The ABS Language Specification

For ABS Version 1.2.0

April 22, 2013

Document Version 19890

Contents

1 Introduction **4**
- 1.1 Notation . . . 4

2 Lexical Structure **5**
- 2.1 Line Terminators and White Spaces . . . 5
- 2.2 Comments . . . 5
  - 2.2.1 End-Of-Line Comments . . . 5
  - 2.2.2 Traditional Comments . . . 5
- 2.3 Identifiers . . . 6
- 2.4 Keywords . . . 6
- 2.5 Literals . . . 6
- 2.6 Separators . . . 6
- 2.7 Operators . . . 7

3 Names and Types **8**
- 3.1 Names . . . 8
- 3.2 Types . . . 8
- 3.3 Type Synonyms . . . 8

4 Algebraic Data Types **10**
- 4.1 Parametric Data Types . . . 10
- 4.2 Predefined Data Types . . . 10
- 4.3 N-ary Constructors . . . 11
- 4.4 Abstract Data Types . . . 12

5 Functions **13**
- 5.1 Parametric Functions . . . 13

6 Pure Expressions **14**
- 6.1 Let Expressions . . . 14
- 6.2 Data Type Constructor Expressions . . . 15
- 6.3 Function Applications . . . 15
- 6.4 If-Then-Else Expression . . . 15
- 6.5 Case Expressions / Pattern Matching . . . 16
  - 6.5.1 Patterns . . . 16
  - 6.5.2 Type Checking . . . 18
- 6.6 Operator Expressions . . . 18

7 Expressions With Side Effects **20**
7.1 New Expression . . . 20
7.1.1 Standard Object Creation . . . 21
7.1.2 COG Object Creation . . . 21
7.2 Synchronous Call Expression . . . 21
7.3 Asynchronous Call Expression . . . 22
7.4 Get Expression . . . 22

8 Statements **23**
8.1 Block . . . 23
8.2 If Statement . . . 23
8.3 While Statement . . . 24
8.4 Variable Declaration Statements . . . 24
8.5 Assign Statement . . . 25
8.6 Await Statement . . . 25
8.7 Suspend Statement . . . 25
8.8 Skip Statement . . . 26
8.9 Assert Statement . . . 26
8.10 Return Statement . . . 26
8.11 Expression Statement . . . 27

9 Classes and Interfaces **28**
9.1 Interfaces . . . 28
9.2 Classes . . . 29
9.2.1 Active Classes . . . 30

10 Modules **32**
10.1 Exporting . . . 32
10.1.1 Exporting Everything . . . 33
10.2 Importing . . . 33
10.2.1 Unqualified Importing . . . 33
10.3 Exporting Imported Names . . . 33

11 Feature Modelling **35**
11.1 Feature Model . . . 35
11.1.1 Syntax . . . 35
11.1.2 Example . . . 36
11.1.3 Semantics . . . 37
11.2 Product Selection . . . 37
11.2.1 Syntax . . . 37
11.2.2 Example . . . 38

12 Delta Modules **39**
12.1 Syntax . . . 39
12.2 Object-oriented modifiers . . . 41
12.2.1 Interfaces . . . 41
12.2.2 Classes . . . 41

12.2.3 Methods 42
12.2.4 Class interfaces 44
12.2.5 Fields 44
12.3 Functional modifiers 44
12.3.1 Functions 45
12.3.2 Data types 45
12.3.3 Type synonyms 45
12.4 Module modifiers 45
12.4.1 Imports and Exports 46
12.5 Unsupported modifications 46

13 Software Product Line Configuration and Product Generation 47
13.1 Software Product Line Configuration 47
13.1.1 Syntax 47
13.1.2 Delta parameters 48
13.1.3 Application Conditions 49
13.1.4 Application Order of Delta Modules 49
13.1.5 Configuration Example 49
13.2 Product Generation 50
13.2.1 Example 50

14 Annotations 52
14.1 Type Annotations 52

15 Models 53

A ABS Standard Library 56

Chapter 1

Introduction

This chapter describes the core ABS language as it is implemented in the ABS tools. The ABS language is a class-based object-oriented language that features algebraic data types and side effect-free functions. Syntactically, the ABS language tries to be as close as possible to the Java language [4] so that programmers that are used to Java can easily use the ABS language without much learning effort.

1.1 Notation

In this chapter we often present the concrete syntax of the ABS language. To do so we use BNF[1] with the following denotations.

- $[x]$ denotes zero or one occurrence of x.
  The same notation is used to represent an optional element in the formal system. In effect $[x]$ corresponds to either nothing or an element of x.
- x^* denotes zero or more occurrences of x.
  Note that in formal semantics the notation $\overline{x}$ is used to represent an explicit sequence of elements $x_1, \ldots, x_n$; similarly $\overline{x} : \overline{T}$ represents $x_1 : T_1, \ldots, x_n : T_n$, following Pierce [10].
- x^+ denotes one or more occurrences of x.
- $x \mid y$ means one of either x or y.
- $[: x :]$ denotes the POSIX character class x.
- text in `monospace` denotes terminal symbols.
- text in *italics* denotes non-terminals in the grammar.

[1] Backus-Naur Form

Chapter 2

Lexical Structure

This section describes the lexical structure of the ABS language. ABS programs are written in Unicode.[1]

2.1 Line Terminators and White Spaces

Line terminators and white spaces are defined as in Java.

Syntax:

LineTerminator	::=	`\n` | `\r` | `rn`
WhiteSpace	::=	*LineTerminator* | `␣` | `\t` | `\f`

2.2 Comments

Comments are code fragments that are completely ignored and have no semantics in the ABS language. ABS supports two styles of comments: *end-of-line comments* and *traditional comments*.

2.2.1 End-Of-Line Comments

An end-of-line comment is a code fragment that starts with two slashes, e.g., `// text`. All text that follows `//` until the end of the line is treated as a comment.

Example:

```
// this is a comment
module A; // this is also a comment
```

2.2.2 Traditional Comments

A traditional comment is a code fragment that is enclosed in /* */, e.g., /* this is a comment */. Nested traditional comments are not possible.

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

Example:

```
/* this
is a multiline
comment */
```

2.3 Identifiers

ABS distinguishes *identifier* and *type identifier*. They differ in the first character, which must be a lower-case character for identifiers and an upper-case character for type identifiers.

Syntax:

Identifier	::=	[:lower:] ([:alpha:] | [:digit:] | _)*
TypeId	::=	[:upper:] ([:alpha:] | [:digit:] | _)*

2.4 Keywords

The following words are keywords in the ABS language and are *not* regarded as identifiers.

```
adds          after        assert    await       builtin  case
cog           core         class     data        def      delta
else          export       features  from        get      hasField
hasInterface  hasMethod    if        implements  import   in
interface     let          modifies  module      new      null
product       productline  removes   return      skip     suspend
this          type         when      while
```

2.5 Literals

A *literal* is a textual representation of a value. ABS supports three kinds of literals, *integer literals*, *string literals*, and the *null literal*.

Syntax:

Literal	::=	*IntLiteral* | *StringLiteral* | *NullLiteral*
IntLiteral	::=	0 | [1-9][0-9]*
StringLiteral	::=	" *StringCharacter** "
NullLiteral	::=	`null`

Where a *StringCharacter* is defined as in the Java language [4, p. 28]

2.6 Separators

The following characters are *separators*:

```
( ) { } [ ] , ; :
```

2.7 Operators

The following tokens are *operators*:

```
||   &&   ==   !=   <   >   <=   >=   +   -   *   /   %   ~   &
```

Chapter 3

Names and Types

3.1 Names

A *name* in ABS can either be a simple identifier as described above, or can be qualified with a type name, which represents a module.

Syntax:

TypeName	::=	*TypeId* (. *TypeId*)*
Name	::=	*Identifier* ∣ *TypeName* . *Name*

Examples for syntactically valid names are: `head`, `x`, `ABS.StdLib.tail`. Examples for type names are: `Unit`, `X`, `ABS.StdLib.Map`.

3.2 Types

Types in ABS are either plain type names or can have type arguments.

Syntax:

Type	::=	*TypeName* [*TypeArgs*]
TypeArgs	::=	< *TypeList* >
TypeList	::=	*Type* (, *Type*)*

Where `TypeName` can refer to a data type, an interface, a type synonym, and a type parameter. Note that classes cannot be used as types in ABS. In addition, only parametric data types can have type arguments. Examples for syntactically valid types are: `Bool`, `ABS.StdLib.Int`, `List<Bool>`, `ABS.StdLib.Map<Int,Bool>`.

3.3 Type Synonyms

Type Synonyms define synonyms for otherwise defined types. Type synonyms start with an uppercase letter.

Syntax:

TypeSynDecl ::= *TypeId* = *TypeName* ;

Example:

```
type Filename = String
type Filenames = Set<Filename>
type Servername = String
type Packet = String
type File =  List<Packet>
type Catalog = List<Pair<Servername,Filenames>>
```

Chapter 4

Algebraic Data Types

Algebraic Data Types make it possible to describe data in an immutable way. In contrast to objects, data types do not have an identify and cannot be mutated. This makes reasoning about data types much simpler than about objects. Data types are built by using *Data Type Constructors* (or *constructors* for short), which describe the possible values of a data type.

Syntax:

DataTypeDecl	::=	`data` *TypeId* [*TypeParams*] [= *DataConstrList*] `;`
TypeParams	::=	`<` *TypeId* (`,` *TypeId*)* `>`
DataConstrList	::=	*DataConstr* (`|` *DataConstr*)*
DataConstr	::=	*TypeId* [`(` [*TypeList*] `)`]

Example:

```
data IntList = NoInt | Cons(Int, IntList);
data Bool = True | False;
```

4.1 Parametric Data Types

Parametric Data Types are useful to define general-purpose data types, such as lists, sets or maps. Parametric data types are declared like normal data types but have an additional *type parameter* section inside broken brackets (< >) after the data type name.

Example:

```
data List<A> = Nil | Cons(A, List<A>);
```

4.2 Predefined Data Types

The following data types are predefined:

- data Bool = True | False;. The boolean type with constructors True and False and the usual Boolean infix and prefix operators.
- data Unit = Unit;. The unit type with only one constructor Unit (for methods without return values).
- data Int;. An arbitrary integer ($\mathbb{Z}$) for which values are constructed by using integer literals and arithmetic expressions.
- data Rat;. A rational number ($\mathbb{Q}$). Rational values are obtained via the division (/) operator and have arbitrary precision. Assigning rational values to variables of type Int, either explicitly or implicitly by passing them to a function or method expecting an integer, rounds towards zero.
- data String;. A string for which values are constructed by using string literals and operators.
- data Fut<T>;. Representing a future. A future cannot be explicitly constructed, but it is the result of an asynchronous method call. The value of a future an only be obtained by using the get expression (Sec. 7.4).
- data List<A> = Nil | Cons(A, List<A>), with constructors Nil and Cons(A, List<A>). This predefined data type is used for implementing arbitrary n-ary constructors (see below).

A complete list of predefined data types is contained in Appendix A which lists the ABS Standard Library.

4.3 N-ary Constructors

For data types of arbitrary size, like lists, maps and sets, it is undesirable having to write them down in the form of nested constructor expressions. For this purpose, ABS provides a special syntax for *n-ary constructors*, which are transformed into constructor expressions via a user-supplied function.

Example:

```
data Set<A> = EmptySet | Insert(A, Set<A>);
def Set<A> set<A>(List<A> l) =
   case l {
      Nil => EmptySet;
      Cons(hd, tl) => Insert(hd, set(tl));
   } ;

{
  Set<Int> s = set[1, 2, 3];
}
```

An expression *type[parameters*]* is transformed into a literal by handing it to a function named *type* which takes one parameter of type *List* and returns an expression of type *Type*. (It is desirable, although not currently enforced, that *type* and *Type* are the same word, just with different capitalization.)

4.4 Abstract Data Types

Using the module system (cf. Sec. 10) it is possible to define *abstract data types*. For an abstract data type, only the functions that operate on them are known to the client, but not its constructors. This can be easily realized in ABS by putting such a data type in its own module and by only exporting the data type and its functions, without exporting the constructors.

Chapter 5

Functions

Functions in ABS define names for parametrized data expressions. A Function in ABS is always side effect-free, which means that it cannot manipulate the heap.

Syntax:

FunctionDecl	::=	`def` Type *Identifier* [`<` *TypeIdList* `>`] `(` *ParamList* `)` `=` *FunBody* `;`
FunBody	::=	`builtin` | *PureExp*
TypeIdList	::=	*TypeId* (`,` *TypeId*)*

Example:

```
def Int length(IntList list) =
  case list {
    Nil => 0;
    Cons(n, ls) => 1 + length(ls);
  };
```

5.1 Parametric Functions

Parametric Functions allow to work with parametric data types in a general way. For example, given a list of any type, a parametric function `head` can return the first element, regardless of its type. Parametric functions are defined like normal functions but have an additional `type parameter` section inside angle brackets (`< >`) after the function name.

Example:

```
def A head<A>(List<A> list) =
  case list {
    Cons(x, xs) => x;
  };
```

Chapter 6

Pure Expressions

Pure Expressions are side effect-free expressions. This means that these expressions cannot modify the heap.

Syntax:

PureExp	::=	*Variable*
	|	*FieldAccess*
	|	*ThisExp*
	|	*NullLiteral*
	|	*LetExp*
	|	*DataConstrExp*
	|	*FnAppExp*
	|	*FnAppListExp*
	|	*IfExp*
	|	*CaseExp*
	|	*OperatorExp*
	|	`(` *PureExp* `)`
Variable	::=	*Identifier*
FieldAccess	::=	*ThisExp* `.` *Identifier*
ThisExp	::=	`this`
PureExpList	::=	*PureExp* (`,` *PureExp*)*

6.1 Let Expressions

Let Expressions bind variable names to pure expressions.

Syntax:

LetExp	::=	`let` `(` *Param* `)` `=` *PureExp* `in` *PureExp*

Example:

```
let (Bool x) = True in ~x
```

6.2 Data Type Constructor Expressions

Data Type Constructor Expressions are expressions that create data type values by using data type constructors. Note that for data type constructors that have no parameters, the parentheses are optional.

Syntax:

DataConstrExp ::= *TypeName*
| *TypeName* `(` [*PureExpList*] `)`

Example:

```
True
Cons(True, Nil)
ABS.StdLib.Nil
```

6.3 Function Applications

Function Applications apply functions to arguments.

Syntax:

FnAppExp ::= *Name* `(` [*PureExpList*] `)`

Example:

```
tail(Cons(True, Nil))
ABS.StdLib.head(list)
```

6.4 If-Then-Else Expression

ABS has a standard if-then-else expression.

Syntax:

IfExp ::= `if` *PureExp* `then` *PureExp* `else` *PureExp*

Example:

```
if 5 == 4 then True else False
```

6.5 Case Expressions / Pattern Matching

ABS supports pattern matching by the *Case Expression*. It takes an expression as first argument, which a series of patterns is matched against. The value of the case expression itself is the value of the expression on the right-hand side of the first matching expression. It is an error if no pattern matches the expression.

Syntax:

CaseExp	::=	`case` *PureExp* `{` *CaseBranch** `}`
CaseBranch	::=	*Pattern* `=>` *PureExp* `;`
Pattern	::=	*Identifier*
	|	*Literal*
	|	*ConstrPattern*
	|	`_`
ConstrPattern	::=	*TypeName* [`(` [*PatternList*] `)`]
PatternList	::=	*Pattern* (`,` *Pattern*)*

6.5.1 Patterns

There are five different kinds of patterns available in ABS:

- Pattern Variables (e.g., `x`, where `x` is not bound yet)
- Bound Variables (e.g., `x`, where `x` is bound)
- Literal Patterns (e.g., `5`)
- Data Constructor Patterns (e.g., `Cons(Nil,x)`)
- Underscore Pattern (`_`)

Pattern Variables

Pattern variables are simply unbound variables. Like the underscore pattern, these variables match every value, but, in addition, bind the variable to the matched value. The bound variable can then be used in the right-hand-side expression of the corresponding branch. Typically, pattern variables are used inside of data constructor patterns to extract values from data constructors. For example:

```
def A fromJust<A>(Maybe<A> a) =
  case a {
    Just(x) => x;
  };
```

Bound Variables

If a bound variable is used as a pattern, the pattern matches if the value of the case expression is equal to the value of the bound variable.

```
def Bool contains<A>(List<A> list, A value) =
  case list {
    Nil => False;
    Cons(value, _) => True;
    Cons(_, rest) => contains(rest, value);
  };
```

Literal Patterns

Literals can be used as patterns. This is similar to bound variables, because the pattern matches if the value of the case expression is equal to the literal value.

```
def Bool isEmpty(String s) =
  case b {
    "" => True;
    _  => False;
  };
```

Data Constructor Patterns

A data constructor pattern is like a standard data constructor expression, but where certain sub expressions can be patterns again.

```
def Bool negate(Bool b) =
  case b {
    True => False;
    False => True;
  };
```

```
def List<A> remainder(List<A> list) =
  case b {
    Cons(_, rest) => rest;
  };
```

Underscore Pattern

The underscore pattern (_) simply matches every value. It is generally used as the last pattern in a case expression to define a default case. For example:

```
def Bool isNil<A>(List<A> list) =
  case list {
    Nil => True;
```

```
    _ => False;
  };
```

6.5.2 Type Checking

A case expression is type-correct if and only if all its expressions and all its branches are type-correct and the right-hand side of all branches have a common super type. This common super type is also the type of the overall case expression.

A branch (a pattern and its expression) is type-correct if its pattern and its right-hand side expression are type-correct. A pattern is type-correct if it can match the corresponding case expression.

6.6 Operator Expressions

ABS has a number of predefined operators which can be used to form *Operator Expressions*.

Syntax:

OperatorExp	::=	*UnaryExp* | *BinaryExp*
UnaryExp	::=	*UnaryOp PureExp*
UnaryOp	::=	`␣` | `-`
BinaryExp	::=	*PureExp BinaryOp PureExp*
BinaryOp	::=	`==` | `!=` | `<` | `<=` | `>` | `>=` | `+` | `-` | `*` | `/` | `%`

Table 6.1 describes the meaning as well as the associativity and the precedence of the different operators. They are grouped according to precedence, as indicated by horizontal rules, from low precedence to high precedence.

Expression	Meaning	Associativity	Argument types	Result type
`e1 || e2`	logical or	left	Bool, Bool	Bool
`e1 && e2`	logical and	left	Bool, Bool	Bool
`e1 == e2`	equality	left	compatible	Bool
`e1 != e2`	inequality	left	compatible	Bool
`e1 < e2`	less than	left	number, number	Bool
`e1 <= e2`	less than or equal to	left	number, number	Bool
`e1 > e2`	greater than	left	number, number	Bool
`e1 >= e2`	greater than or equal to	left	number, number	Bool
`e1 + e2`	concatenation	left	String, String	String
`e1 + e2`	addition	left	number, number	number
`e1 - e2`	subtraction	left	number, number	number
`e1 * e2`	multiplication	left	number, number	number
`e1 / e2`	division	left	number, number	Rat
`e1 % e2`	modulo	left	number, number	Int
`~ e`	logical negation	right	Bool	Bool
`- e`	integer negation	right	number	number

Table 6.1: Operator expressions, grouped according to precedence from low to high. “number” is either Int or Rat.

Chapter 7

Expressions With Side Effects

Beside pure expressions, ABS has expressions with side effects. However, these expressions are defined in such a way that they can only have a single side effect. This means that subexpressions of expressions can only be pure expressions again. This restriction simplifies the reasoning about ABS expressions.

Syntax:

Exp	::=	*PureExp* | *EffExp*
EffExp	::=	*NewExp*
	|	*SyncCall*
	|	*AsyncCall*
	|	*GetExp*

7.1 New Expression

A *New Expression* creates a new object from a class name and a list of arguments. In ABS objects can be created in two different ways. Either they are created in the current COG, using the standard `new` expression, or they are created in a new COG by using the `new cog` expression.

Syntax:

NewExp ::= `new` [`cog`] *TypeName* (*PureExpList*)

Example:

```
new Foo(5)
new cog Bar()
```

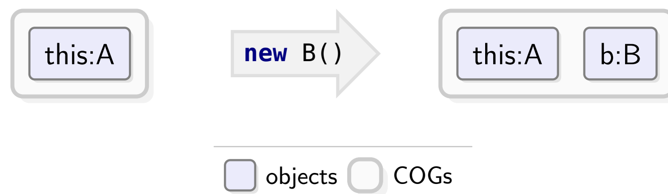

Figure 7.1: Process of creating an object inside the current COG by using the standard **new** expression.

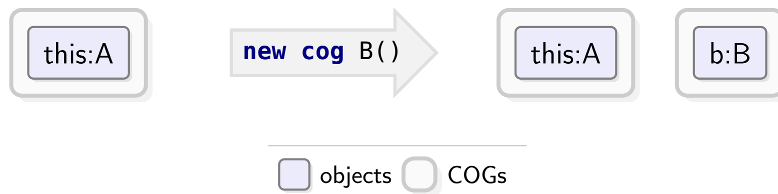

Figure 7.2: Process of creating an object in a new COG by using the **new cog** expression.

7.1.1 Standard Object Creation

When using the standard **new** expression, the new object is created in the *current* COG, i.e., the COG of the current receiver object. Figure 7.1 illustrates this by showing two different runtime states, one before the creation of an object `b` and one after its creation.

7.1.2 COG Object Creation

The concurrency model of ABS is based on the notion of COGs [7]. An ABS system at runtime is a set of concurrently running COGs. A COGs can be seen as an isolated subsystem, which has its own state (an object-heap) and its own internal behavior. COGs are created implicitly when creating a new object by using the **new cog** expression. Figure 7.2 illustrates this by showing two different runtime states, one before the creation of an object `b` using the **new cog** expression and one after its creation. In the second runtime state, two COGs exists.

7.2 Synchronous Call Expression

A *Synchronous Call* consists of a target expression, a method name, and a list of argument expressions.

Syntax:

SyncCall ::= *PureExp* . *Identifier* (*PureExpList*)

Example:

```
Bool b = x.m(5);
```

7.3 Asynchronous Call Expression

An *Asynchronous Call* consists of a target expression, a method name, and a list of argument expressions. Instead of directly invoking the method, an asynchronous method call creates a new *task* in the target COG, which is executed asynchronously. This means that the calling task proceeds independently after the call, without waiting for the result [7]. The result of an asynchronous method call is a future (`Fut<V>`), which can be used by the calling task to later obtain the result of the method call. That future is *resolved* by the task that has been created in the target COG to execute the method.

Syntax:

AsyncCall ::= *PureExp* `!` *Identifier* `(` *PureExpList* `)`

Example:

```
Fut<Bool> f = x!m(5);
```

7.4 Get Expression

A *Get Expression* is used to obtain the value from a future. The current task is blocked until the value of the future is available, i.e., until the future has been resolved. No other task in the COG can be activated in the meantime [7].

Syntax:

GetExp ::= *PureExp* `.` `get`

Example:

```
Bool b = f.get;
```

Chapter 8

Statements

In contrast to expressions, *Statements* in ABS are not evaluated to a value. If one wants to assign a value to statements it would be the `Unit` value.

Syntax:

Statement	::=	*CompoundStmt*
	|	*VarDeclStmt*
	|	*AssignStmt*
	|	*AwaitStmt*
	|	*SuspendStmt*
	|	*SkipStmt*
	|	*AssertStmt*
	|	*ReturnStmt*
	|	*ExpStmt*
CompoundStmt	::=	*Block*
	|	*IfStmt*
	|	*WhileStmt*

8.1 Block

A block consists of a sequence of statements and defines a name scope for variables.

Syntax:

Block ::= { *Statement** }

8.2 If Statement

Syntax:

IfStmt ::= `if` `(` *PureExp* `)` *Stmt* [`else` *Stmt*]

Example:

```
if (5 < x) {
  y = 6;
} else {
  y = 7;
}

if (True)
  x = 5;
```

8.3 While Statement

Syntax:

`while` `(` *PureExp* `)` *Stmt*

Example:

```
while (x < 5)
   x = x + 1;
```

8.4 Variable Declaration Statements

A variable declaration statement is used to declare variables.

Syntax:

VarDeclStmt ::= *TypeName Identifier* [= *Exp*] `;`

A variable has an optional *initialization expression* for defining the initial value of the variable. The initialization expression is *mandatory* for variables of data types. It can be left out only for variables of reference types, in which case the variable is initialized with **`null`**.

Example:

```
Bool b = True;
```

8.5 Assign Statement

The *Assign Statement* assigns a value to a variable or a field.

Syntax:

AssignStmt	::=	*Variable* = *PureExp* ;
	|	*FieldAccess* = *PureExp* ;

Example:

```
this.f = True;
x = 5;
```

8.6 Await Statement

Await Statements suspend the current task until the given *guard* is true [7]. The task will not be suspended if the guard is already initially true. While the task is suspended, other tasks within the same COG can be activated. Await statements are also called *scheduling points*, because they are the only source positions, where a task may become suspended and other tasks of the same COG can be activated.

Syntax:

AwaitStmt	::=	`await` *Guard* ;
Guard	::=	*ClaimGuard*
	|	*PureExp*
	|	*Guard* & *Guard*
ClaimGuard	::=	*Variable* ?
	|	*FieldAccess* ?

Example:

```
Fut<Bool> f = x!m();
await f?;
await this.x == True;
await f? & this.y > 5;
```

8.7 Suspend Statement

A *Suspend Statement* causes the current task to be suspended.

Syntax:

SuspendStmt ::= `suspend` ;

Example:

```
suspend;
```

8.8 Skip Statement

The *Skip Statement* is a statement that does nothing.

Syntax:

SkipStmt ::= `skip` ;

8.9 Assert Statement

An *Assert Statement* is a statement for asserting certain conditions.

Syntax:

AssertStmt ::= `assert` *PureExp* ;

Example:

```
assert x != null;
```

8.10 Return Statement

A *Return Statement* defines the return value of a method. A return statement can only appear as a last statement in a method body.

Syntax:

ReturnStmt ::= `return` *PureExp* ;

Example:

```
return x;
```

8.11 Expression Statement

An *Expression Statement* is a statement that only consists of a single expression. Such statements are only executed for the effect of the expression.

Syntax:

ExpStmt ::= *Exp* `;`

Example:

```
new C(x);
```

Chapter 9

Classes and Interfaces

Objects in ABS are built from *classes*, which implement *interfaces*. Only interfaces can be used as types in ABS.

9.1 Interfaces

Interfaces in ABS are similar to interfaces in Java. They have a name, which defines a nominal type, and they can *extend* arbitrary many other interfaces. The interface body consists of a list of method signature declarations. Method names start with a lowercase letter.

Syntax:

InterfaceDecl	::=	`interface` *TypeId* [`extends` *TypeName* (`,` *TypeName*)*] `{` *MethSig** `}`
MethSig	::=	*Type Identifier* `(` [*ParamList*] `)` `;`
ParamList	::=	*Param* (`,` *Param*)*
Param	::=	*Type Identifier*

The interfaces in the example below represent a database system, providing functionality to store and retrieve files, and a node of a peer-to-peer file sharing system. Each node of a peer-to-peer system plays both the role of a server and a client. The data types are defined in the ABS standard library, included in Appendix A, and the remainder types are type synonyms included in Section 3.3.

Example:

```
interface DB {
  File getFile(Filename fId);
  Int getLength(Filename fId);
  Unit storeFile(Filename fId, File file);
  Filenames listFiles();
}

interface Client {
  List<Pair<Server,Filenames>> availFiles(List<Server> sList);
```

```
  Unit reqFile(Server sId, Filename fId);
}

interface Server {
  Filenames inquire();
  Int getLength(Filename fId);
  Packet getPack(Filename fId, Int pNbr);
}

interface Peer extends Client, Server {
  List<Server> getNeighbors();
}
```

9.2 Classes

Like in typical class-based languages, classes in ABS are used to create objects. Classes can implement an arbitrary number of interfaces. ABS does not support inheritance, as code reuse in ABS is realized by delta modules (see Chapter 12). Classes do not have constructors in ABS but instead have *class parameters* and an optional *init block*. Class parameters actually define additional fields of the class that can be used like any other declared field.

Syntax:

ClassDecl	::=	*class TypeId* [(*ParamList*)] [implements *TypeName* (, *TypeName*)*] { [*FieldDeclList*] [*Block*] [*MethDeclList*] }
FieldDeclList	::=	*FieldDecl* (, *FieldDecl*)*
FieldDecl	::=	*TypeId Identifier* [= *PureExp*] ;
MethDeclList	::=	*MethDecl* (, *MethDecl*)*
MethDecl	::=	*Type Identifier* (*ParamList*) *Block*

We continue the peer-to-peer example with an implementation of the `DB` interface, and the signature of a node that implements the `Peer` interface.

Example:

```
class DataBase(Map<Filename,File> db) implements DB {
  File getFile(Filename fId) {
      return lookup(db, fId);
  }

  Int getLength(Filename fId){
      return length(lookup(db, fId));
  }

  Unit storeFile(Filename fId, File file) {
      db = insert(Pair(fId,file), db);
  }

  Filenames listFiles() {
      return keys(db);
  }
}

class Node(DB db, Peer admin, Filename file) implements Peer {
  Catalog catalog;
  List<Server> myNeighbors;

  // implementation...
}
```

9.2.1 Active Classes

A class can be *active* or *passive*. Active classes start an activity on their own upon creation. Passive classes only react to incoming method calls. A class is active if and only if it has a *run method*:

Example:

```
Unit run() {
  // active behavior ...
}
```

The run method is called after object initialization.

Chapter 10

Modules

For name spacing, code structuring, and code hiding purposes, ABS offers a module system. The module system of ABS is very similar to that of Haskell [9]. It uses, however, a different syntax that is similar to that of Java [4] and Python.

Syntax:

ModuleDecl	::=	`module` *TypeName* `;` [*ExportList*] [*ImportList*] *Decl** [*Block*]
ExportList	::=	*Export* (`,` *Export*)*
ImportList	::=	*Import* (`,` *Import*)*
Export	::=	`export` *AnyNameList* [`from` *TypeName*] `;`
	|	`export` $*$ [`from` *TypeName*] `;`
Import	::=	`import` *AnyNameList* [`from` *TypeName*] `;`
	|	`import` $*$ `from` *TypeName* `;`
AnyNameList	::=	*AnyName* [`,` *AnyName*]
AnyName	::=	*Name* | *TypeName*
Decl	::=	*FunDecl* | *TypeSynDecl* | *DataTypeDecl*
	|	*InterfaceDecl* | *ClassDecl* | *DeltaDecl*

A module with name `MyModule` is declared by writing

```
module MyModule;
```

This declaration introduces a new module name `MyModule` which can be used to qualify names. All declarations which follow this statement belong to the module `MyModule`. A module name is a type name and must always start with an upper case letter.

10.1 Exporting

By default, modules do not export any names. In order to make names of a module usable to other modules, the names have to be *exported*. Exporting is done by writing one or several *exports* after the module declaration. For example, to export a data type and a data constructor, one can write something like this:

```
module Drinks;
export Drink, Milk;
```

```
data Drink = Milk | Water;
```

Note that in this example, the data constructor `Water` is not exported, and can thus not be used by other modules. By only exporting the data type without any of its constructors one can realize *abstract data types* (cf. Section 4.4).

10.1.1 Exporting Everything

Sometimes it is required to export everything from a module. This can be achieved by writing:

```
export *;
```

In this case, all names that are *defined* in the module are exported, in particular, this means that imported names are *not* exported.

10.2 Importing

In order to use exported names of a module in another module, the names have to be *imported*. After the list of export statements follows an optional list of *imports*, which are used to import names from other modules. For example, to write a module that imports the `Drink` data type of the module `Drinks` one can write:

```
module Bar;
import Drinks.Drink;
```

After a name has been imported, it can be used inside the module in a fully qualified way.

10.2.1 Unqualified Importing

To use a name from another module in an unqualified way requires an *unqualified import*. For example, to use the `Milk` data constructor inside the `Bar` module, without having to qualify it with the `Drinks` module each time, the following unqualified import statement is used:

```
module Bar;
import Milk from Drinks;
```

Note that this kind of import also imports the qualified names. So in this example the names `Milk` and `Drinks.Milk` can be used inside the module `Bar`.

To use all exported names from another module in an unqualified way one can write:

```
import * from SomeModule;
```

10.3 Exporting Imported Names

It is possible to export names that have been imported. For example,

```
module Bar;
export Drink;
import * from Drinks;
```

exports data type `Drink` that has been imported from `Drinks`

To export all names imported from a certain module one can write

```
export * from SomeModule;
```

In this case, all names that have been imported from module `SomeModule` are exported. For example,

```
module Bar;
export * from Drinks;
import * from Drinks;
```

exports all names that are exported by module `Drinks`.

However, in this example:

```
module Bar;
export * from Drinks;
import Drink from Drinks;
```

only `Drink` is exported as this is the only name imported from module `Drinks`. Note: only names that are visible in a module can be exported by that module.

To only export some names from a certain module one can write, for example:

```
module Bar;
export Drink from Drinks;
import * from Drinks;
```

This only exports `Drink` from module `Drinks`.

Chapter 11

Feature Modelling

ABS provides language constructs and tools for modelling variable systems following Software Product Line (SPL) [11] engineering practices. The *Micro Textual Variability Language* μTVL [2] is the part of ABS used to model all products of an SPL by using features and feature attributes. A *Product Selection* identifies individual products that are of particular interest to the project. Section 11.1 covers feature modelling with μTVL and Section 11.2 describes how to specify product selections.

11.1 Feature Model

As part of the requirements engineering process, software variability is commonly expressed using *features*. Features are organised in a *feature model* [1, 8], which is essentially a set of logical constraints expressing the dependencies between features. Thus the feature model defines a set of legal feature combinations. These represent the set of valid software products that can be built from the given features.

11.1.1 Syntax

The grammar of μTVL is given in Figure 11.1. Assume the presence of two global sets: FID of feature names and AID of attribute names. Names in FID follow type identifier syntax, while names in AID follow identifier syntax (cf. Chapter 2).

Attributes and values in μTVL range either over integers or booleans. The *Model* clause specifies a number of 'orthogonal' root feature models along with a number of extensions that specify additional constraints, typically cross-tree dependencies. The *FeatureDecl* clause specifies the details of a given feature, firstly by giving it a name (FID), followed by a number of possibly optional sub-features, the feature's attributes and any relevant constraints. The *FeatureExtension* clause specifies additional constraints and attributes for a feature. This is particularly useful for specifying constraints that do not fit into the tree structure given by the root feature model. The *Cardinality* clause describes the number of elements of a group that may appear in a result. The *AttributeDecl* clause specifies the declaration of both integer (bounded or unbounded) and boolean attributes of features.

The *Constraint* clause specifies constraints on the presence of features and on attributes. An `ifin` constraint is only applicable if the current feature is selected. Similarly, an `ifout` constraint is only applicable if the current feature is not selected. A `require` clause specifies that the current

```
Model            ::=  (root FeatureDecl)* FeatureExtension*

FeatureDecl      ::=  FID [{ [Group] AttributeDecl* Constraint* }]
FeatureExtension ::=  extension FID { AttributeDecl* Constraint*}

Group            ::=  group Cardinality { [opt] FeatureDecl, ([opt] FeatureDecl)* }
Cardinality      ::=  allof | oneof | [n1 .. *] | [n1 .. n2]
AttributeDecl    ::=  Int AID ; | Int AID in [ Limit .. Limit ] ; | Bool AID ;
Limit            ::=  n | *

Constraint       ::=  Expr ; | ifin: Expr ; | ifout: Expr ;
                   |  require: FID ; | exclude: FID ;
Expr             ::=  True | False | n | FID | AID | FID.AID
                   |  UnOp Expr | Expr BinOp Expr | ( Expr )
UnOp             ::=  ! | -
BinOp            ::=  || | && | -> | <-> | == | != | > | < | >= | <= | + | - | * | / | %
```

Figure 11.1: Grammar of μTVL, the feature modelling language of ABS

feature requires some other feature, whereas `exclude` expresses the mutual incompatibility between the current feature and some other feature. The *Expr* clause expresses a boolean constraint over the presence of features and attributes, using standard boolean and arithmetic operators. Features are referred to by identity (FID). Attributes are referred to either using an unqualified name (AID), for in scope attributes, or using a qualified name (FID.AID) for attributes of other features.

11.1.2 Example

Feature models are often represented graphically as feature diagrams such as the one shown in Figure 11.2. It represents a multi-lingual "Hello World" product line, which describes a range of software products that can output "Hello World" in some particular language some number of times. Below we show how the feature model underlying the feature diagram in Figure 11.2 is

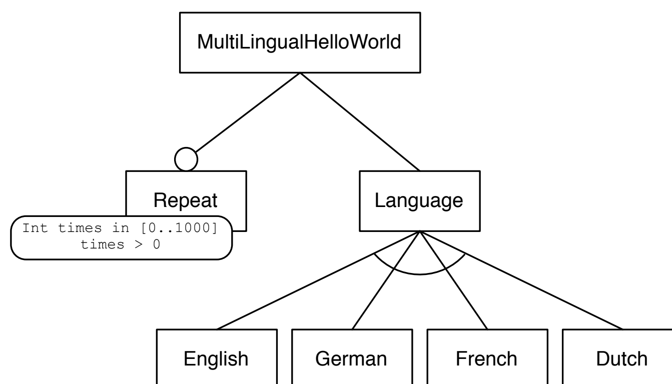

Figure 11.2: Example feature diagram for a "Hello World" software product line

encoded in ABS using the textual variability language μTVL.

```
root MultiLingualHelloWorld {
  group allof {
```

```
    Language {
      group oneof { English, Dutch, German, French }
    },
    opt Repeat {
      Int times in [0..1000];
      ifin: times > 0;
    }
  }
}
extension English {
  ifin: Repeat -> (Repeat.times >= 2 && Repeat.times <= 5);
}
```

The "Hello World" product line in this example has two main features, *Language* and *Repeat*, under the root feature and joined with the `allof` combinator. The *Language* feature requires one out of four possible features: *English*, *Dutch*, *French*, or *German*. The *Repeat* feature is optional, it has no associated sub-features, and it has an attribute *times* which ranges between 0 and 1000, with an added condition that it must be strictly greater than 0. In this example an extension for the *English* feature is given. When the *English* and the *Repeat* features are present, the attribute `times` must be between 2 and 5, inclusive.

11.1.3 Semantics

The semantics of μTVL is given by integer constraints, that is, a feature model is encoded as a constraint over features and feature attributes. Each solution for this constraint represents a valid product of the feature model.

11.2 Product Selection

ABS allows the developer to name products that are of particular interest, in order to easily refer to them later when the actual code needs to be generated. A product definition states which features are to be included in the product and sets attributes of those features to concrete values.

11.2.1 Syntax

Figure 11.3 shows the grammar of the ABS product selection language. The *Selection* clause specifies a product by giving it a name, by stating the features and optional attribute assignments that are included in that product.

$$
\begin{array}{rcl}
\textit{Selection} & ::= & \texttt{product}\ \textit{TypeId}\ (\ \textit{FeatureSpecs}\ )\ ; \\
\textit{FeatureSpecs} & ::= & \textit{FeatureSpec}\ (,\ \textit{FeatureSpec})^{*} \\
\textit{FeatureSpec} & ::= & \mathsf{FID}\ [\textit{AttributeAssignments}] \\
\textit{AttributeAssignments} & ::= & \{\ \textit{AttributeAssignment}\ (,\ \textit{AttributeAssignment})^{*}\ \} \\
\textit{AttributeAssignment} & ::= & \mathsf{AID} = \textit{Literal}
\end{array}
$$

Figure 11.3: Grammar of the product selection language

11.2.2 Example

A valid product from the example "Hello World" product line can be defined as follows.

```
product P1 (English);
```

This denotes the inclusion of a single feature, `English`. Implicitly, its parent node is also selected. The product `P2` defined below includes the features `Dutch` and `Repeat`, where `Repeat`'s attribute `times` is initialised to 3. To select features with attributes, we need to include an assignment of attribute variables. ABS currently supports boolean and integer feature attributes.

```
product P2 (Dutch, Repeat{times=3});
```

Below is an *invalid* product definition, that is, one that does not satisfy the feature model, as the `Repeat.times` attribute is assigned a value (2222) outside the valid range defined by the feature model (`[0..1000]`). The constraint checker that comes with ABS can evaluate all product selections with respect to the feature model and warn about invalid products.

```
product P3 (English, Repeat{times=2222});
```

Chapter 12

Delta Modules

ABS implements the delta-oriented programming model [12], an approach that aids the development of a set of programs simultaneously from a single code base, following the software product line engineering approach [11]. In delta-oriented programming, features defined by a feature model are associated with code modules that describe modifications to a core program. In ABS, these modules are called *delta modules*. Hence the implementation of a software product line in ABS is divided into a *core* and a set of delta modules.

The core consists of a set of ABS modules containing the classes that implement a complete software product of the corresponding software product line. Delta modules (or *deltas* in short) describe how to change the core program to obtain new products. This includes adding new classes and interfaces, modifying existing ones, or even removing some classes from the core. Delta modules can also modify the functional entities of an ABS program, that is, they can add and modify data types and type synonyms, and add functions.

Deltas are applied to the core program by the ABS compiler front end. The choice of which delta modules to apply depends on the selection of a set of features, that is, a particular product of the SPL. The role of the ABS compiler front end is to translate textual ABS models into an internal representation and check the models for syntax and semantic errors. The role of the compiler back end is to generate code for the models targeting some suitable execution or simulation environment.

12.1 Syntax

Figure 12.1 specifies the ABS syntax related to delta modeling. The *DeltaDecl* clause specifies the syntax of delta modules, consisting of an unique identifier, a module access directive, a list of parameters and a sequence of module modifiers. The *module access* directive gives the delta access to the namespace of a particular module. In other words it specifies the ABS module to which the modifications specified by the delta apply by default. A delta can still apply changes to several modules by fully qualifying the *TypeName* of module modifiers.

The *ModuleModifier* clause describes the syntax of modifications at the level of modules. Such a modification can, for example, add a class or interface declaration, modify an existing class or interface, remove a class or interface, and also add functions, data types and type synonyms. Class modifications include the ability to change the interface of a class by adding or removing items from the class's list of implemented interfaces. The *InterfaceModifiers* clause describes how to modify existing interface declarations, either by adding new or removing existing method

```
        DeltaDecl ::= delta TypeId [DeltaParams] ; [ModuleAccess] ModuleModifier*
   ModuleModifier ::= adds ClassDecl
                    | removes class TypeName ;
                    | modifies class TypeName
                      [adds TypeId (, TypeId)*] [removes TypeId (, TypeId)*]
                      { Modifier* }
                    | adds InterfaceDecl
                    | removes interface TypeName ;
                    | modifies interface TypeName { InterfaceModifier* }
                    | adds FunctionDecl
                    | adds DataTypeDecl
                    | modifies DataTypeDecl
                    | adds TypeSynDecl
                    | modifies TypeSynDecl
                    | adds Import
                    | adds Export

InterfaceModifier ::= adds MethSig ;
                    | removes MethSig ;

         Modifier ::= adds FieldDecl
                    | removes FieldDecl
                    | adds MethDecl
                    | removes MethSig
                    | modifies MethDecl

      DeltaParams ::= (DeltaParam (, DeltaParams)* )
       DeltaParam ::= Identifier HasCondition*
                    | Type Identifier

     ModuleAccess ::= uses TypeId ;

     HasCondition ::= hasField FieldDecl
                    | hasMethod MethSig
                    | hasInterface TypeId
```

Figure 12.1: ABS Grammar: Delta Modules.

signatures.

The *Modifier* clause specifies the modifications that can occur within a class or interface body. These include adding and removing fields and methods, and modifying methods, which amounts to replacing a method implementation with a new one, while enabling the original method to be called using the `original` keyword. The aim of **`original`** is to enable the method being replaced to be called from the delta module that replaces it. This is implemented by renaming the original method, and replacing the call to **`original`** with a call to the renamed method.

A method can be replaced multiple times by applying a succession of deltas. To call a specific version of such a method, a *targeted* **`original`** call can be used. The target specifies hereby the delta or the core program.

12.2 Object-oriented modifiers

To modify an object-oriented ABS program, delta modules support adding new classes and removing existing ones. Existing classes can be also modified by adding new methods and also removing or modifying existing methods. Deltas can also add new interface declarations, remove existing interface declarations, and modify interface declarations by adding or removing operations. Furthermore, deltas can change the interface of a class by adding or removing interfaces from the class's list of implemented interfaces. Lastly, delta modules can introduce new fields to classes and remove existing fields.

12.2.1 Interfaces

Deltas can introduce new interface declarations and remove or modify existing interface declarations. The syntax is illustrated by the following examples.

```
delta D1;
adds interface MyModule.I { Unit foo(); }

delta D2;
uses MyModule;
removes interface I;

delta D3;
uses MyModule;
modifies interface I { removes Unit foo(); adds Unit bar(); }
```

12.2.2 Classes

Deltas can introduce new classes and remove existing classes. The syntax is illustrated by the following examples.

```
delta D1;
adds class MyModule.DataBase(Map<Filename,File> db) implements DB {...}

delta D2;
uses MyModule;
```

```
removes class Node();
```

The first delta D1 above declares a new class DataBase inside the module MyModule. Delta D2 removes the class Node from the same module. Specifying to which module such code modifications apply can be done in two ways. First, as exemplified by delta D1, the class name can be qualified with a module name. An alternative way is to include a **uses** <Module Name> clause at the beginning of the delta module, which *opens* a module so that names don't need to be qualified. When a delta specifies modifications to a single module, this method is more concise. When a delta specifies modifications across multiple modules, it is more convenient to qualify each class modifier with a module name. Using both methods together is also possible, in which case unqualified class names will refer to classes defined inside the *used* module.

Deltas can also *modify* existing classes by adding new methods and removing or modifying methods; by adding or removing fields; and by manipulating the list of interfaces that the class implements. These operations are illustrated in the following sections.

12.2.3 Methods

Methods can be added, removed or modified from within deltas. The following example shows a delta module designed to modify the behaviour of the class Greeter by modifying its sayHello method. The class is assumed to have been declared in the core program inside the Hello module.

```
delta Nl;
uses Hello;
modifies class Greeter {
    modifies String sayHello() {
        return "Hallo wereld";
    }
}
```

The above Nl delta module applies its changes to the core ABS module Hello, as specified by the **uses** clause. It provides a new implementation for the method sayHello() in class Greeter by declaring a so-called method *modifier*. The method modifier is introduced by the **modifies** keyword and followed by the method signature and a block of code providing the method's new implementation.

Adding entirely new methods is also supported using the **adds** keyword followed by the method signature and its implementation. Similarly, it is possible to remove methods from classes using **removes** followed by the method signature, as shown in the following.

```
delta D;
modifies class M.Foo {
    adds Int bar() { return 17; }
    removes Unit moo();
}
```

Calling original

Calling **original** from within a method modifier body makes it possible to access the method's previous behaviour, that is, the behaviour implemented in the previously applied delta or in the

core. This is similar to calling `super` to access the superclass behaviour of a method in a language with class inheritance such as Java. An `original` call has to supply a list of arguments that conforms with the original method's list of formal parameters.

Targeted `original` calls

Original calls can be targeted towards a given delta by prefixing the call with the name of the delta, or towards the core ABS code by using the keyword `core`:

```
core.original(params);
Delta.original(params);
```

Regular (untargeted) original calls invoke the method behaviour defined by the previously applied delta. For example, if a method `m` is defined in the core, and then a set of deltas `D1..D3`, which each modify `m`, are applied in sequence, then calling `original` from within `m`'s modifier in `D3` will run the version of `m` defined in `D2`. With a targeted call, one can access any version of `m`, that is, the versions defined in `D2`, `D1` and in the core.

This allows a tighter control of which code is actually executed when calling `original`. As the order of delta application is often not uniquely defined, it is not always determinable which behaviour will be invoked upon calling `original`. With a targeted original call, the user can specify exactly which code to execute and even invoke multiple versions of a method. This, of course, implies that the target delta has been applied already; otherwise the compiler will indicate an error.

```
module M;
class C {
    String m(String s) { return(s) };
}
delta D1;
modifies M.C {
    modifies String m(String s) { return prefix + original(s); }
}
delta D2;
modifies M.C {
    modifies String m(String s) { return original(s) + suffix; }
}
delta Resolve;
modifies M.C {
    modifies String m(String s) { return prefix + core.original(s) + suffix; }
}
```

Consider the above example. `D1` and `D2` both modify method `m` in different, non-compatible ways. We say that these two deltas are in conflict. Assume that `D1` and `D2` can be applied in any order, and that delta `Resolve` has to be applied after `D1` and `D2`. By calling `original` from within `Resolve`, we cannot be sure which version of `m` will actually be invoked: this depends on whether `D1` or `D2` has been applied last. By targeting the original call towards a specific delta, we can control the behaviour precisely, and resolve the conflict in a meaningful way.

Targeted original calls were required for the implementation of the delta modelling workflow (DMW) [6, 5], which is described in more detail in Deliverable 5.3 [3]. The DMW describes a process of applying delta modelling to obtain a model of a software product line that is globally unambiguous and complete. A focus of DMW is the systematic reconciliation of conflicting feature functionality.

12.2.4 Class interfaces

A delta module can change the list of interfaces that a class implements. Adding or dropping interfaces from that list is achieved using the familiar **removes** and **adds** keywords.

The following example shows a core ABS program defining a `Logger` class that implements the `Output` interface. It further declares a delta module that modifies the `Logger` class such that it implements a different interface. This new `IO` interface is introduced in the same delta.

```
module M;
interface Input { String read(); }
interface Output { Unit write(String s); }
class Logger implements Output {
    Unit write(String s) {...}
}

delta IO;
adds interface IO extends Input, Output {}
modifies class Logger adds IO removes Output {
    adds String read() {...}
}
```

12.2.5 Fields

In addition to modifying object behaviour, ABS allows adding or removing fields. New fields are introduced by the **adds** keyword followed by the field's type, name, and an optional value assignment. Similarly, fields can be removed using the **removes** keyword. The following example demonstrates this.

```
delta D;
modifies class M.Foo {
    adds List<Item> items;
    adds Int itemsCount = items.size();
    removes String name;
}
```

12.3 Functional modifiers

Functional program elements can also be modified from within deltas. ABS supports the addition of functions, data types and type synonyms, and the modification of data types and type synonyms.

Qualifying functional elements with the module name is currently unsupported, therefore when adding functional elements, a **uses** clause has to be specified.

12.3.1 Functions

Example of adding a function.

```
delta MyDelta;
uses MyModule;
adds def Int min(Int a, Int b) = case a < b { True => a; False => b; };
```

12.3.2 Data types

Example of adding a data type.

```
adds data Schedule = Schedule(
    String schedname,
    List<Item> items,
    Int sched,
    Deadline dline) | NoSchedule;
```

Example of modifying a data type.

```
modifies data Schedule = Schedule(
    String schedname,
    List<Item> items,
    Int sched,
    Deadline dline) | NoSchedule(String reason);
```

When modifying a datatype, the given constructors supersede the previous list of constructors.

12.3.3 Type synonyms

Example of adding a type synonym.

```
adds type ClientId = Int;
```

Example of modifying a type synonym.

```
modifies type ClientId = String;
```

12.4 Module modifiers

Deltas support, to some extent, modifications to ABS's module system.

12.4.1 Imports and Exports

The addition of (qualified and unqualified) **imports** and **exports** to modules (cf. Chapter 10) is supported, as shown in the following examples.

```
delta D1;
uses Drinks;
adds export Drink, Milk;

delta D2;
uses Bar;
adds export *;
adds import Drinks.Milk;

delta D3;
uses MyModule;
adds import * from Bar;
```

The `adds import` and `adds export` directives apply to the module defined by the `uses` statement.

12.5 Unsupported modifications

While delta modelling supports a broad range of ways to modify an ABS model, not all ABS program entities are modifiable. These unsupported modifications are listed here for completeness. While these modifications could be easily specified and implemented, we opted not to overload the language with features that have not been regarded as necessary in practice.

Class parameters and init block Deltas currently do not support the modification of class parameter lists or class init blocks.

Functional program elements Deltas currently only support adding functions, and adding and modifying data types and type synonyms. Removal is not supported.

Modules Deltas currently do not support adding new modules or removing modules.

Imports and Exports While deltas do support the addition of `import` and `export` statements to modules, they do not support their modification or removal.

Main block Deltas currently do not support the modification of the program's main block.

Chapter 13

Software Product Line Configuration and Product Generation

13.1 Software Product Line Configuration

The ABS configuration language links feature models, which describe the structure of a SPL (Chapter 11), to delta modules (Chapter 12), which implement behaviour. This link is illustrated in Fig. 13.1. The configuration defines, for each selection of features satisfied by the product selection, which delta modules should be applied to the core. Furthermore, it guides the code generation by ordering the application of the delta modules.

Features and delta modules are associated through *application conditions*, which are logical expressions over the set of features and attributes in a feature model. The collection of applicable delta modules is given by the application conditions that are true for a particular feature and attribute selection. By not associating the delta modules directly with features, a degree of flexibility is obtained.

13.1.1 Syntax

The SPL *Configuration* clause specifies the name of the product line, the set of *Features* it provides, and the set of delta modules used to implement those features. The feature names are included so that certain simple self-consistency checks can be performed. The *DeltaClause* is used to specify each delta module, linking it to the feature model.

Syntax:

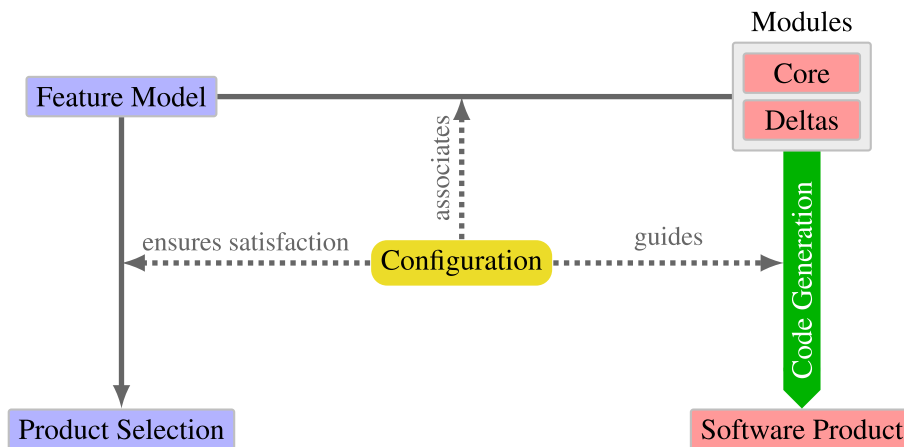

Figure 13.1: Variability modelling framework of ABS

Configuration	::=	`productline` *TypeId* ; *Features* ; *DeltaClauses*
Features	::=	`features` FID (, FID)*
DeltaClauses	::=	*DeltaClause**
DeltaClause	::=	`delta` *DeltaSpec* [*AfterCondition*] [*ApplicationCondition*] ;
DeltaSpec	::=	*TypeName* [(*DeltaParams*)]
DeltaParams	::=	*DeltaParam* (, *DeltaParam*)*
DeltaParam	::=	FID | FID.AID
AfterClause	::=	`after` *TypeName* (, *TypeName*)*
WhenClause	::=	`when` *AppCond*
AppCond	::=	*AppCond* && *AppCond*
	|	*AppCond* || *AppCond*
	|	~ *AppCond*
	|	(*AppCond*)
	|	FID

Each delta clause has a *DeltaSpec*, specifying the name of a delta module name and, optionally, a list of parameters; an *AfterClause*, specifying the delta modules that the current delta must be applied after; and an application condition *AppCond*, specifying an arbitrary predicate over the feature and attribute names in the feature model that describes when the given delta module is applied.

13.1.2 Delta parameters

A delta clause takes an optional list of *DeltaParams*, which are either features or fully qualified feature attributes. When a particular product of the product line is selected (as described in Section 11.2), the value `true` is assigned to each of its features and a value is assigned to each of its features' attributes. These values can be used inside the delta module if the corresponding delta clause specifies the feature or feature attribute as one of its parameters.

13.1.3 Application Conditions

The configuration is used to specify conditions under which a delta module should be applied to the core. Such application conditions are propositional formulas over the set of features and attributes defined by the feature model. They can be in terms of the presence and absence of features and feature combinations, as well as attributes of features and integer and boolean constants. When the propositional formula holds for a given product, the delta module is applicable. Application conditions are introduced by the `when` keyword, as shown in the examples below.

Example:

```
delta D when A or B and ~C;
delta LargeCache when Mem.size > 1024;
```

13.1.4 Application Order of Delta Modules

For each delta module, the configuration establishes a partial ordering relation with respect to other deltas by using the `after` keyword. The partial order states which delta modules, when applicable, should be applied before the given delta module.

13.1.5 Configuration Example

A configuration specifies the name of the product line, the set of features it provides, and the set of delta modules used to implement those features.

Example:

```
productline MultiLingualHelloWorld;
    features English, German, Dutch, Repeat;
    delta De when German;
    delta Nl when Dutch;
    delta Fr when French;
    delta Rpt(Repeat.times) after De, Nl, Fr when Repeat;
```

The example above first names the set of features from the feature model (cf. Section 11.1.2) used to configure this product line. The `delta` clauses link each delta module to the feature model through an application condition (`when` clause); in this case, a delta module is applied simply when the specified feature is selected (e.g. "`De when German`"). There is no delta module corresponding to the feature `English`, as the core module provides support for the English language by default. In addition, `Rpt` has to be applied `after` `De`, `Nl` and `Fr`. The `Rpt` delta has a parameter `Repeat.times`, the `times` attribute of the feature `Repeat`; its value (defined by product selection, see Section 11.2) is propagated to the `Rpt` delta as described in Section 13.1.2.

13.2 Product Generation

The process of generating a software product from a given software product line relies on the full ABS specification of an SPL, that is, a feature model with a set of product declarations (Chapter 11), a set of delta modules (Chapter 12), and a configuration that connects the two (Section 13.1 above). To generate a particular product, this has to be selected either using the Eclipse IDE (cf. Figure 13.2), or given as an argument when invoking the compiler on the command line. For example:

```
generateJava -product=P2 Hello.abs
```

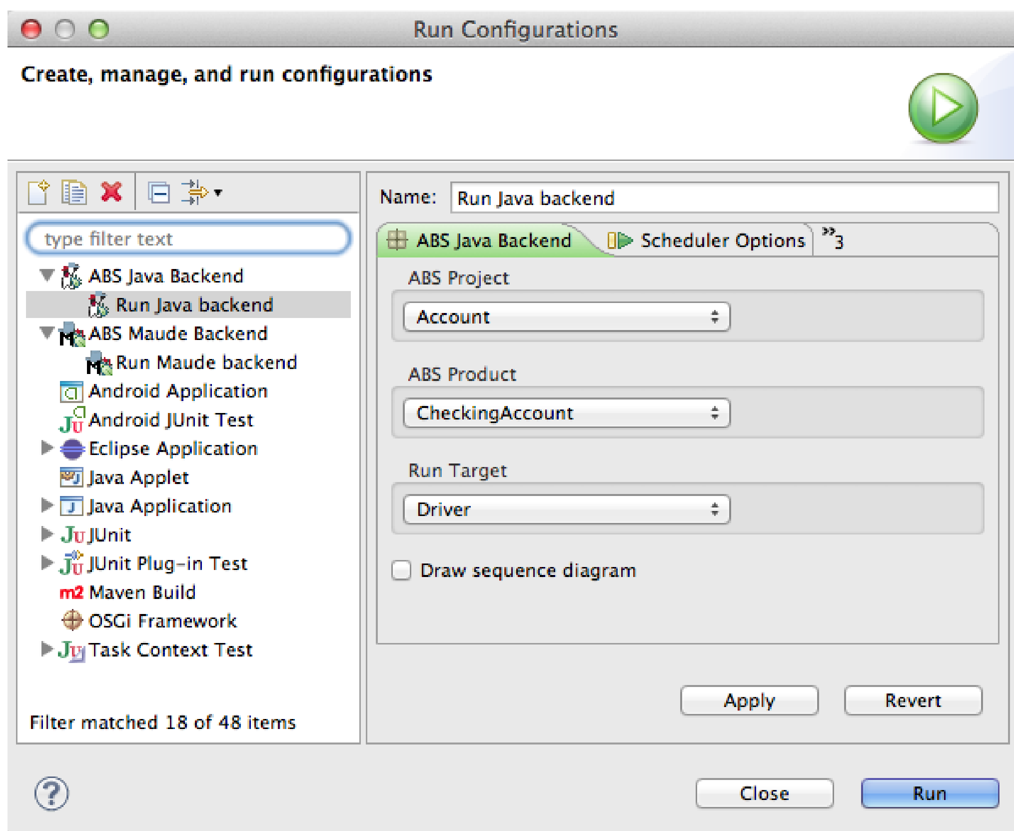

Figure 13.2: SPL product generation using the Eclipse IDE

13.2.1 Example

For example, selecting the product `P2` of the "Hello world" product line triggers the compiler to execute through the following steps.

1. Check if the product is valid with respect to the feature model. In this case the product `P2` is valid.

2. Find all applicable delta modules. In this case the deltas modules with an application condition that holds are `Nl` and `Rpt(3)`.

3. Linearise the application order of delta modules. The restriction here is over the delta module `Rpt`, which has to follow any of the language deltas.

4. Apply the deltas sequentially. First, the `Nl` delta is applied to the Core ABS program. Then the `Rpt(3)` delta is applied to the result of the previous application.

The result is a core ABS program.

Chapter 14

Annotations

ABS supports *Annotations* to enrich an ABS model with additional information, for example, to realize pluggable type systems. Annotations can appear before any declaration and type usage in ABS programs (which is not given in the grammar definitions, to improve readability).

Syntax:

Annotation ::= `[` [*TypeName* `:`] *PureExp* `]`

Example:

```
[LocationType:Near] Peer p;
[Far] Network n;
List<[Near] Peer> peers = Nil;
```

14.1 Type Annotations

ABS has a predefined meta-annotation `TypeAnnotation` to declare annotations to be *Type Annotations*. Data types that are annotated with that annotation are specially treated by the ABS compiler to support an easier implementation of pluggable type systems.

Example:

```
[TypeAnnotation]
data LocationType = Far | Near | Somewhere | Infer;
```

Chapter 15

Models

A *model* in ABS represents a type-closed set of *modules*. A module defines a set of declarations and an optional *main block*. Modules reside in *compilation units*, which are typically represented by files ending with `.abs`. A model is thus set of compilation units.

Syntax:

Model	::=	*CompilationUnit**
CompilationUnit	::=	*ModuleDecl**

Bibliography

[1] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated Analysis of Feature Models: Challenges Ahead. *Commun. ACM*, 49(12), 2006.

[2] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In *Formal Methods for Components and Objects*, volume 6957 of *LNCS*. Springer-Verlag, 2011.

[3] Evaluation of Modeling, March 2012. Deliverable 5.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[4] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[5] M. Helvensteijn, R. Muschevici, and P.Y.H. Wong. Delta Modeling in Practice, a Fredhopper Case Study. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.

[6] Michiel Helvensteijn. Delta Modeling Workflow. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.

[7] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[8] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University Software Engineering Institute, 1990.

[9] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[10] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[11] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[12] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, September 2010.

Appendix A

ABS Standard Library

```
module ABS.StdLib;
export *;

data Unit = Unit;               // builtin
data String;                    // builtin
data Int;                       // builtin
data Bool = True | False;       // builtin
data Fut<A>;                    // builtin

def Bool and(Bool a, Bool b) = a && b;
def Bool not(Bool a) = ~a;

def Int max(Int a, Int b) =
    case a > b { True => a; False => b; };

def Int abs(Int x) =
    case x > 0 { True => x; False => -x; };

data Maybe<A> = Nothing | Just(A);

def A fromJust<A>(Maybe<A> a) = case a { Just(j) => j; };
def Bool isJust<A>(Maybe<A> a) =
    case a { Just(j) => True; Nothing => False; };

data Either<A, B> = Left(A) | Right(B);

def A left<A,B>(Either<A, B> val) =
    case val { Left(x) => x; };

def B right<A,B>(Either<A, B> val) =
    case val { Right(x) => x; };

def Bool isLeft<A,B>(Either<A, B> val) =
```

```
    case val { Left(x) => True; _ => False; };

def Bool isRight<A,B>(Either<A, B> val) = ~isLeft(val);


data Pair<A, B> = Pair(A, B);

def A fst<A, B>(Pair<A, B> p) = case p { Pair(s, f) => s; };
def B snd<A, B>(Pair<A, B> p) = case p { Pair(s, f) => f; };


data Triple<A, B, C> = Triple(A, B, C);

def A fstT<A, B, C>(Triple<A, B, C> p) =
    case p { Triple(s, f, g) => s; };

def B sndT<A, B, C>(Triple<A, B, C> p) =
    case p { Triple(s, f, g) => f; };

def C trd<A, B, C>(Triple<A, B, C> p) =
    case p { Triple(s, f, g) => g; };

// Sets
data Set<A> = EmptySet | Insert(A, Set<A>);

// set constructor helper
def Set<A> set<A>(List<A> l) =
    case l {
       Nil => EmptySet;
       Cons(x,xs) => Insert(x,set(xs));
    };

/**
 * Returns True if set 'ss' contains element 'e', False otherwise.
 */
def Bool contains<A>(Set<A> ss, A e) =
  case ss {
    EmptySet => False ;
    Insert(e, _) => True;
    Insert(_, xs) => contains(xs, e);
  };

/**
 * Returns True if set 'xs' is empty, False  otherwise.
 */
def Bool emptySet<A>(Set<A> xs) = (xs == EmptySet);
```

```
/**
 * Returns the size of set 'xs'.
 */
def Int size<A>(Set<A> xs) =
   case xs {
      EmptySet => 0 ;
      Insert(s, ss) => 1 + size(ss);
   };

def Set<A> union<A>(Set<A> set1, Set<A> set2) =
   case set1 {
      EmptySet => set2;
      Insert(a, s) => union(s,insertElement(set2,a));
   };

/**
 * Returns a set with all elements of set 'xs' plus element 'e'.
 * Returns 'xs' if 'xs' already containts 'e'.
 */
def Set<A> insertElement<A>(Set<A> xs, A e) =
  case contains(xs, e) {
    True => xs;
    False => Insert(e, xs);
  };

/**
 * Returns a set with all elements of set 'xs' except element 'e'.
 */
def Set<A> remove<A>(Set<A> xs, A e) =
  case xs {
     EmptySet => EmptySet ;
     Insert(e, ss) => ss;
     Insert(s, ss) => Insert(s,remove(ss,e));
  };

// checks whether the input set has more elements to be iterated.
def Bool hasNext<A>(Set<A> s) = ~ emptySet(s);

// Partial function to iterate over a set.
def Pair<Set<A>,A> next<A>(Set<A> s) =
   case s {
      Insert(e, set2) => Pair(set2,e);
   };

// Lists
```

```
data List<A> = Nil | Cons(A, List<A>);

def List<A> list<A>(List<A> l) = l; // list constructor helper

/**
 * Returns the length of list 'list'.
 */
def Int length<A>(List<A> list) =
   case list {
      Nil => 0 ;
      Cons(p, l) => 1 + length(l) ;
   };

/**
 * Returns True if list 'list' is empty, False otherwise.
 */
def Bool isEmpty<A>(List<A> list) = list == Nil;

/**
 * Returns the first element of list 'list'.
 */
def A head<A>(List<A> list) =
   case list { Cons(p,l) => p ; };

/**
 * Returns a (possibly empty) list containing all elements of 'list'
 * except the first one.
 */
def List<A> tail<A>(List<A> list) =
   case list { Cons(p,l) => l ; };

/**
 * Returns element 'n' of list 'list'.
 */
def A nth<A>(List<A> list, Int n) =
  case n {
    0 => head(list) ;
    _ => nth(tail(list), n-1);
  };

/**
 * Returns a list where all occurrences of a have been removed
 */
def List<A> without<A>(List<A> list, A a) =
  case list {
     Nil => Nil;
```

```
     Cons(a, tail) => without(tail,a);
     Cons(x, tail) => Cons(x, without(tail,a));
  };

/**
 * Returns a list containing all elements of list 'list1'
 * followed by all elements of list 'list2'.
 */
def List<A> concatenate<A>(List<A> list1, List<A> list2) =
  case list1 {
    Nil => list2 ;
    Cons(head, tail) =>  Cons(head, concatenate(tail, list2));
  };

/**
 * Returns a list containing all elements of list 'list' followed by 'p'.
 */
def List<A> appendright<A>(List<A> list, A p) =
    concatenate(list, Cons(p, Nil));

/**
 * Returns a list containing all elements of 'list' in reverse order.
 */
def List<A> reverse<A>(List<A> list) =
  case list {
     Cons(hd, tl) => appendright(reverse(tl), hd);
     Nil => Nil;
  };

/**
 * Returns a list of length 'n' containing 'p' n times.
 */
def List<A> copy<A>(A p, Int n) =
   case n { 0 => Nil; m => Cons(p,copy(p,m-1)); };




// Maps
data Map<A, B> = EmptyMap | InsertAssoc(Pair<A, B>, Map<A, B>);
 // map constructor helper (does not preserve injectivity)
def Map<A, B> map<A, B>(List<Pair<A, B>> l) =
  case l {
     Nil => EmptyMap;
     Cons(hd, tl) => InsertAssoc(hd, map(tl));
  };
```

```
def Map<A, B> removeKey<A, B>(Map<A, B> map, A key) = // remove from the map
  case map {
    InsertAssoc(Pair(key, _), map) => map;
    InsertAssoc(pair, tail) => InsertAssoc(pair, removeKey(tail, key));
  };


def List<B> values<A, B>(Map<A, B> map) =
  case map {
    EmptyMap => Nil ;
    InsertAssoc(Pair(_, elem), tail) => Cons(elem, values(tail)) ;
  };

/**
 * Returns a set containing all keys of map 'map'.
 */
def Set<A> keys<A, B>(Map<A, B> map) =
  case map {
    EmptyMap => EmptySet ;
    InsertAssoc(Pair(a, _), tail) => Insert(a, keys(tail));
  };

/**
 * Returns the value associated with key 'k' in map 'ms'.
 */
def B lookup<A, B>(Map<A, B> ms, A k) = // retrieve from the map
  case ms {
     InsertAssoc(Pair(k, y), _) => y;
     InsertAssoc(_, tm) => lookup(tm, k);
  };

/**
 * Returns the value associated with key 'k' in map 'ms', or the value 'd'
 * if 'k' has no entry in 'ms'.
 */
def B lookupDefault<A, B>(Map<A, B> ms, A k, B d) = // retrieve from the map
  case ms {
     InsertAssoc(Pair(k, y), _) => y;
     InsertAssoc(_, tm) => lookupDefault(tm, k, d);
     EmptyMap => d;
  };

/**
 * Returns a map with all entries of 'map' plus an entry 'p',
```

```
 * which might override but not remove another entry with the same key.
 */
def Map<A, B> insert<A, B>(Map<A, B> map, Pair<A, B> p) = InsertAssoc(p, map);

/**
 * Returns a map with all entries of 'ms' plus an entry mapping 'k' to 'v',
 * minus the first entry already mapping 'k' to a value.
 */
def Map<A, B> put<A, B>(Map<A, B> ms, A k, B v) =
  case ms {
    EmptyMap => InsertAssoc(Pair(k, v),EmptyMap);
    InsertAssoc(Pair(k, _), ts) => InsertAssoc(Pair(k, v), ts);
    InsertAssoc(p, ts) => InsertAssoc(p, put(ts, k, v));
  };

/**
 * Returns a string with the base-10 textual representation of 'n'.
 */
def String intToString(Int n) =
  case n < 0 {
    True => "-" + intToStringPos(-n);
    False => intToStringPos(n);
  };

def String intToStringPos(Int n) =
  let (Int div) = (n / 10) in
  let (Int res) = (n % 10) in
  case n {
    0 => "0"; 1 => "1"; 2 => "2"; 3 => "3"; 4 => "4";
    5 => "5"; 6 => "6"; 7 => "7"; 8 => "8"; 9 => "9";
    _ => intToStringPos(div) + intToStringPos(res);
  };

/**
 * Returns a substring of string str of the given length starting from start (inclusive)
 * Where the first character has index 0
 *
 * Example:
 *    substr("abcde",1,3) => "bcd"
 *
 */
def String substr(String str, Int start, Int length) = builtin;

/**
 * Returns the length of the given string
 */
```

```
def Int strlen(String str) = builtin;

// A Time datatype.
data Time = Time(Int);
def Int currentms() = builtin;
def Time now() = Time(currentms());
def Int timeval(Time t) = case t { Time(v) => v; };
// use this like so:
//   Time t = now(); await timeDifference(now(), t) > 5;
def Int timeDifference(Time t1, Time t2) =
  abs(timeval(t2) - timeval(t1));

/**
 * Annotation data type to define the type of annotations
 * currently only TypeAnnotation exists
 */
data Annotation = TypeAnnotation;

[TypeAnnotation]
data LocationType = Far | Near | Somewhere | Infer;

/**
 * Can be used to annotated classes and to ensure that
 * classes are always instantiated in the right way.
 * I.e. classes annotated with [COG] must be created by using
 * new cog, class annotated with [Plain] must be created by using
 * just new, without cog.
 */
data ClassKindAnnotation = COG | Plain;

/**
 * Declare local variables to be final
 */
data FinalAnnotation = Final;

/**
 * Declare methods to be atomic, i.e., such methods must not
 * contain scheduling code and also no .get
 */
data AtomicityAnnotation = Atomic;
```