and} \quad (1)$$

$$\langle r''''', \mathbf{LABEL}\ n_1 : \mathcal{CB}[\![be]\!] : \mathbf{JUMPF}\ n_2 : \mathcal{CS}[\![S]\!] : \mathbf{JUMP}\ n_1 : \mathbf{LABEL}\ n_2, \epsilon, sto''\rangle \triangleright^{k_2} \langle r', \epsilon, \epsilon, sto'\rangle \quad (2)$$

where $k_1 + k_2 = k_0 - 2$. Since $k_1 \leq k_0$ we can apply the induction hypothesis to the computation sequence (1). We therefore get that $\langle S, sto\rangle \rightarrow sto'$. And since $k_2 \leq k_0$we can also apply the induction hypothesis to the computation sequence (2) and we get that $\langle while(be)\{S\}, sto''\rangle \rightarrow sto'$. Using the rule [S-while-true] we get $\langle while(be)\{S\}, sto\rangle \rightarrow sto'$ as required.

The proof of the case $while(be)\{S\}\ false$ is analogous.

The case: *skip*;

We have that $\mathcal{CS}[\![\mathbf{skip};]\!] = \mathbf{NOOP}$. That gives us the configuration

$$\langle r, \mathbf{NOOP}, \epsilon, sto\rangle \triangleright \langle r', \epsilon, e, sto'\rangle$$

Since $e = \epsilon$ and $sto' = sto$ and $\langle \mathbf{skip};, sto\rangle \rightarrow_S sto$ the proof is complete.

The case: $setProperty(ae, exp)$;

We have that

$$\mathcal{CS}[\mathbf{setProperty}(ae, exp)] = \mathcal{CE}[\![exp]\!] : \mathcal{CA}[\![ae]\!] : \mathbf{SAVES}\ [SD]$$

This give us the configuration

$$\langle r, \mathcal{CE}[\![exp]\!] : \mathcal{CA}[\![ae]\!] : \mathbf{SAVES}\ [SD], \epsilon, sto\rangle \triangleright^{k_0+1} \langle r^1, \epsilon, \epsilon, sto^1\rangle$$

We can split this into

$$\begin{array}{lll} \langle r, \mathcal{CE}[\![exp]\!], \epsilon, sto\rangle & \triangleright^{k_1} & \langle r^2, \epsilon, e^1, sto^2\rangle \\ \langle r^2, \mathcal{CA}[\![ae]\!], e^1, sto^2\rangle & \triangleright^{k_2} & \langle r^3, \epsilon, e^2, sto^3\rangle \\ \langle r^3, \mathbf{SAVES}\ [SD], e^2, sto^3\rangle & \triangleright^{k_3} & \left\langle r', \epsilon, e, sto'\right\rangle \end{array}$$

We have that $k_0 + 1 = k_1 + k_2 + k_3$, $e' = v_1$, $e^2 = z_1 : v_1$ and $e = \epsilon$.

From [S-getProperty] we have that $ae \rightarrow z_2 = l$ and $exp \rightarrow v_2$. Because $z_1 = z_2$ and $v_1 = v_2$ this completes the proof.

The case: $cmem\ x = exp$;

We have that

$$\mathcal{CS}\,[\![\mathbf{cmem}\ x{=}exp;]\!] = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ n\ [CM]$$

so we assume that

$$\langle r, \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ n\ [CM], \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

Since we can split the instruction sequence into two we have

$$\langle r, \mathcal{CE}\,[\![exp]\!], \epsilon, sto\rangle \rhd^{k_1} \langle r'', \epsilon, e'', sto''\rangle \text{ and}$$

$$\langle r, \mathbf{SAVE}\ n\ [CM], e'', sto''\rangle \rhd^{k_2} \langle r', \epsilon, e, sto'\rangle$$

where $k_1 + k_2 = k_0 + 1$. From the expression lemmas we get that $sto'' = sto$ and $e'' = v$ where $exp \to v$. Using the semantics of **SAVE** we see that $sto' = sto[(r(CM) + z) \mapsto v]$. It follows from [S-common memory assign] and the definition of CM that $\langle \mathbf{cmem}\ x{=}exp;, sto\rangle \to sto'$, which completes the proof.

The proofs of *tmem* and *pmem* are analogous.

The case: return *exp*;

We have that

$$\mathcal{CS}\,[\![\mathbf{return}\ exp;]\!] = \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ 0\ [LS]$$

so we assume that

$$\langle r, \mathcal{CE}\,[\![exp]\!] : \mathbf{SAVE}\ 0\ [LS], \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

Since we can split the instruction sequence into two we have

$$\langle r, \mathcal{CE}\,[\![exp]\!], \epsilon, sto\rangle \rhd^{k_1} \langle r'', \epsilon, e'', sto''\rangle \text{ and}$$

$$\langle r, \mathbf{SAVE}\ 0\ [LS], e'', sto''\rangle \rhd^{k_2} \langle r', \epsilon, e, sto'\rangle$$

where $k_1 + k_2 = k_0 + 1$. From the expression lemmas we get that $sto'' = sto$ and $e'' = v$ where $exp \to v$. Using the semantics of **SAVE** we see that $sto' = sto[r(LS) \mapsto v]$. It follows from [S-return] and the definition of **LS** that $\langle \mathbf{return}\ exp;, sto\rangle \to sto'$, which completes the proof.

The case: $r\,(P_A)\,;$

We have that

$$\mathcal{CS}[\![r\,(P_A)\,;]\!] = \mathcal{CP}_{\mathcal{A}}\,[\![P_A]\!] : \mathbf{CALL}\ n_1, n_2$$

where $n_1 = ploc(r)$ and $n_2 = parameter\ count$, so we assume that

$$\langle r, \mathcal{CP}_{\mathcal{A}}\,[\![P_A]\!] : \mathbf{CALL}\ n_1, n_2, \epsilon, sto\rangle \rhd^{k_0+1} \langle r', \epsilon, e, sto'\rangle$$

We can split up the code component, so there must be a configuration on the form $\langle r'', \epsilon, e'', sto''\rangle$ such that

$$\langle r, \mathcal{CP_A}\,[\![P_A]\!], \epsilon, sto\rangle \rhd^{k_1} \langle r'', \epsilon, e'', sto''\rangle \quad \text{and}$$

$$\langle r'', \mathbf{CALL}\ n_1, n_2, e'', sto''\rangle \rhd^{k_2} \langle r', \epsilon, e, sto'\rangle$$

where $k_1 + k_2 = k_0 + 1$. Using lemma A.2.2 we see that e'' contains the actual parameters, if there are any. Using the semantics of **CALL** we can rewrite the last computation.

$$\langle r'', \mathcal{CP_F}\,[\![P_F]\!] : \mathcal{CD_V}\,[\![D_V]\!] : \mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \mathbf{RETURN}, e'', sto''\rangle \rhd^{k_2}$$
$$\langle r', \epsilon, e, sto'\rangle$$

This sequence can also be split up, so there must be a configuration on the form $\langle r''', \epsilon, e''', sto'''\rangle$, such that

$$\langle r'', \mathcal{CP_F}\,[\![P_F]\!], e'', sto''\rangle \rhd^{k_3} \langle r''', \epsilon, e''', sto'''\rangle \text{ and}$$

$$\langle r''', \mathcal{CD_V}\,[\![D_V]\!] : \mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \mathbf{RETURN}, e''', sto'''\rangle \rhd^{k_4} \langle r', \epsilon, e, sto'\rangle$$

where $k_3 + k_4 = k_2$. Using lemma A.2.1 we see that $e''' = \epsilon$ and that $sto''' = sto''$. Again we can split up the code component, so there must be a configuration on the form $\langle r^4, \epsilon, e^4, sto^4\rangle$ such that

$$\langle r''', \mathcal{CD_V}\,[\![D_V]\!], e''', sto'''\rangle \rhd^{k_5} \langle r^4, \epsilon, e^4, sto^4\rangle \text{ and}$$

$$\langle r^4, \mathcal{CD_A}\,[\![D_A]\!] : \mathcal{CS}\,[\![S]\!] : \mathbf{RETURN}, e^4, sto^4\rangle \rhd^{k_6} \langle r', \epsilon, e, sto'\rangle$$

where $k_5 + k_6 = k_3$. Using lemma A.1.5 we see that $e^4 = e''' = \epsilon$. Again we can split up the code component, so there must be a configuration on the form $\langle r^5, \epsilon, e^5, sto^5\rangle$ such that

$$\langle r^4, \mathcal{CD_A}\,[\![D_A]\!], e^4, sto^4\rangle \rhd^{k_7} \langle r^5, \epsilon, e^5, sto^5\rangle \text{ and}$$

$$\langle r^5, \mathcal{CS}\,[\![S]\!] : \mathbf{RETURN}, e^5, sto^5\rangle \rhd^{k_8} \langle r', \epsilon, e, sto'\rangle$$

where $k_7 + k_8 = k_6$. Using lemma A.1.2 we see that $e^5 = e^4 = \epsilon$. Once again we split up the code component, so there must be a configuration on the form $\langle r^6, \epsilon, e^6, sto^6\rangle$ such that

$$\langle r^5, \mathcal{CS}\,[\![S]\!] :, e^5, sto^5\rangle \rhd^{k_9} \langle r^6, \epsilon, e^6, sto^6\rangle \text{ and}$$

$$\langle r^6, \mathbf{RETURN}, e^6, sto^6\rangle \rhd^{k_{10}} \langle r', \epsilon, e, sto'\rangle$$

where $k_9 + k_{10} = k_8$. We can now apply the induction hypothesis to S, which gives

$$\langle S, sto^5\rangle \to sto^6 \text{ and } e^6 = \epsilon$$

Using the rule [S-rulecall] we get that $\langle r(P_A);, sto\rangle \to_S sto'$, which completes the proof.

Appendix B

SableCC Generated File

The following is the file that SableCC uses to generate our scanner and parser along with the tree, and the included tree walkers.

```
Package awl.compiler.parser;
Helpers
letter = ( ['a'..'z'] | ['A'..'Z'] );
digit = ['0'..'9'];
true = 'true'; false = 'false';
left = 'left'; right = 'right';
up = 'up'; down = 'down';
ht = 0x0009; lf = 0x000a;
ff = 0x000c; cr = 0x000d; sp = ' ';
Tokens
t_world = 'world';
t_main = 'main';
t_rule = 'rule';
t_turn = 'turn';
t_anttype= 'anttype';
t_endturn = 'endturn';
t_process = 'process';
t_getproperty = 'getProperty';
t_setproperty = 'setProperty';
t_createteam = 'createTeam';
t_createant = 'createAnt';
t_random = 'random';
t_return = 'return';
t_skip = 'skip';
t_if = 'if';
t_else = 'else';
t_while = 'while';
t_common = 'common';
t_private = 'private';
t_teambrain = 'teambrain';
t_cmem = 'cmem';
t_pmem = 'pmem';
t_tmem = 'tmem';
t_var = 'var';
t_array = 'array';
```

```
t_eof = 'eof';
t_or = 'or';
t_and = 'and';
t_integer= 'integer';
t_boolean = 'boolean';
t_direction = 'direction';
t_integer_literal = digit*;
t_boolean_literal = ( true | false );
t_direction_literal = (left | right | up | down);
t_identifier = letter (letter | digit)*;
t_lpar = '(';
t_rpar = ')';
t_lbrace = '{';
t_rbrace = '}';
t_lbracket = '[';
t_rbracket = ']';
t_semicolon = ';';
t_colon = ':';
t_comma = ',';
t_dot = '.';
t_assign = '=';
t_bang = '!';
t_gt = '>';
t_lt = '<';
t_eq = '==';
t_ne = '!=';
t_le = '<=';
t_ge = '>=';
t_plus = '+';
t_minus = '-';
t_star = '*';
t_slash = '/';
t_newline = cr | lf | cr lf;
t_whitespace = (sp | ht | ff)*;
t_comment = '#' (digit | letter | ' ')*;
Ignored Tokens
t_newline, t_whitespace, t_comment;
Productions
/* Program */
program = world;
main = t_main t_lbrace team_declaration*
variable_init* array_init* commands t_rbrace;
world = t_world t_lpar [size]:t_integer_literal
[comma1]:t_comma [ants]:t_integer_literal
[comma2]:t_comma [foods]:t_integer_literal t_rpar t_lbrace
common_decl* teambrain_decl* private_decl*
ntb_declaration* tb_declaration* ant_type_declaration*
main
t_rbrace;
/* Commands*/
commands = command*;
command =
{assign} t_identifier t_assign expression t_semicolon |
{rulecall} t_identifier t_lpar actual_parm_list
t_rpar t_semicolon |
```

```
{arrayassign} t_identifier t_lbracket[index]:expression
t_rbracket t_assign [value]:expression t_semicolon |
{if} t_if t_lpar [condition]:expression t_rpar
[lbrace1]:t_lbrace [com]:commands [rbrace1]:t_rbrace
t_else [lbrace2]:t_lbrace
[else_com]:commands [rbrace2]:t_rbrace |
{while} t_while t_lpar expression
t_rpar t_lbrace commands t_rbrace |
{endturn} t_endturn t_identifier
t_lpar actual_parm_list t_rpar t_semicolon |
{return} t_return expression t_semicolon |
{skip} t_skip t_semicolon |
{cmem} t_cmem t_identifier
t_assign expression t_semicolon |
{tmem} t_tmem t_identifier
t_assign expression t_semicolon |
{pmem} t_pmem t_identifier t_assign expression
t_semicolon |
{process} t_process t_lpar [team]:expression
[comma1]:t_comma [ant]:expression [comma2]:t_comma
t_identifier t_rpar t_semicolon |
{setproperty} t_setproperty t_lpar
[index]:expression t_comma [value]:expression
t_rpar t_semicolon |
{createant} t_createant t_lpar
expression t_rpar t_semicolon;
/* Memory */
common_decl = t_common variable_init;
teambrain_decl = t_teambrain
variable_declaration t_semicolon;
private_decl = t_private
variable_declaration t_semicolon;
/* Declarations*/
ntb_declaration =
{noreturn} ntb_declaration_noreturn |
{return} ntb_declaration_return;
ntb_declaration_noreturn = t_rule t_identifier
t_lpar formal_parm_list t_rpar t_lbrace
variable_init* array_init* commands t_rbrace;
ntb_declaration_return = t_rule t_identifier
t_lpar formal_parm_list t_rpar t_colon
simple_type t_lbrace variable_init*
array_init* commands t_rbrace;
tb_declaration = t_turn t_identifier t_lpar
formal_parm_list t_rpar t_lbrace variable_init*
array_init* commands t_rbrace;
return_type = t_colon simple_type;
formal_parm = variable_declaration t_semicolon;
formal_parm_list = formal_parm*;
actual_parm = expression t_semicolon;
actual_parm_list = actual_parm*;
ant_type_declaration = t_anttype t_identifier
t_lbrace variable_init* array_init* commands t_rbrace;
team_declaration = t_createteam t_lpar
t_identifier t_rpar t_semicolon;
```

```
/* Variable Declarations*/
variable_init = variable_declaration
t_assign expression t_semicolon;
array_init = array_declaration t_assign expression
t_semicolon;
variable_declaration = t_var t_identifier
t_colon simple_type;
array_declaration = t_array t_identifier
t_lbracket t_integer_literal t_rbracket t_colon simple_type;
simple_type =
{integer} t_integer |
{boolean} t_boolean |
{direction} t_direction;
/* Expressions*/
primary_expression =
{par} t_lpar expression t_rpar |
{constant} literal |
{identifier} t_identifier |
{array} t_identifier t_lbracket expression t_rbracket |
{function} t_identifier t_lpar actual_parm_list t_rpar |
{cmem_identifier} t_cmem t_identifier |
{tmem_identifier} t_tmem t_identifier |
{pmem_identifier} t_pmem t_identifier |
{getproperty} t_getproperty t_lpar expression t_rpar |
{random} t_random t_lpar expression t_rpar;
expression = or_expression;
or_expression =
{or} and_expression t_or or_expression |
{bubble} and_expression;
and_expression =
{and} eq_expression t_and and_expression |
{bubble} eq_expression;
eq_expression =
{equals} rel_expression t_eq eq_expression |
{notequals} rel_expression t_ne eq_expression |
{bubble} rel_expression;
rel_expression =
{greater} add_expression t_gt rel_expression |
{lower} add_expression t_lt rel_expression |
{greaterequals} add_expression t_ge rel_expression |
{lowerequals} add_expression t_le rel_expression |
{bubble} add_expression;
add_expression =
{plus} mult_expression t_plus add_expression |
{minus} mult_expression t_minus add_expression |
{bubble} mult_expression;
mult_expression =
{mult} unary_expression t_star mult_expression |
{div} unary_expression t_slash mult_expression |
{bubble} unary_expression;
unary_expression =
{minus} t_minus primary_expression |
{bang} t_bang primary_expression |
{bubble} primary_expression;
/* Literal*/
```

```
literal =
{boolean} t_boolean_literal |
{integer} t_integer_literal |
{direction} t_direction_literal;
```