ArchWare ADL – Release 1

A User Reference Manual

(Project Deliverable D4.3)

Authors: Dharini Balasubramaniam – USTAN
          Ron Morrison – USTAN
          Graham Kirby – USTAN
          Katherine Mickan – USTAN

Editor:

Document ID: D4.3

Date: August 2003

Version: V1.1

Status: Issued

Reviewed by:

Distribution: Public

Abstract: This documentation of deliverable D4.3 is intended to serve as a user reference manual for the first release of ArchWare ADL.

Copyright © 2002 by the ArchWare Consortium
CPR – Consorzio Pisa Ricerche – Italy
InterUnec – Université de Savoie – France
Victoria University of Manchester – UK
ENGINEERING – Ingegneria Informatica S.p.A. – Italy
INRIA – Institut National de Recherche en Informatique et Automatique – France
THESAME – Mecatronique et Management – France
University Court of the University of St. Andrews – UK
All Rights Reserved
Acknowledgments

This report is a result of many useful discussions and feedback from our ArchWare partners. The authors would particularly like to thank Prof Flavio Oquendo and Dr Sorana Cimpan at InterUnec and Prof Brian Warboys, Dr Bob Snowdon, Dr Ian Robertson, Dr Mark Greenwood and Mr Wykeen Seet at VUM for their contributions to this report. The structure of this report is based on the existing ProcessBase Language Reference Manual [1].

Caveat

This ArchWare ADL – Release 1 is subject to evolutions based on feedbacks from its usage within the project. Only the most recent version of this document should be used. New releases of this document will be advertised to Project Participants and be made available on the project cooperative work support web site.

Disclaimer

The authors have utilized their professional expertise in preparing this report. However, neither the ArchWare Consortium nor the authors make any representation or warranties with respect to the contents of this report. Rather, this report is provided on an AS IS basis, without warranty, express or implied, INCLUDING A FULL DISCLAIMER OF THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
# Table of Contents

1 INTRODUCTION 7

1.1 Overview 7

1.2 Document Structure 7

2 THE ARCHADL SYSTEM 8

2.1 Installation of ProcessBase System 8

3 UNIVERSE OF DISCOURSE 9

3.1 Universe of Discourse 9

3.2 The Type Algebra 10

3.3 Type Equivalence 10

3.4 First Class Citizenship 11

4 LANGUAGE DEFINITION RULES 12

4.1 Context-Free Syntax 12

4.2 Type Rules 12

5 DECLARATIONS 14

5.1 Identifiers 14

5.2 Declaration of Value Identifiers 14

5.3 Declaration of Types 15

6 LITERALS 16

6.1 Integer Literals 16

6.2 Real Literals 16

6.3 Boolean Literals 17

6.4 String Literals 17
6.5 View Literals 17
6.6 Sequence Literals 18
6.7 Connection Literals 18
6.8 Behaviour Literal 18
6.9 Abstraction Literals 18
6.10 Function Literals 19

7 EXPRESSIONS AND OPERATORS 20
7.1 Evaluation Order 20
7.2 Parentheses 20
7.3 Boolean Expressions 20
7.4 Comparison Operators 21
7.5 Arithmetic Expressions 23
7.6 Arithmetic Precedence Rules 24
7.7 String Expressions 24
7.8 Precedence Table 25

8 CLAUSES 26
8.1 If Clause 26
8.2 Prefix Clauses 26
8.3 Replicate Clause 27
8.4 Choose Clause 27
8.5 Compose Clause 28
8.6 Decompose Clause 29
8.7 Project Clause 30
8.8 Assignment Clause 30
8.9 Iterate Clause 30

9 ANY 31
9.1 Injection into Type any 31
9.2 Projection from Type any 31
9.3 Equality and Equivalence 32

10 VIEW 33
10.1 Creation of View 33
10.2 Dereferencing A View 33
10.3 Equality and Equivalence 34

11 LOCATION 35
11.1 Creation and Dereference 35
11.2 Assignment 35
11.3 Equality and Equivalence 35

12 FUNCTION 36
12.1 Creation 36
12.2 Application 36
12.3 Equality and Equivalence 37

13 SEQUENCE 38
13.1 Creation of Sequences 38
13.2 Indexing and Iterating 39
13.3 Adding and Removing Elements 40
13.4 Equality and Equivalence 40

14 CONNECTION 41
14.1 Creating Connections 41
14.2 Communicating via Connections 41
14.3 Equality and Equivalence 41

15 BEHAVIOUR 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1</td>
<td>Creating Behaviours</td>
<td>42</td>
</tr>
<tr>
<td>15.2</td>
<td>Querying Behaviours</td>
<td>42</td>
</tr>
<tr>
<td>15.3</td>
<td>Equality and Equivalence</td>
<td>43</td>
</tr>
<tr>
<td>16</td>
<td>ABSTRACTION</td>
<td>44</td>
</tr>
<tr>
<td>16.1</td>
<td>Abstraction Application</td>
<td>44</td>
</tr>
<tr>
<td>16.2</td>
<td>Partial Application</td>
<td>44</td>
</tr>
<tr>
<td>16.3</td>
<td>Equality and Equivalence</td>
<td>45</td>
</tr>
<tr>
<td>17</td>
<td>AN EXAMPLE: A CLIENT - SERVER SYSTEM</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>APPENDIX A: CONTEXT-FREE SYNTAX DEFINITION</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>APPENDIX B: TYPE RULES</td>
<td>57</td>
</tr>
</tbody>
</table>
**Document History and Evolutions**

### Changes Log

<table>
<thead>
<tr>
<th>Documen t Version</th>
<th>Ch. Request Origin</th>
<th>Changes and Reasons for Change</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Original</td>
<td></td>
<td>DB et al</td>
</tr>
<tr>
<td>1.1</td>
<td>Updated following discussions with Annecy and VUM at St Andrews on compose, decompose and functions</td>
<td>DB et al</td>
<td></td>
</tr>
</tbody>
</table>

### Open Issues/To Do Items

<table>
<thead>
<tr>
<th>Issue Number</th>
<th>Date Logged</th>
<th>Description of Open Issue/ To Do Item</th>
<th>Priority</th>
<th>Status</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 Overview

ArchWare ADL is a programming language based on the $\lambda$-calculus proposed by Milner [2]. The abstract and concrete syntax definitions of the language along with some examples are described in Deliverables D1.1 [3] and D1.2 [4] respectively. These documents provide the rationale behind the design of the language.

This document is released as part of Deliverable D4.3 (the implementation of the ArchWare ADL reference model) and is designed to serve as a user reference manual for release 1 of the language. Release 1 is a subset of the ArchADL containing the following features: base types and operations, some constructed types (view, location, function, sequence, connection, abstraction) and operations, $\lambda$-calculus constructs and decomposition. Examples are provided where appropriate to illustrate the user of ArchADL.

1.2 Document Structure

The structure of the report is as follows:

- Chapter 2 describes the implementation details of the ArchADL system
- Chapter 3 defines the universe of discourse of ArchADL (Release1)
- Chapter 4 describes the notations used to define the language
- Chapters 5 to 16 provide details of various constructs of ArchADL
- Chapter 17 provides a non-trivial example coded in ArchADL
- Appendices A and B contain the context free definition and the type rules for Release 1 of the language
2 THE ARCHADL SYSTEM

The ArchADL is implemented by using ProcessBase language as the virtual machine (ArchVM). All software for the current implementation of ArchADL (Release 1) can be found in the ArchWare software repository under AW/VM.

The current ArchADL compiler is written in ProcessBase. The following ProcessBase libraries are needed to compile the ADL compiler: safeOpLib, ioLib, arithLib, stringLib, psLib, stdfunLib, sysclib, argvLib and symbolLib. A pre-compiled version (adlc.out) is provided which can be run as a ProcessBase executable.

Assuming pbc executes the ProcessBase compiler, pbr executes the ProcessBase interpreter and adlc executes the ADL compiler, the following sequence of commands are required to execute an ADL source program called test:

```
adlc test
pbc test.pb
pbr test.pb.out
```

2.1 Installation of ProcessBase System

All relevant ProcessBase software is also available in the ArchWare repository under AW/VM. The following steps are needed to set up the ProcessBase from a pb.tar.gz archive.

Extract the archive. This results in the following directories: bin (binary), store, bstore, int (interpreter), lib (library) and compiler. The source of all the standard libraries for ProcessBase can be found in lib.

Set the architecture of the machine being used.

Set the following environment variables: PBASESTORE, PBASEHEAP.

The bin directory contains executable scripts for pbc (compiler), pbr (interpreter) and mkstore (creating store). pbc contains a path to latest in the compiler directory. latest is a symbolic link to the latest version of the compiler executable (currently pbc.out). Ensure that this bin directory in your path.

mkstore can be used as:

```
mkstore <store file> <store size> <shadow store size>
```
3 Universe of Discourse

The ADL type system is based on the notion of types as a set structure imposed over the value space. Membership of the type sets is defined in terms of common attributes possessed by values, such as the operations defined over them. These sets or types partition the value space. The sets may be predefined, like integer, or they may be formed by using one of the predefined type constructors, like view.

The constructors obey the Principle of Data Type Completeness [5, 6]. That is, where a type may be used in a constructor, any type is legal without exception. This has two benefits. Firstly, since all the rules are very general and without exceptions, a very rich type system may be described using a small number of defining rules. This reduces the complexity of the defining rules. The second benefit is that the type constructors are as powerful as is possible since there are no restrictions on their domain.

3.1 Universe of Discourse

The following base types are defined in ArchADL:

1. The scalar data types are integer, real, and boolean.
2. Type string is the type of a character string; this type embraces the empty string and single characters.
3. Type any is an infinite union type; values of this type consist of a value of any type together with a representation of that type.
4. Type behaviour is the type of an executing process in the ADL.

The following type constructors are defined in ArchWare ADL:

5. For any type T, location [T] is the type of a location that contains a value of type T.
6. For any type T, sequence[T] is the type of a sequence with elements of type t.
7. For identifiers I1,..., In and types t1,..., tn, view[I1: t1,..., In: tn] is the type of a view with fields Ii and corresponding types tni for i = 1..n and n ≥ 0.
8. For any types t and t1,..., tn, function[t1,..., tn] [ t is the type of a function with parameter types tni for i = 1..n, where n ≥ 0 and result type t. Functions abstract over expressions.
9. For types T1, ..., Tn, connection[T1, ..., Tn] is the type of a connection (channel in λ-calculus) which can send or receive values of types T1, ..., Tn.
10. For any types $t_1, \ldots, t_n$, $\text{abstraction}[t_1, \ldots, t_n]$ is the type of an abstraction with parameter types $t_i$, for $i = 1 \ldots n$, where $n \geq 0$. Abstractions abstract over behaviours.

The world of data values is defined by the closure of rules 1 to 4 under the recursive application of rules 5 to 10. In addition to the above, clauses which yield no value are of type void.

### 3.2 The Type Algebra

ArchADL provides a simple type algebra that allows the succinct definition of types within programs. As well as the base types and constructors already introduced, types may be defined with the use of aliasing and recursive definitions.

#### 3.2.1 Aliasing

Any legal type description may be aliased by an identifier to provide a shorthand for that type. For example

```plaintext
type count is integer
type person is view [ name : string ; age : integer ]
```

After its introduction an alias may be used in place of the full type description.

#### 3.2.2 Recursive Definitions

Further expressibility may be achieved in the type algebra by the introduction of recursive types. The reserved word `recursive` introduced before a type alias allows instances of that alias to appear in the type definition. Mutually recursive types may also be defined by the grouping of aliases with ampersands. In this case, binding of identifiers within the mutual recursion group takes precedence over identifiers already in scope.

```plaintext
recursive type intList is view [head : integer ; tail : realList]
& realList is view [head : real ; tail : intList]
```

### 3.3 Type Equivalence

Type equivalence in ArchADL is based upon the meaning of types, and is independent of the way the type is expressed within the type algebra. Thus any aliases and recursion variables are fully factored out before equivalence is assessed. This style of type equivalence is normally referred to as structural equivalence.

The structural equivalence rules are as follows:

- Every base type is equivalent only to itself.
For two constructed types to be equivalent, they must have the same constructor and be constructed over equivalent types.

- The size of a sequence is not significant for type equivalence.
- For view constructors the labels are a significant part of the type, but their ordering is not.
- For abstraction types, the parameter ordering is a significant part of the type, but parameter names are not.

ArchADL has no subtyping or implicit coercion rules. Values may be substituted by assignment or parameter passing only when their types are known statically to be equivalent.

The types of all expressions in ArchADL are inferred. There is no other type inference mechanism; in particular, the types of all abstraction parameters must be explicitly stated by the programmer.

### 3.4 First Class Citizenship

The application of the *Principle of Data Type Completeness* [5, 6] ensures that all data types may be used in any combination in the language. For example, a value of any data type may be a parameter to an abstraction. In addition to this, there are a number of properties possessed by all values of all data types that constitute their civil rights in the language and define first class citizenship. All values of data types in ArchADL have first class citizenship.

The additional civil rights that define first class citizenship are:

- the right to be declared,
- the right to be assigned,
- the right to have equality defined over them, and,
- the right to persist.
4 Language Definition Rules

4.1 Context-Free Syntax

The context-free syntax of the ArchWare ADL is specified using Extended Backus-Naur Form (EBNF). A set of productions in EBNF defines the syntax of the language. There is one production per syntactic category.

Productions consist of terminal and non-terminal symbols in the language and meta-symbols of EBNF. The most commonly used meta-symbols and their meanings are given below.

```
 ::= defines
 | choice
 [ a ] a is optional
 + one or more
 * zero or more
```

For example,

```
identifier ::= letter [ letter | digit | _ ]*
```

defines syntactic category identifier as letter optionally followed by zero or more letters, digits or underscores.

4.2 Type Rules

Type judgements, which together with EBNF productions determine the legal, typed set of programs in the ADL, are described using type rules.

These type rules make use of two environments (collections of bindings). Environment binds type identifiers to types in the form <t, T> where t is a type identifier and T is a type. Environment d binds value identifiers to types in the form <x, T> where x is a value identifier and T is a type. In both cases the environment is annotated with a natural number to indicate scope level. For example, [] indicates the collection of type identifier to type bindings available at scope level i. A::b::A denotes that A contains a binding b and A ++ B is a concatenation of collections A and B.

A number of meta operations are also used by the type rules. Meta-operation takes a scope level and one or more type identifier to type bindings are its parameters and updates the type environment for the appropriate scope level. Similarly meta-operation takes a scope level and one or more value identifier to type bindings are its parameters and updates the value environment for the appropriate scope.
level. Meta-operation *id_substitution* takes a scope level and one or more pairs of value identifier names and substitutes the latter name with the former at the specified scope level.

Type rules have the structure of proof rules. For example

\[
\frac{A_1 \ldots A_n}{B}
\]

indicates that if \( A_1 \) to \( A_n \) are true then \( B \) is true. In the case of type rules, each \( A_i \) and \( B \) are of generally the form \( \llbracket, \rrbracket \) \( i \mapsto B \) which is used to indicate that the type judgement \( B \) is deducible from type environment \( \llbracket \) and value environment \( \rrbracket \) for scope \( i \).

Thus the type rule

\[
\frac{\llbracket, \rrbracket \mapsto e_1: \text{integer} \quad \llbracket, \rrbracket \mapsto e_2: \text{integer}}{\llbracket, \rrbracket \mapsto e_1 + e_2: \text{integer}}
\]

is read as “if expression \( e_1 \) can be deduced to be of type integer from environments \( \llbracket \) and \( \rrbracket \), and expression \( e_2 \) can be deduced to be of type integer from environments \( \llbracket \) and \( \rrbracket \) then expression \( e_1 + e_2 \) can be deduced to be of type integer from environments \( \llbracket \) and \( \rrbracket \)”.
5 DECLARATIONS

5.1 Identifiers

In ArchADL, an identifier may be bound to a data value, an abstraction parameter, a view field, or a type. An identifier may be formed according to the syntactic rule

\[
\text{identifier} ::= \text{letter} [\text{id\_follow}] \\
\text{id\_follow} ::= \text{letter} [\text{id\_follow}] | \text{digit} [\text{id\_follow}] | _ [\text{id\_follow}]
\]

That is, an identifier consists of a letter followed by any number of underscores, letters or digits. The following are legal ArchADL identifiers:

\[
x1 \text{ myValue} \text{ look\_for\_Record1} \text{ John}
\]

Note that case is significant in identifiers.

The use of an identifier is governed by the syntactic rule

\[
\text{expression} ::= \text{identifier}
\]

The type rule states that the type of an identifier can be deduced from the value environment \([\cdot]\).

\[
\emptyset, \cdot \vdash <x, T> : \cdot \vdash x : T
\]

5.2 Declaration of Value Identifiers

Before an identifier can be used in ArchADL, it must be declared. The action of declaring a data value associates an identifier with a typed value.

When introducing an identifier, the programmer must indicate the identifier and its value. Identifiers are declared using the following syntax:

\[
\text{value\_declaration} ::= \text{value} \text{id} = \text{clause} [, \text{identifier\_clause\_list}]
\]

\[
\emptyset, \cdot \vdash \text{value} x = e : \text{void} \text{id\_declaration(i, (x,T))}
\]

An identifier is declared by

\[
\text{value} \text{id} = \text{clause}
\]

For example,
value a = 1

introduces an integer identifier with value 1. Notice that the compiler deduces the type.
Identifiers can also be declared for locations, for example,

value discrim = location (b * b - 4.0 * a * c)

introduces a real number location with the calculated value. The value in the location may
be updated by assignment.

### 5.3 Declaration of Types

Type names may be declared by the user in ArchADL. The name is used to represent a set
of values drawn from the value space and may be used wherever a type identifier is legal.
The syntax of type declarations is:

\[
\text{type_declaration} ::= \text{type identifier is type}
\]

\[
\begin{align*}
\emptyset & \mapsto \text{Type} \\
\emptyset & \mapsto \text{type } t \text{ is } T & \text{: void type_declaration}(i, (<t,T>))
\end{align*}
\]

Thus,

**type decision is boolean**

is a type declaration aliasing the identifier `decision` with the boolean type. They are the
same type and may be used interchangeably.
6 Literals

Literals are the basic building blocks of ArchADL programs that allow values to be introduced. A literal is defined by:

\[
\text{literal} ::= \text{integer.literal} \mid \text{real.literal} \mid \text{boolean.literal} \mid \text{string.literal} \mid 
\text{view.literal} \mid \text{sequence.literal} \mid \text{connection.literal} \mid 
\text{behaviour.literal} \mid \text{abstraction.literal} \mid \text{function.literal}
\]

6.1 Integer Literals

These are of type \text{integer} and are defined by:

\[
\text{integer.literal} ::= [\text{add.op}] \text{digit} [\text{digit}]^* \\
\text{add.op} ::= + | - \\
n :: \text{Integer} \\
n : \text{int}
\]

An integer literal is one or more digits optionally preceded by a sign. For example,

\[
\begin{array}{cccc}
1 & 0 & 1256 & -8797 \\
\end{array}
\]

6.2 Real Literals

These are of type \text{real} and are defined by

\[
\text{real.literal} ::= \text{integer.literal} [\text{digit}]^*[e \text{integer.literal}]
\]

\[
r :: \text{Real} \\
r : \text{real}
\]

Thus, there are a number of ways of writing a real literal. For example,

\[
\begin{array}{cccc}
1.2 & 3.1e2 & 5.05 \\
1. & 3.4e-2 & 3.4e+4 \\
\end{array}
\]

3.1e-2 means 3.1 times 10 to the power -2 (i.e. 0.031)
6.3 Boolean Literals

There are two literals of type boolean: true and false. They are defined by

\[ boolean \_ literal \quad ::= \quad true \mid false \]

\[ b \quad \text{Boolean} \]
\[ b : \text{bool} \]

6.4 String Literals

A string literal is a sequence of characters in the character set (ASCII) enclosed by double quotes. The syntax is

\[ string \_ literal \quad ::= \quad "[char]^*" \]

\[ s \quad \text{STRING} \]
\[ s : \text{string} \]

The empty string is denoted by "". Examples of other string literals are:

"This is a string literal", and, "I am a string"

The programmer may wish to have a double quote itself inside a string literal. This requires using a single quote as an escape character and so if a single or double quote is required inside a string literal it must be preceded by a single quote. For example,

"a"" has the value a", and, "a"" has the value a'.

There are a number of other special characters that may be used inside string literals. They are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Character</th>
<th>ASCII code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>backspace</td>
<td>8</td>
</tr>
<tr>
<td>'t'</td>
<td>horizontal tab</td>
<td>9</td>
</tr>
<tr>
<td>'n'</td>
<td>newline</td>
<td>10</td>
</tr>
<tr>
<td>'p'</td>
<td>newpage</td>
<td>12</td>
</tr>
<tr>
<td>'o'</td>
<td>carriage return</td>
<td>13</td>
</tr>
</tbody>
</table>

6.5 View Literals

There is one literal for each view constructor type. It is used to ground recursion in view types.

\[ view \_ literal \quad ::= \quad \text{nilview} \ (type) \]

\[ [] :< t, view[l_1 : T_1, ..., l_n : T_n] >: [j, i] \mapsto \text{nilview} (t) : view[l_1 : T_1, ..., l_n : T_n] \]
6.6 Sequence Literals

There is one literal for each sequence constructor type. It is defined by:

\[
\text{sequence_literal} ::= \text{nilsequence}(\text{type})
\]

\[
\square : \langle t, \text{sequence}[T] \rangle \mapsto \text{nilsequence}(t) : \text{sequence}[T]
\]

6.7 Connection Literals

Connection Literals are defined by:

\[
\text{connection_literal} ::= \text{connection}(\text{type_list})
\]

\[
T \square \text{ Type}
\]

\[
\square, \square \mapsto \text{connection}(T) : \text{connection}[T]
\]

For example,

\[
\text{connection}(\text{integer})
\]

is a connection literal for communicating integer values.

6.8 Behaviour Literal

There is one behaviour literal in ArchADL. It is defined by:

\[
\text{behaviour_literal} ::= \text{done}
\]

\[
\text{done} : \text{behaviour}
\]

done has the same meaning as 0 in \([\_]-calculus and is used to terminate behaviour expressions.

6.9 Abstraction Literals

Abstractions are introduced by their literal values. An abstraction literal is defined by:

\[
\text{abstraction_literal} ::= \text{abstraction}(\{\text{identifier_type_list}\}) ; \text{clause}
\]
For example,

\[
\text{abstraction}(n : \text{integer}) ;
\{
\text{value } c = \text{connection}(\text{integer}) ;
\text{via } c \text{ send } n ; \text{done}
\}
\]

is an abstraction literal.

6.10 Function Literals

Functions are introduced by their literal values. A function literal is defined by:

\[
\text{function literal} ::= \text{function}(\{ \text{identifier type list} \}) -> \text{type} ; \text{clause}
\]

\[
\quad, :< x_1, T_1 >: \ldots :< x_n, T_n >: \quad_2 \mapsto e : S
\]

\[
\quad, :< x_1, T_1 >: \ldots :< x_n, T_n >: \quad_2 \mapsto \text{function}(x_1 : T_1, \ldots, x_n : T_n) ; e : \text{function}[T_1, \ldots, T_n] \quad S
\]

For example,

\[
\text{function}(n : \text{integer}) -> \text{integer} ; n * 2
\]

is a function literal which takes a parameter of type \text{integer} and returns an integer which is two times the parameter value.
7 EXPRESSIONS AND OPERATORS

7.1 Evaluation Order

The order of execution of a ArchADL program is strictly from left to right and top to bottom except where the flow of control is altered by one of the language clauses. This rule becomes important in understanding side effects in the store. Parentheses in expressions can be used to override the precedence of operators.

7.2 Parentheses

In the syntactic description there are two productions:

\[
\text{clause} ::= \ldots \mid \text{expression} \\
\text{expression} ::= \ldots \mid (\text{clause})
\]

\[
\mathcal{E},\mathcal{T} \rightarrow e : T \\
\mathcal{E},\mathcal{T} \rightarrow (e) : T
\]

These rules allow expressions in ArchADL to be written within parentheses. The effect of this is to alter the order of evaluation so that the expressions in parentheses are evaluated first. For example:

\[3 * (2 - 3)\]

evaluates to -3 and not 3.

7.3 Boolean Expressions

Values of type boolean in ArchADL can have the value true or false. There are only two boolean literals, true and false, and four operators. There is one boolean unary operator, \text{\texttt{\~}}, and three boolean binary operators, and, or and implies. They are defined by the truth table below:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>~ a</th>
<th>a or b</th>
<th>a and b</th>
<th>a implies b</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

The syntax rules for boolean expressions are:
expression ::= expression \ expression or expression \ expression and expression \ expression implies expression

\[ \lnot e : boolean \]
\[ e \text{ or } e \]
\[ e \text{ and } e : boolean \]
\[ e \text{ implies } e : boolean \]

The precedence of the operators is important and is defined in descending order as:

\[ \sim \]
\[ \text{and} \]
\[ \text{or} \]
\[ \text{implies} \]

Thus,

\[ \sim a \text{ or } b \text{ and } c \]

is equivalent to

\[ (\sim a) \text{ or } (b \text{ and } c) \]

The evaluation of a boolean expression in ArchADL is non-strict. That is, in the left to right evaluation of the expression, no more computation is performed on the expression than is necessary. For example,

\[ \text{true or expression} \]

gives the value \text{true} without evaluating expression and

\[ \text{false and expression} \]

gives the value \text{false} without evaluating expression.

**7.4 Comparison Operators**

Expressions of type \text{boolean} can also be formed by some other binary operators. For example, \(a = b\) is either \text{true} or \text{false} and is therefore boolean. These operators are called the comparison operators and are:

\[ < \text{ less than} \]
\[ \leq \text{ less than or equal to} \]
> greater than
>= greater than or equal to
= equal to
~= not equal to

The syntactic rules for the comparison operators are:

\[
\begin{align*}
expression & ::= \hspace{1em} expression \hspace{0.5em} rel\_op \hspace{0.5em} expression \\
rel\_op & ::= \hspace{1em} eq\_op \hspace{0.5em} \mid \hspace{0.5em} co\_op \\
eq & ::= \hspace{1em} | \hspace{0.5em} \sim \hspace{0.5em} = \\
< & ::= \hspace{1em} \mid \hspace{0.5em} \ll \hspace{0.5em} \mid \hspace{0.5em} \gg \\
\ll & ::= \hspace{1em} e_1 : T \hspace{0.5em} \ll \hspace{0.5em} e_2 : T \\
\gg & ::= \hspace{1em} e_1 \hspace{0.5em} \ll \hspace{0.5em} T \hspace{0.5em} \ll \hspace{0.5em} e_2 \hspace{0.5em} : \hspace{0.5em} boolean \\
\sim & ::= \hspace{1em} e_1 \hspace{0.5em} \sim \hspace{0.5em} e_2 : boolean \\
\leq & ::= \hspace{1em} e_1 \hspace{0.5em} \leq \hspace{0.5em} e_2 : boolean \\
\geq & ::= \hspace{1em} e_1 \hspace{0.5em} \geq \hspace{0.5em} e_2 : boolean \\
\lt & ::= \hspace{1em} e_1 \hspace{0.5em} \lt \hspace{0.5em} e_2 : boolean \\
\gt & ::= \hspace{1em} e_1 \hspace{0.5em} \gt \hspace{0.5em} e_2 : boolean
\end{align*}
\]

where \( T \subseteq \{ \text{integer}, \text{real}, \text{string} \} \)

\[
\begin{align*}
\ll & ::= \hspace{1em} e_1 : T \hspace{0.5em} \ll \hspace{0.5em} e_2 : T \\
\geq & ::= \hspace{1em} e_1 \hspace{0.5em} \leq \hspace{0.5em} e_2 : boolean \\
\lt & ::= \hspace{1em} e_1 \hspace{0.5em} \lt \hspace{0.5em} T \hspace{0.5em} \lt \hspace{0.5em} e_2 \hspace{0.5em} : \hspace{0.5em} boolean
\end{align*}
\]

where \( T \subseteq \{ \text{integer}, \text{real}, \text{string} \} \)

\[
\begin{align*}
\gt & ::= \hspace{1em} e_1 \hspace{0.5em} \geq \hspace{0.5em} T \hspace{0.5em} \geq \hspace{0.5em} e_2 \hspace{0.5em} : \hspace{0.5em} boolean \\
\lt & ::= \hspace{1em} e_1 \hspace{0.5em} \lt \hspace{0.5em} T \hspace{0.5em} \lt \hspace{0.5em} e_2 : boolean
\end{align*}
\]

where \( T \subseteq \{ \text{integer}, \text{real}, \text{string} \} \)

Note that the operators \(<, \leq, \gt \) and \(\geq \) are defined on integers, reals and strings whereas \(= \) and \(\sim \) are defined on all ArchADL data types. The interpretation of these operations is given with each data type as it is introduced.

Equality for types other than base types is defined as identity.
7.5 Arithmetic Expressions

Arithmetic may be performed on data values of type `integer` and `real`. The syntax of arithmetic expressions is:

```
expression ::= add_op expression | expression add_op expression |
               mult_op expression
add_op ::= + | -
mult_op ::= * | div | rem
real_mult_op ::= * | /
```

```
<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>subtraction</td>
</tr>
<tr>
<td><code>*</code></td>
<td>multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>real division</td>
</tr>
<tr>
<td><code>div</code></td>
<td>integer division throwing away the remainder</td>
</tr>
</tbody>
</table>
```
**7.6 Arithmetic Precedence Rules**

The order of evaluation of an expression in ArchADL is from left to right and based on the precedence table:

<table>
<thead>
<tr>
<th></th>
<th>*</th>
<th>/</th>
<th>div</th>
<th>rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

That is, the operations *, /, div, rem are always evaluated before + and -. However, if the operators are of the same precedence then the expression is evaluated left to right. For example,

\[ 6 \text{ div } 4 \text{ rem } 2 \]

gives the value 1

Brackets may be used to override the precedence of the operator or to clarify an expression. For example,

\[ 3 * ( 2 - 1 ) \]

yields 3 not 5

**7.7 String Expressions**

The string operator, ++, concatenates two operand strings to form a new string. For example,

"abc" ++ "def"

results in the string

"abcdef"

The syntax rule is:

```
expression ::= expression [++ expression]*
```

```
[]:[] ::= e1::string []:[] ::= e2::string
```

A new string may be formed by selecting a substring of an existing string. For example, if s is the string "abcdef" then s (3 | 2) is the string "cd". That is, a new string is formed by selecting 2 characters from s starting at character 3. The syntax rule is:
expression ::= expression (clause \ clause)

e ::= string e ::= integer e ::= string

e ::= e1 | e2

For the purposes of substring selection the first character in a string is numbered 1. The selection values are the start position and the length respectively.

To compare two strings, the characters are compared in pairs, one from each string, from left to right. Two strings are considered equal only if they have the same characters in the same order and are of the same length, otherwise they are not equal.

The characters in a string are ordered according to the ASCII character code. Thus,

"a" < "z"

is true.

The null string is less than any other string. Thus the less-than relation can be resolved by taking the characters pair by pair in the two strings until one is found to be less than the other. When the strings are not of equal length then they are compared as above and then the shorter one is considered to be less that the longer. Thus,

"abc" < "abcd"

The other relations can be defined by using = and <.

### 7.8 Precedence Table

The full precedence table for operators in ArchADL is:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td>div</td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td>rem</td>
</tr>
<tr>
<td>:=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>implies</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8 CLAUSES

This section describes the clauses in ArchADL which allow data values to be manipulated and which provide control over flow of the program.

8.1 If Clause

There are two forms of if clause in ArchADL: if-do clause and if-then-else clause. Their syntax is defined by:

\[
\text{clause} ::= \text{if } \text{clause } \text{do } \text{clause} \mid \text{if } \text{clause } \text{then } \text{clause } \text{else } \text{clause}
\]

\[
\text{if } e : \text{boolean} \text{ then } P : T \quad \text{where } T \in \{\text{void, behaviour}\}
\]

\[
\text{if } e : \text{boolean} \text{ then } P : T \text{ else } Q : T
\]

In the first version, if the condition after if is true then the clause following do is executed. For example,

\[
\text{if } \text{'a} < 0 \text{ do } \text{a} := \text{'a} + 1
\]

The second version allows a choice between two clauses to be made. If the condition after if is true then the clause following then is executed; otherwise the clause following else is executed. For example,

\[
\text{if } \text{'a} < 0 \text{ then } \text{'a} + 1 \text{ else } \text{'a} - 1
\]

8.2 Prefix Clauses

Prefix clauses in ArchADL can be defined by:

\[
\text{prefix} ::= \text{via } \text{identifier send } [\text{clause_list}] \mid \text{via } \text{identifier receive } [\text{identifier_type_list}] \mid \text{observable}
\]
Send and receive clauses are used for communication via connections. These are explained in detail in Section 14.2.

*unobservable* is equivalent to [] in []-calculus. Its type rule is:

\[
\text{unobservable : behaviour}
\]

### 8.3 Replicate Clause

Replicate clause is equivalent to ! in []-calculus. Its syntax is defined by:

\[
\text{clause ::= replicate clause}
\]

\[
\begin{align*}
[] , [] & \mapsto P : \text{behaviour} \\
[] , [] & \mapsto \text{replicate} P : \text{behaviour}
\end{align*}
\]

For example,

\[
\text{replicate} \{ \text{via } \text{connection1 } \text{send} 100 ; \text{via } \text{connection2 } \text{receive} \; s : \text{string} \}
\]

### 8.4 Choose Clause

The *choose* clause is equivalent to + in []-calculus. It allows one of a number of behaviours to be chosen for execution. Its syntax is defined by:

\[
\text{clause ::= choose \{ clause or clause .... \}}
\]

\[
\begin{align*}
[] , [] & \mapsto P_1 : \text{behaviour} \ldots [] , [] & \mapsto P_n : \text{behaviour} \\
[] , [] & \mapsto \text{choose \{ } P_1 \text{ or } \ldots \text{ or } P_n \text{ } : \text{behaviour}
\end{align*}
\]

For example,

\[
\text{choose \{}
\begin{align*}
& \{ \text{via } \text{connection1 } \text{send} 100 ; \text{via } \text{connection2 } \text{receive} \; s : \text{string} \} \\
\text{or}
& \{ \text{via } \text{connection3 } \text{receive} \; i : \text{integer} ; \text{via } \text{connection4 send } \text{“Hello”} \}
\end{align*}
\]

\[
\text{\}}
\]
8.5 Compose Clause

The compose clause is similar to \( \mid \) in \([\cdot]\)-calculus. It allows a number of behaviours to execute and possibly communicate in parallel. Its syntax is defined by:

\[
\text{clause} ::= \text{compose} \{ \text{labelled\_clause} \text{ and } \text{labelled\_clause} \ldots \}
\]

\[
\begin{align*}
\text{where} & \{ \text{unification} \} \\
\text{free} & \text{labelled\_identifier\_list} \\
\text{precedence\_list} & \}
\end{align*}
\]

\[
\text{labelled\_clause} ::= \text{label as clause}
\]

\[
\text{unification} ::= \text{labelled\_identifier} \text{ unifies } \text{dynamic\_identifier} [ , \text{unification} ]
\]

\[
\text{precedence\_list} ::= \text{identifier} > \text{identifier} [ , \text{precedence\_list} ]
\]

\[
\text{dynamic\_identifier} ::= \text{labelled\_identifier} \text{ identifier string\_literal}
\]

\[
\begin{align*}
\pi \cdot \pi & \mapsto \text{P}_1 : \text{behaviour} \ldots \pi \cdot \pi \mapsto \text{P}_n : \text{behaviour}
\end{align*}
\]

where unification stands for \( x_1 \text{ unifies } y_1, \ldots, x_n \text{ unifies } y_n \)

Sub stands for id\_substitution(\( +1, (x_1, y_1), \ldots, (x_n, y_n) \))

The compose clause gives a single handle to a number of behaviours executing in parallel. The where construct with the unifies keyword is used for unifying connection values. Behaviours can communicate either when they have identical connections or when pairs of connections are explicitly unified to the same value in a compose clause. Only those connections which are available in the scope of a compose clause can be unified by that clause.

Any unused connections of the listed behaviours will automatically be available to communicate after the compose. The free construct allows us to specify any unified connections which will still be available for communication after the compose clause.

Labelling behaviours allows users to specify connections even when two or more behaviours have connections of the same name. Meaningful labels can also help users identify the functionality and purpose of behaviours when composed behaviours are decomposed (see section 8.6) as labels are returned along with component behaviours.

The precedence construct permits users to indicate communication from which connections should be given priority over others. Precedence is indicated by pairs of connections.

For example,

\[
\begin{align*}
\text{value } b1 = \{ \text{ via } c1 \text{ send } 100 ; \text{ via } c2 \text{ receive } s : \text{string} ; \text{ done } \} ; \\
\text{value } b2 = \{ \text{ via } c3 \text{ receive } i : \text{integer} ; \text{ via } c4 \text{ send } \text{“Hello”} ; \text{ done } \} ; \\
\text{compose }\
\end{align*}
\]
send_int_receive_string as b1

and

receive_int_send_string as b2

where { c1 unifies c3, c2 unifies c4 } free c1

8.6 Decompose Clause

The \emph{decompose} clause allows us to access the components of a composite behaviour produced by the \emph{compose} clause. Its syntax is defined by:

\begin{verbatim}
clause ::= decompose clause
\end{verbatim}

\begin{verbatim}
[] ,[] \rightarrow P : behaviour
[] ,[] \rightarrow decompose P : sequence[ T ]
\end{verbatim}

where T stands for \texttt{view[ bhvr : behaviour ; label : string ;
connections : sequence[ any ] ]}

For example,

\begin{verbatim}
value composite = compose {
    send_int_receive_string as b1
    and
    receive_int_send_string as b2
    where { c1 unifies c3, c2 unifies c4 }
}

... 
value behaviour_seq = decompose composite ;
value b_value1 = behaviour_seq::1.bhvr ; ! b1
value b_label2 = behaviour_seq::1.label ! "receive_int_send_string"
\end{verbatim}

decomposing \emph{composite} will result in a sequence of two views with details of b1 and b2 respectively. Decompose will also undo any unification performed as part of composition.
8.7 Project Clause

The *project* clause allows values to be projected from any. It is described in detail in Section 9.2.

8.8 Assignment Clause

The assignment clause allows the contents of locations to be updated. It is described in detail in Section 11.2.

8.9 Iterate Clause

The *iterate* clause allows users to iterate over sequences of values. It is described in detail in Section 13.2.
9 ANY

Type any is the type of the union of all values in ArchADL. Values must be explicitly injected into and projected from type any. Both of these operations are performed dynamically and, in particular, the projection from any to another type involves a dynamic type check.

9.1 Injection into Type any

Values may be injected into type any by the following syntax:

\[ \text{any (clause)} \]

\[ \emptyset,\emptyset \mapsto e : T \]

\[ \emptyset,\emptyset \mapsto \text{any}(e) : \text{any} \]

For example,

```
value int_any = any(-42)
```

which declares int_any to be the integer value -42 injected into type any.

Values of type any may be passed as parameters. For example, the following is an abstraction that takes a parameter of type any.

```
value abs_any = abstraction (x : any) :
{ 
  value any_c = free connection( any ) ;
  via any_c send x ; done 
}
```

Thus polymorphic abstractions may be written by using type any and injecting the parameters into any before the call.

9.2 Projection from Type any

Values may be projected from type any by use of the project clause.

\[ \text{project} \quad ::= \quad \text{project clause as identifier onto project list default : clause} \]

\[ \text{project list} \quad ::= \quad \text{type id : clause} \quad ; \quad \text{[project list]} \]

\[ \emptyset,\emptyset \mapsto e : \text{any} \quad T_1,\ldots,\quad T_n \quad \text{Type} \quad \emptyset,\emptyset \{1\ldots n\} \quad \emptyset,\emptyset' \mapsto e_i : T \]

\[ \emptyset,\emptyset \mapsto \text{project e as x onto (project list ; default e_n+1) : T} \]
where \([\mathbb{P}_i]\) stands for \([\mathbb{P}_i]\) and project_list stands for \(T_1 : e_1 ; \ldots ; T_n : e_n\).

The projected value is bound to the identifier following the \texttt{as}. The scope of the identifier is the clauses on the right hand side of the colons. This mechanism prevents side effects on the projected value inside the evaluation of the right hand side clauses and allows for static type checking therein. For projection, the type is compared to each of the types on the left hand side of the colons. The first match causes the corresponding clause on the right hand side to be executed. Within the clause, the identifier has the type of the projected value. After execution of the \texttt{project} clause, control passes to the clause following the \texttt{project} clause.

An example of projection is:

```
project x as X onto
  integer : "type is integer"
  real : "type is a real"
default : "type is neither integer nor real"
```

### 9.3 Equality and Equivalence

Two values of type \texttt{any} are equal if and only if they can be projected onto equivalent types and the projected values are equal.

All values of type \texttt{any} are type equivalent.
10 View

10.1 Creation of View

Values of different types can be grouped together into a view. The fields of a view have
t Identifiers that are unique within that view. The views are sets of labelled cross products
from the value space. Views are created using a type identifier. The syntax of view types is:

\[
\text{type} \quad ::= \quad \text{view} [\text{labelled_type_list}]
\]
\[
\text{labelled_type_list} \quad ::= \quad \text{identifier_list} : \text{type} \mid \text{labelled_type_list}
\]

For example, a view type may be declared as follows:

\[
\text{type person is view [name : string ; age, height : integer]}
\]

This declares a view type, person, with three fields of type string, integer and integer, with
labels: name, age and height respectively.

A view may be created by the following syntax:

\[
\text{expression} \quad ::= \quad \text{view} (\text{value_init_list})
\]
\[
\text{value_init_list} \quad ::= \quad \text{value_init} [, \text{value_init}]
\]
\[
\text{value_init} \quad ::= \quad \text{identifier} = \text{clause}
\]

\[
\{,\} \mapsto e_1 : T_1 \ldots \{,\} \mapsto e_n : T_n
\]

\[
\{,\} \mapsto \text{view}(l_1 = e_1, \ldots, l_n = e_n) : \text{view}[l_1 : T_1, \ldots, l_n : T_n]
\]

For example,

\[
\text{value jane = view (name = "Jane Doe", age = 40, height = 160)}
\]

creates a view with field labels name, age and height and with field values "Jane Doe", 40
and 160 respectively.

10.2 Dereferencing A View

To obtain a field of a view, the field identifier is used as an index. The syntax is

\[
\text{expression} \quad ::= \quad \text{clause.identifier}
\]

\[
\{,\} \mapsto e : \text{view}[l_1 : T_1, \ldots, l_n : T_n]
\]

\[
\{,\} \mapsto e . l_i : T_i
\]
For example, if \textit{jane} is declared as above, then,

\texttt{jane.age}

yields 40. For the indexing operation to be legal, the view must contain a field with that identifier.

Field identifiers, when used as indices, are only in scope after the dot following a view expression. Thus these identifiers need only be unique within each view type.

### 10.3 Equality and Equivalence

Two views are equal if they have the same identity (pointer).

The type of a view is the set of the field identifier-type pairs. Thus the view \textit{jane} has type:

\begin{verbatim}
view [name : string ; age : integer ; height : integer]
\end{verbatim}

Two views have equivalent types when the types have the same set of identifier-type pairs for the fields. Note that the order of the fields is unimportant.
11 Location

Values may be stored in locations and subsequently retrieved.

11.1 Creation and Dereference

The constructor location creates a location and initialises the value in it. The operator ’ (dereference) retrieves the value from the location. Since locations are values in ArchADL they may also be stored in locations. The syntactic rules are:

expression ::= location (clause) | ’clause

\[ \text{[1]} : e : T \]

\[ \text{[2]} : location(e) : location[T] \]

\[ \text{[3]} : e : location[T] \]

\[ \text{[4]} : ’e : T \]

For example, if a is of type location [integer] with the value location (3) then ’a has the value 3.

11.2 Assignment

The assignment clause allows the content of a location to be updated. Its syntax is defined by:

clause ::= expression ::= clause

\[ \text{[5]} : e : location[T] \]

\[ \text{[6]} : e : e_1 : T \]

\[ \text{[7]} : e : := e_1 : \text{void} \] where \( T \neq \text{void} \)

For example,

```
value my_loc = location(100);
my_loc := ’my_loc * 100
```

11.3 Equality and Equivalence

Two locations are equal if they have the same identity, that is, the same location. Two locations are type equivalent if they have equivalent content types.
12 FUNCTION

Functions in ArchADL constitute abstractions over expressions if they return a value and over clauses of type void if they do not.

The formal parameters of a function representing data values must have a name and a type. A function which returns a value must also specify its return type. The scope of the formal parameters is from their declaration to the end of the function clause.

12.1 Creation

In ArchADL functions are introduced by their literal values. Section 6.10 describes how function literals may be defined.

For example,

\[
\text{value int_id = function( n : integer ) -> integer ; n}
\]

defines the integer identity function int_id.

Recursive and mutually recursive functions are also permitted in ArchADL. For the example the factorial function may be defined as

\[
\text{recursive value factorial = function( n : integer ) -> integer}
\]

\[
\text{if n = 0 then 1 else n * factorial( n - 1 )}
\]

12.2 Application

In function applications, there must be a one-to-one correspondence between the actual and formal parameters and their types. The syntax for applications is

\[
\text{expression ::= expression( [ clause_list ] )}
\]

\[
\text{[ ]} \quad \text{e : function}[T_1, ..., T_n] \quad [S] \quad \text{[ ]} \quad \text{e_1 : T_1} \quad [S] \quad \text{[ ]} \quad \text{e_n : T_n}
\]

Thus, to call the integer identity function given above, the following could be used,
int_id (42)

which will evaluate to the integer 42.

### 12.3 Equality and Equivalence

Two functions are equal in ArchADL if and only if their values are derived from the same evaluation of the same function expression. This means that they have the same closure.

Two function types are structurally equivalent if they have the same parameter types in one-one correspondence and the same result type.
13 SEQUENCE

13.1 Creation of Sequences

A sequence provides a method of grouping together values of the same type. Since ArchADL does not allow uninitialised objects, all the initial values of the elements must be specified. The syntax is:

\[
\text{expression} \quad ::= \quad \text{sequence}( \text{clause\_list} ) ~ | ~ \\
\text{sequence for identifier = clause to clause using clause}
\]

\[
\text{clause\_list} \quad ::= \quad e_1 : T \ldots e_n : T \\
\text{sequence}( e_1, \ldots, e_n ) : \text{sequence}[ T ]
\]

\[
\text{clause\_list} \quad ::= \quad e_1 : \text{integer} \quad e_2 : \text{integer} \quad e : \text{function}[ \text{integer} ] - \to T \\
\text{sequence for} \quad i = e_1 \text{ to } e_2 \text{ using } e : \text{sequence}[ T ]
\]

For example,

\begin{verbatim}
value abc = sequence ( "a", "b", "c" )
\end{verbatim}

declares \textit{abc} to be a sequence of strings, whose type is written as \texttt{sequence[ string ]}, with elements initialised to “a”, “b” and “c”.

Multi-dimensional sequences, which are not necessarily rectangular, can also be created. For example,

\begin{verbatim}
value Pascal = sequence ( 
  sequence ( 1 ),
  sequence ( 1, 1 ),
  sequence ( 1, 2, 1 ),
  sequence ( 1, 3, 3, 1 ),
  sequence ( 1, 4, 6, 4, 1 ),
  sequence ( 1, 5, 10, 10, 5, 1 ) )
\end{verbatim}

\textit{Pascal} is of type \texttt{sequence[ sequence[ integer ] ]}.

A second form of sequence initialisation is provided to allow the elements of a sequence to be initialised by a function over the index. For example,
value square_fun = function( k : integer ) -> integer ; k * k ;
value squares_sequence = sequence for i = 1 to 10 using square_fun

In the initialisation, the function \textit{square\_fun} is called for every index of the sequence in order. The corresponding element is initialised to the value of its own index being used by the function. In the above case, the sequencer \textit{squares\_sequence} has elements 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100.

For a sequence of type \textit{sequence}\[ t \], the initialising function must be of type \textit{function[ integer ] -> t}.

### 13.2 Indexing and Iterating

To obtain the elements of a sequence, indexing is used. For sequences, the index is always an integer value. The syntax is:

\[
\begin{align*}
expression & ::= expression :: clause \\
\emptyset,
\{ & \mapsto e : sequence( T ) \} \emptyset,
\{ & \mapsto e_1 : integer \\
\emptyset,
\{ & \mapsto e :: e_1 : T
\end{align*}
\]

For example,

a (3 + 4)

selects the element of the sequence \textit{a} which is associated with the index value 7. Multi-
dimensional sequences may be indexed by using commas to separate the indices.

An iteration construct is also provided for sequences. This is defined by:

\[
\begin{align*}
expression & ::= iterate clause [by identifier : type] from identifier = clause accumulate clause [as identifier] \\
\emptyset,
\{ & \mapsto e :: sequence( T ) \} \emptyset,
\{ & \mapsto e_1 : T_i \} \emptyset, \{ & \mapsto e_2 : S
\end{align*}
\]

where \(\emptyset\) stands for \(\emptyset::<i, T>:<a, T_1>::\emptyset\) and \(\emptyset^\uparrow\) stands for \(\emptyset::<a, T_1>::\emptyset\) and \(\emptyset^\uparrow\) stands for \(id\_declaraton(i, (i, T), (a, T_1))\)

For example,

value int_sequence = sequence( 1, 2, 3, 4, 5, 6, 7, 8, 9 ) ;
iterate int_sequence by k from j = location( 100 ) accumulate \(\text{\textquoteleft}j + k\text{\textquoteleft}\)

will result in \(j\) having a final value of \text{\textquoteleft}location( 145 )\text{\textquoteleft}.
13.3 Adding and Removing Elements

Sequences are not of a fixed length. Elements may be added or removed from a sequence by using the following syntax:

\[
\begin{align*}
\text{expression} & \ ::= \text{expression including expression} \setminus \text{expression excluding expression} \\
\text{sequence}[\,] & \ ::= \, e : \, \text{sequence}[\,] \\
\text{sequence} & \ ::= \, e \, \text{including} \, e_1 : \, \text{sequence}[\,] \\
\text{sequence} & \ ::= \, e \, \text{excluding} \, e_1 : \, \text{sequence}[\,]
\end{align*}
\]

Including appends the new element to the end of a sequence whilst excluding removes the first occurrence of the value from the sequence.

13.4 Equality and Equivalence

Two sequences are equal if they have the same identity, that is, the same pointer. Two sequences are type equivalent if they have equivalent element types. Notice that the bounds are not part of the type.
14 CONNECTION

14.1 Creating Connections

Connections correspond to channels in \( \tau \)-calculus. They are the means of communication in ArchADL. Connection literals may be defined by using the syntax given in Section 6.7. For example, a connection for communicating integers may be defined as

\[
\text{value} \quad \text{int\_connection} = \text{connection} ( \text{integer} )
\]

ArchADL corresponds to the polyadic \( \tau \)-calculus and hence connections can communicate more than one value at a time. In the following example, the connection defined can be used to communicate a pair of values: an integer and a string.

\[
\text{value} \quad \text{pair\_connection} = \text{connection} ( \text{integer, string} )
\]

14.2 Communicating via Connections

ArchADL provides two communication constructs. Values can be sent or received via connections using the following syntax:

\[
\text{prefix} \quad ::= \quad \text{via} \ \text{identifier} \ \text{send} \ [ \ \text{clause\_list} \ ] \ |
\text{via} \ \text{identifier} \ \text{receive} \ [ \ \text{identifier\_type\_list} \ ]
\]

\[
\text{send}: \quad \emptyset , \emptyset \mapsto x : \text{connection}[S] \quad \emptyset , \emptyset \mapsto y : S
\]
\[
\text{receive}: \quad \emptyset , \emptyset \mapsto \text{via} \ \times \ \text{send} \ y : \text{behaviour}
\]

\[
\emptyset , \emptyset \mapsto \text{via} \ \times \ \text{receive} \ y : S : \text{behaviour} \quad \text{id\_declaration}(i, (y, S))
\]

14.3 Equality and Equivalence

Two connections are equal if they have the same identity, that is, the same pointer. Two connections are type equivalent if they have equivalent types to be communicated.
15 Behaviour

Behaviours in ArchADL may be obtained by applying abstraction literals with any required actual parameters (Section 16.1). Most clauses in the language, such as compose and choose and all communication prefixes are also typed as behaviour.

15.1 Creating Behaviours

In addition to the above means, behaviours can also be created by:

\[
\text{expression} ::= \text{behaviour clause}
\]

\[
\emptyset \rightarrow c : \text{behaviour}
\]

\[
\emptyset \rightarrow \text{behaviour e : behaviour}
\]

15.2 Querying Behaviours

ArchADL provides a construct to query a behaviour to find any connections it may have. This can be done with the following syntax:

\[
\text{expression} ::= \text{connections( clause )}
\]

\[
\emptyset \rightarrow c : \text{behaviour}
\]

\[
\emptyset \rightarrow \text{connections(e): sequence[ any ]}
\]

connections returns a sequence containing all connection values defined within e injected into anys.

In the following example,

```plaintext
value abs1 = abstraction()
{      value c1 = connection( integer ), c2 = connection( string ) ;
        via c1 send 100 ; via c2 receive s : string ; done
} ;
value b1 = abs1() ;
value con_seq = connections( b1 ) ;
value con1_any = con_seq::1 ;
value con2_any = con_seq::2
```
con_seq is a sequence of two any values as there are two connections in the behaviour b1.

15.3 Equality and Equivalence

Two behaviours are equal if they have the same identity, that is, the same pointer. All values of type behaviour are type equivalent.
16 ABSTRACTION

Abstractions in ArchADL abstract over behaviours. Abstraction literals may be defined as shown in Section 6.9.

For example,

```
abstraction( i : integer )
{
  value c1 = connection( integer ), c2 = connection( string ) ;
  via c1 send i ; via c2 receive s : string ; done
}
```

defines an abstraction literal.

16.1 Abstraction Application

Abstractions can be applied by associating each formal parameter with an actual parameter value to produce behaviours. The syntax for abstraction application is:

```
expression :::= expression( { identifier_clause_list } )
```

```
[[,...]] e : abstraction[ x_1 : T_1 , ..., x_n : T_n ]
[[,...]] e_1 : T_1 ...
[[,...]] e_n : T_n
[[,...]] e( x_1 = e_1 , ... , x_n = e_n ) : behaviour
```

The following example shows an abstraction application:

```
value abs2 = abstraction( i : integer )
{
  value c1 = connection( integer ), c2 = connection( string ) ;
  via c1 send i ; via c2 receive s : string ; done
}
```

16.2 Partial Application

In ArchADL, it is possible to have a partial application of an abstraction i.e. not all formal parameters need to have actual parameters associated with them at the same time. A
partial application of an abstraction results in an abstraction of a different type. The type rule for partial application is:

\[
\mathcal{e}_1 \mapsto e \mathbin{\text{ abstraction}} \mathcal{e}_2 \mapsto e \mathbin{\text{ abstraction}} \mathcal{e}_3 \mapsto e \mathbin{\text{ abstraction}} \mathcal{e}_4 \mapsto e \mathbin{\text{ abstraction}}
\]

where \(1 \leq j < n\)

For example,

```plaintext
value abs3 = abstraction( i : integer, r : real, b : boolean )
{    value c1 = connection( integer ), c2 = connection( string ) ;
    value c3 = connection( real ), c4 = connection( boolean ) ;
    via c1 send i ; via c2 receive s : string ;
    via c3 send r ; via c4 send b ; done
}

value abs4 = abs3( i = 100, b = false )
```

the above code will result in abs4 being typed as \(\text{abstraction}[r : \text{real}]\).

### 16.3 Equality and Equivalence

Two abstractions are equal if they have the same identity, that is, the same pointer. Two abstractions are type equivalent if they have equivalent parameter types.
17 AN EXAMPLE: A CLIENT-SERVER SYSTEM

The following example, illustrating the use of ArchADL, is taken from [7]. Certain details are abstracted over in order to make the example easier to understand.

Consider a server that disseminates data about time, date and the position of a satellite. A number of clients may request data from this server.

The functionality of the server and the client can be modelled as abstractions in the ArchWare ADL. When applied, these abstractions yield executing behaviours. Such behaviours are the components that make up the client-server system. The repetitive nature of both client and server functionalities is captured using the replicate clause.

Components interact by communicating via connections. Each component may specify the connections it uses to communicate with others. At the time of composition, these connections may be unified to make communication possible.

```archware
! client
value client_abs = abstraction()
replicate
{   value c_put = connection () ;   ! request connection
    value c_get = connection( string ) ;   ! reply connection
    via c_put send ;   ! send request
    via c_get receive s : string ;   ! receive reply
    via c_display send s ;   ! display reply
}
```

In the simple client code above, a client sends a signal via connection \texttt{c\_put}, then receives a reply via connection \texttt{c\_get}, and then sends the reply value via connection \texttt{c\_display}.

In the example server below, the connection used determines the nature of the request. For example, a request received via connection \texttt{c\_put\_s\_get\_time} will be for time. The server will choose to receive a request from one of the three connections and respond to it.
! Global data items to keep count of server activities
value time_count, date_count, pos_count = location(integer) ;

! server
value server_abs = abstraction()
replicate {
  value c_put_s_get_time, c_put_s_get_date, ! connections to receive requests
c_put_s_get_pos = connection();
  value s_put_c_get_time, s_put_c_get_date, ! connections to send data
  s_put_c_get_pos = connection(string);
choose{
    ! server makes a choice of which request to service
    { via c_put_s_get_time receive ; ! request for time
time_count := time_count + 1 ; ! increment time count
via s_put_c_get_time send time } ! send time
or
    { via c_put_s_get_date receive ; ! request for date
date_count := date_count + 1 ; ! increment date count
via s_put_c_get_date send date } ! send date
or
    { via c_put_s_get_pos receive ; ! request for satellite position
pos_count := pos_count + 1 ; ! increment position count
via s_put_c_get_pos send satellite_position } } ! send position
};

Having defined server and client abstractions, we will now create a client-server system by composing one server and three client instances with appropriate renaming. Note that other topologies are also possible, for example two servers and five clients. Renaming ensures that corresponding client and server connections are matched for communication. Defining the composition as a value gives us a handle (CS_system1) to the resulting behaviour.
! build client-server system
value CS_system1 =
  compose{
    ! compose components
    c1 as client_abs() ! client for time
    and c2 as client_abs() ! client for date
    and c3 as client_abs() ! client for position
    and s1 as server_abs() ! server
  
  where{ c1::c_put unifies s1::c_put_s_get_time,
            c1::c_get unifies s1::s_put_c_get_time,
            c2::c_put unifies s1::c_put_s_get_date,
            c2::c_get unifies s1::s_put_c_get_date,
            c3::c_put unifies s1::s_put_c_get_pos,
            c3::c_get unifies s1::s_put_c_get_pos 
  }
} ;

Once the system starts executing, we may wish to change its structure. Feedback from the system, efficiency concerns and changing requirements can contribute to such a decision. We begin this process by decomposing the system into its component parts. The decompose clause returns a sequence whose elements contain details of the components.

! decompose system
value CS_seq = decompose CS_system1 ;

Necessary changes can then be made by evolving or redefining some components. In this case we wish to split the functionality of the server into two by creating two new servers, one serving time alone and the other serving date and satellite position. Therefore we create two new abstractions to replace the old server_abs.

Using hyper-code representations of the abstractions will enable us to define the new abstractions to use the current values of the count variables without them having to be stored and explicitly reinitialised.
! time server
value time_server_abs = abstraction()
replicate
{   value s_get_time = connection();
    value s_put_time = connection(string);
    via s_get_time receive;
    time_count := 'time_count + 1 ;          ! reference to extant data
    via s_put_time send time
} ;

! date and satellite position server
value date_sat_server_abs = abstraction()
replicate
{   value s_get_date, s_get_sat_pos = connection();
    value s_put_date, s_put_sat_pos = connection(string);
    choose {
        { via s_get_date receive;
            date_count := 'date_count + 1 ;          ! reference to extant data
            via s_put_date send date 
        }
        or
        { via s_get_sat_pos receive;
            pos_count := 'pos_count + 1 ;          ! reference to extant data
            via s_put_sat_pos send satellite_position } } }

A new client-server system can then be formed by composing the two new servers with the decomposed clients appropriately.
! make new client-server system
value CS_system2 =
    compose{
        nc1 as CS_seq::1.bhvr
        and nc2 as CS_seq::2.bhvr
        and nc3 as CS_seq::3.bhvr
        and ts as time_server_abs()
        and dss as date_sat_server_abs()

        where{
            nc1::c_put unifies ts::c_put_s_get_time,
            nc1::c_get unifies ts::s_put_c_get_time,
            nc2::c_put unifies dss::c_put_s_get_date,
            nc2::c_get unifies dss::s_put_c_get_date,
            nc3::c_put unifies dss::s_put_c_get_pos,
            nc3::c_get unifies dss::s_put_c_get_pos 
        }
    };

Now client c1 will communicate with time_server and clients c2 and c3 will communicate with date_sat_server.
REFERENCES


APPENDIX A: CONTEXT-FREE SYNTAX DEFINITION

Declaration

description ::= declaration [ ; description] |
               clause [ ; description]

declaration ::= type_declaration | value_declaration

Type declaration

type_declaration ::= type type_definition |
                   recursive type type_definition
                   [ & type_definition]*

type_definition ::= identifier is type

Type descriptor

type ::= integer | real | boolean | string | any |
       connection[ [type_list] ] | behaviour |
       abstraction[ [type_list] ] | identifier |
       view[ identifier_type_list ] |
       function[ [type_list] ] -> type |
       location[ type ] | sequence[ type ]

type_list ::= type [, type_list ]

identifier_type_list ::= identifier : type [, identifier_type_list ]

Value declaration

value_declaration ::= value identifier_clause_list |
                   recursive value identifier_literal_list

identifier_clause_list ::= identifier = clause [, identifier_clause_list ]
identifier_literal_list  ::=  identifier = literal [ & identifier_literal_list ]

**Clause**

clause  ::=  if clause then clause else clause |

if clause do clause |

replicate clause |

compose { parallel_list [ where { unification } ] |

[ free labelled_identifier_list ] |

[ precedence precedence_list ] } |

decompose clause |

prefix |

choose { choice_list } |

project clause as identifier onto project_list |

default : clause |

eexpression ::= clause |

iterate clause [ by identifier : type ]

from identifier = clause accumulate clause |

[ as identifier ] |

eexpression

parallel_list  ::=  label as clause [ and parallel_list ]

unification  ::=  labelled_identifier unifies dynamic_identifier |

[ , unification ]

labelled_identifier  ::=  label :: identifier

labelled_identifier_list ::= labelled_identifier [ , labelled_identifier_list ]

dynamic_identifier  ::=  labelled_identifier | identifier string_literal

precedence_list ::= identifier > identifier [ , precedence_list ]

choice_list ::= clause [ or choice_list ]

project_list ::= type : clause ; [ project_list ]
prefix ::= via identifier send [clause_list]
    | via identifier receive [identifier2_type_list]
    | unobservable

identifier_type_list ::= identifier_type [, identifier_type_list]
identifier2_type_list ::= identifier [: type] [, identifier2_type_list]
identifier_type ::= identifier : type

Expression

expression ::= ( clause )
    | { description } [ verify text end ]
    | behaviour clause
    | literal
    | not expression
    | expression and expression
    | expression or expression
    | expression implies expression
    | add_operator expression
    | expression relational_operator expression
    | expression add_operator expression
    | expression multiply_operator expression
    | expression ++ expression
    | expression ( clause | clause )
    | any( clause )
    | expression( [ identifier_clause_list ] )
    | expression( [ clause_list ] )
    | connections( clause )
    | view( identifier_clause_list )
clause . identifier |

location( clause ) |

* clause |

sequence( clause_list ) |

sequence for identifier = clause to clause using clause |

expression :: clause |

expression including expression |

expression excluding expression |

identifier |

clause_list ::= clause [, clause_list]

identifier_list ::= identifier [, identifier_list]

relational_operator ::= equality_operator | ordering_operator

equality_operator ::= = | <>

ordering_operator ::= < | <= | > | =>

add_operator ::= + | -

multiply_operator ::= integer_multiply_operator |

real_multiply_operator

integer_multiply_operator ::= * | div | rem

real_multiply_operator ::= * | /

**Literal**

literal ::= integer_literal | real_literal |

boolean_literal | string_literal |

connection_literal | behaviour_literal |

abstraction_literal | view_literal |

sequence_literal | function_literal
integer_literal ::= [add_operator] digit [digit]*
real_literal ::= integer_literal.[digit]*[e integer_literal]
boolean_literal ::= true | false
string_literal ::= "character"*
connection_literal ::= connection( type_list )
behaviour_literal ::= done
abstraction_literal ::= abstraction( [ identifier_type_list ] ) ; clause
view_literal ::= nilview( type )
sequence_literal ::= nilsequence( type )
function_literal ::= function( [ identifier_type_list ] ) [ -> type ] ; clause

**Identifier**
identifier ::= letter [letter | digit | _]*
label ::= letter [letter | digit | _]*
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 |
        | 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
character ::= any ASCII character
APPENDIX B : TYPE RULES

Declaration

[description]

\[ \frac{\text{\{} \leftrightarrow \text{D : void Decl} \quad \text{\{} ++ \text{\} ++} \leftrightarrow e : T \quad \text{\{} \leftrightarrow \text{D ; e : T} } \]

where Decl stands for type_declaration( i, \{\} ) and id_declaration( i, \{\} )

[type declaration]

\[ \frac{\{} \leftrightarrow T \{\} \text{ Type} \quad \{} \leftrightarrow \text{type t is T : void type_declaration(i, (t, T))} \]

[recursive type declaration]

\[ \frac{\text{\{} \leftrightarrow \text{T_1 \{\} Type} \quad \text{\{} \leftrightarrow \text{T_2 \{\} Type} \quad \{} \leftrightarrow \text{recursive type t_1 is T_1 & t_2 is T_2 : void Decl} } \]

where Decl stands for type_declaration( i, (t_1, T_1), (t_2, T_2) ) and

\[ \} \] stands for \{\}::< t_1, T_1 >::< t_2, T_2 >::<\}

[value declaration]

\[ \frac{\text{\{} \leftrightarrow e : T \quad \text{\} \leftrightarrow value x = e : void id_declaration(i, (x, T))} \]

[recursive value declaration]

\[ \frac{\{} \leftrightarrow e_1 : T_1 \quad \text{\{} \leftrightarrow e_2 : T_2 \quad \{} \rightarrow \text{recursive value x_1 = e_1 & x_2 = e_2 : void Decl} } \]

where Decl stands for id_declaration(i, (x_1, T_1), (x_2, T_2) ) and

\[ \} \] stands for \{\}::< x_1, T_1 >::< x_2, T_2 >::<\}
Clause

[send] \[x : \text{connection}[S] \quad y : \text{S} \quad \text{via} \times \text{send} \quad y : \text{behaviour} \]

[receive] \[x : \text{connection}[S] \quad \text{via} \times \text{receive} \quad y : S : \text{behaviour} \quad \text{id}\_\text{declaration}(i, (y, S)) \]

[unobservable] \[\text{unobservable} : \text{behaviour} \]

[then] \[P : S \quad Q : T \quad \text{P} ; \text{Q} : T \]
where \( S \in \{ \text{void, behaviour} \} \)

[summation] \[P_1 : \text{behaviour} \ldots P_n : \text{behaviour} \quad \text{choose} \{ P_1 \text{ or } \ldots \text{ or } P_n \} : \text{behaviour} \]

[replication] \[P : \text{behaviour} \quad \text{replicate} \quad P : \text{behaviour} \]

[guard-1] \[e : \text{boolean} \quad P : T \quad \text{if} \ e \ \text{do} \quad P : T \]
where \( T \in \{ \text{void, behaviour} \} \)

[guard-2] \[e : \text{boolean} \quad P : T \quad Q : T \quad \text{if} \ e \ \text{then} \quad \text{P else} \quad Q : T \]
[compose]

\[ \emptyset \emptyset \mapsto P_1 : \text{behaviour} \quad \ldots \quad \emptyset \emptyset \mapsto P_n : \text{behaviour} \]

\[ \emptyset \emptyset \mapsto \text{compose} \{ P_1 \text{ and } \ldots \text{ and } P_n \text{ where } \{ \text{unification} \} \} : \text{behaviour} \]

where unification stands for \( x_1 \text{ unifies } y_1 \), \( \ldots \), \( x_n \text{ unifies } y_n \) and Sub stands for \( \text{id\_substitution}(i+1, (x_1, y_1), \ldots, (x_n, y_n)) \)

[decompose]

\[ \emptyset \emptyset \mapsto P : \text{behaviour} \]

\[ \emptyset \emptyset \mapsto \text{decompose} \ P : \text{sequence}[T] \]

where \( T \) stands for \text{view} [ bhvr : \text{behaviour} ; \text{label} : \text{string} ; \\ \text{connections} : \text{sequence}[\text{any}]]

[assignment]

\[ \emptyset \emptyset \mapsto e : \text{loc}[T] \quad \emptyset \emptyset \mapsto e_1 : T \]

\[ \emptyset \emptyset \mapsto e := e_1 : \text{void} \]

where \( T \neq \text{void} \)

[iterate]

\[ \emptyset \emptyset \mapsto e : \text{sequence}[T] \quad \emptyset \emptyset \mapsto e_1 : T_1 \quad \emptyset \emptyset \mapsto e_2 : S \quad \emptyset \emptyset \mapsto e_3 : S_1 \]

\[ \emptyset \emptyset \mapsto \text{iterate} \ e \text{ by } i \text{ from } a = e_1 \text{ accumulate } e_2 \text{ in } e_3 : S_1 \text{ Decl} \]

where \( \emptyset \emptyset \) stands for \( \emptyset_n : <i, T> :<a, T_1> : \emptyset_2 \) and \( \emptyset^n \) stands for \( \emptyset_n :<a, T_1> : \emptyset_2 \) and Decl stands for \( \text{id\_declaraton}(i, (i, T), (a, T_1)) \)

Expression

Boolean

[negation]

\[ \emptyset \emptyset \mapsto e : \text{boolean} \]

\[ \emptyset \emptyset \mapsto \text{not} \ e : \text{boolean} \]
[or] \[ \varnothing, \varnothing \mapsto e_1 : \text{boolean}, \varnothing, \varnothing \mapsto e_2 : \text{boolean} \quad \varnothing, \varnothing \mapsto e_1 \text{ or } e_2 : \text{boolean} \]

[and] \[ \varnothing, \varnothing \mapsto e_1 : \text{boolean}, \varnothing, \varnothing \mapsto e_2 : \text{boolean} \quad \varnothing, \varnothing \mapsto e_1 \text{ and } e_2 : \text{boolean} \]

[implies] \[ \varnothing, \varnothing \mapsto e_1 : \text{boolean}, \varnothing, \varnothing \mapsto e_2 : \text{boolean} \quad \varnothing, \varnothing \mapsto e_1 \text{ implies } e_2 : \text{boolean} \]

Comparison

[equality] \[ \varnothing, \varnothing \mapsto e_1 : T, \varnothing, \varnothing \mapsto e_2 : T \quad \varnothing, \varnothing \mapsto e_1 = e_2 : \text{boolean} \]

[non equality] \[ \varnothing, \varnothing \mapsto e_1 : T, \varnothing, \varnothing \mapsto e_2 : T \quad \varnothing, \varnothing \mapsto e_1 \neq e_2 : \text{boolean} \]

[less] \[ \varnothing, \varnothing \mapsto e_1 : T, \varnothing, \varnothing \mapsto e_2 : T \quad \varnothing, \varnothing \mapsto e_1 < e_2 : \text{boolean} \]

where \( T \in \{ \text{integer, real, string} \} \)

[less equal] \[ \varnothing, \varnothing \mapsto e_1 : T, \varnothing, \varnothing \mapsto e_2 : T \quad \varnothing, \varnothing \mapsto e_1 \leq e_2 : \text{boolean} \]

where \( T \in \{ \text{integer, real, string} \} \)

[greater] \[ \varnothing, \varnothing \mapsto e_1 : T, \varnothing, \varnothing \mapsto e_2 : T \quad \varnothing, \varnothing \mapsto e_1 > e_2 : \text{boolean} \]

where \( T \in \{ \text{integer, real, string} \} \)
[greater equal]  \[ e_1 : T \mid e_2 : T \]
\[ e_1 \geq e_2 : \text{boolean} \]

where \( T \in \{ \text{integer, real, string} \} \)

**Numeric Expression**

In the following type rules, \( T \in \{ \text{integer, real} \} \).

[plus]
\[ e_1 : T \mid e_2 : T \]
\[ e_1 + e_2 : T \]

[minus]
\[ e_1 : T \mid e_2 : T \]
\[ e_1 - e_2 : T \]

[add]
\[ e_1 : T \mid e_2 : T \]
\[ e_1 + e_2 : T \]

[subtract]
\[ e_1 : T \mid e_2 : T \]
\[ e_1 - e_2 : T \]

[times]
\[ e_1 : T \mid e_2 : T \]
\[ e_1 \times e_2 : T \]

[division]
\[ e_1 : \text{integer} \mid e_2 : \text{integer} \]
\[ e_1 \text{ div } e_2 : \text{integer} \]
[remainder] \[
\{ r \} \mapsto e_1 : \text{integer} \quad \{ r \} \mapsto e_2 : \text{integer} \\
\{ r \} \mapsto e_1 \text{ rem } e_2 : \text{integer}
\]

[real division] \[
\{ r \} \mapsto e_1 : \text{real} \quad \{ r \} \mapsto e_2 : \text{real} \\
\{ r \} \mapsto e_1 / e_2 : \text{real}
\]

**String Expression**

[concatenation] \[
\{ s \} \mapsto e_1 : \text{string} \quad \{ s \} \mapsto e_2 : \text{string} \\
\{ s \} \mapsto e_1 + e_2 : \text{string}
\]

[substring] \[
\{ s \} \mapsto e : \text{string} \\
\{ s \} \mapsto e_1 : \text{integer} \quad \{ s \} \mapsto e_2 : \text{integer} \\
\{ s \} \mapsto e \left( e_1 \mid e_2 \right) : \text{string}
\]

**Block**

[brackets] \[
\{ b \} \mapsto e : T \\
\{ b \} \mapsto (e) : T
\]

[{}\] \[
\{ b \} \mapsto e : T \\
\{ b \} \mapsto \{ e \} : T
\]

**Infinite Union**

[any injection] \[
\{ a \} \mapsto e : T \\
\{ a \} \mapsto \text{any}(e) : \text{any}
\]

[any projection]
\[ f, f' \mapsto e : \text{any} \quad T_1, ..., T_n \quad \text{Type} \quad \text{if} \{1...n\} (f, f') \mapsto e_i : T \]
\[ f, f' \mapsto \text{project} \ e \ \text{as} \ x \ \text{onto} \ \{\text{project\_list} ; \ \text{default} \ e_{n+1}\} : T \]

where \( \Box \) stands for \( \Box_1 : x, T_1 > : \Box_2 \) and

\[ \text{project\_list} \ \text{stands for} \ T_1 : e_1 ; \ldots ; T_n : e_n \]

**Abstraction**

[abstraction application]

\[ f, f' \mapsto e : \text{abstraction}[x_1 : T_1, \ldots, x_n : T_n] \quad f, f' \mapsto e_1 : T_1 \quad f, f' \mapsto e_n : T_n \]
\[ f, f' \mapsto e (x_1 = e_1, \ldots, x_n = e_n) : \text{behaviour} \]

[partial application]

\[ f, f' \mapsto e : \text{abstraction}[x_1 : T_1, \ldots, x_n : T_n] \quad f, f' \mapsto e_1 : T_1 \quad f, f' \mapsto e_j : T_j \]
\[ f, f' \mapsto e(x_1 = e_1, \ldots, x_j = e_j) : \text{abstraction}[T_{j+1}, \ldots, T_n] \]

where \( 1 < j < n \)

**Behaviour**

[connections]

\[ f, f' \mapsto e : \text{behaviour} \]
\[ f, f' \mapsto \text{connections}(e) : \text{sequence}[\text{any}] \]

[behaviour value]

\[ f, f' \mapsto e : \text{behaviour} \]
\[ f, f' \mapsto \text{behaviour} \ e : \text{behaviour} \]

**Identifier**

[id]

\[ f, f' :: x : T > : f' :: x : T \]
**View**

[view value] \( ... \) \( e : T \) \( ... \) \( e : T_n \)

\( \text{view}(l_1 = e_1, ..., l_n = e_n) : \text{view}[T_1, ..., T_n] \)

[view dereference] \( ... \) \( e : \text{view}[T_1, ..., T_n] \)

\( \text{view}[T_i] \)

**Location**

[location value] \( \text{loc}(e) : \text{loc}[T] \)

[location dereference] \( \text{loc}[T] \)

\( e : T \)

**Sequence**

[sequence value-1] \( e_1 : T \) \( ..., e_n : T \)

\( \text{sequence}[T] \)

[sequence value-2] \( e_1 : \text{integer} \)

\( e_2 : \text{integer} \)

\( e : \text{function}[\text{integer}] \rightarrow T \)

\( \text{sequence for } i = e_1 \text{ to } e_2 \text{ using } e : \text{sequence}[T] \)

[sequence index] \( e : \text{sequence}[T] \)

\( e : e_1 : T \)
[including]
\[
\ell.\ell \mapsto e : \text{sequence}[ T ]
\]
\[
\ell.\ell \mapsto e \quad \text{including} \quad e_1 : \text{sequence}[ T ]
\]

[excluding]
\[
\ell.\ell \mapsto e : \text{sequence}[ T ]
\]
\[
\ell.\ell \mapsto e \quad \text{excluding} \quad e_1 : \text{sequence}[ T ]
\]

**Function**

[function application]
\[
\ell.\ell \mapsto e : \text{function}[ T_1, \ldots, T_n ] \quad S \quad \ell.\ell \mapsto e_1 : T_1 \quad \ell.\ell \mapsto e_n : T_n
\]
\[
\ell.\ell \mapsto e(e_1, \ldots, e_n) : S
\]

**Literal**

[integer literal]
\[
i \in \text{INTEGER}
\]
\[
i : \text{integer}
\]

[real literal]
\[
r \in \text{REAL}
\]
\[
r : \text{real}
\]

[boolean literal]
\[
b \in \text{BOOLEAN}
\]
\[
b : \text{boolean}
\]

[string literal]
\[
s \in \text{STRING}
\]
\[
s : \text{string}
\]

[connection literal]
\[
T \in \text{BaseType}
\]
\[
\ell.\ell \mapsto \text{connection}(T) : \text{connection}[T]
\]
[behaviour literal] 
\[\text{done} : \text{behaviour}\]

[abstraction literal]
\[
\square_1, \square_2 : \square : \square_1 \vdash e : \text{behaviour}
\]
\[
\square_1, \square_2 \mapsto \text{abstraction}(x_1 : T_1, \ldots, x_n : T_n) \ ; \ e : \text{abstraction}[T_1, \ldots, T_n]
\]

[view literal]
\[
\square : \langle t, \text{view}[l_1 : T_1, \ldots, l_n : T_n] \rangle : \square \mapsto \text{nilview}(t) : \text{view}[l_1 : T_1, \ldots, l_n : T_n]
\]

[sequence literal]
\[
\square : \langle t, \text{sequence}[T] \rangle : \square \mapsto \text{nilsequence}(t) : \text{sequence}[T]
\]

[function literal]
\[
\square_1, \square_2 : \square : \square_1 \vdash e : S
\]
\[
\square_1, \square_2 \mapsto \text{function}(x_1 : T_1, \ldots, x_n : T_n) \ ; \ e : \text{function}[T_1, \ldots, T_n] \ ; S
\]

ARCHWARE ADL – RELEASE 1  
VERSION V1.1  
66