ACSL: ANSI C Specification Language

Preliminary design (version 1.2, February 13, 2008)

Patrick Baudin[1], Jean-Chirstophe Filliâtre[4,3], Claude Marché[3,4],
Benjamin Monate[1], Yannick Moy[2,4,3], Virgile Prevosto[1]

[1] CEA LIST, Software Reliability Laboratory, Saclay, F-91191
[2] France Télécom, Lannion, F-22307
[3] INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
[4] LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

This work has been supported by the 'CAT' ANR project (ANR-05-RNTL-0030x) and by the ANR CIFRE contract 2005/973.

Contents

Foreword

This is a preliminary design of the ACSL language, a deliverable of the task 7.2 of the ANR RNTL project CAT (http://www.rntl.org/projet/resume2005/cat.htm). In this project, a reference implementation of ACSL is provided: the Frama-C platform (http://www.frama-c.cea.fr).

This is the version 1.2 of ACSL design. Several features may still evolve in the future. In particular, some features in this document are considered *experimental*, meaning that their syntax and semantics is not already fixed. These features are marked with EXPERIMENTAL. They must also be considered as advanced features, which are not supposed to be useful for a basic use of that specification language.

Chapter 1

Introduction

This document is a reference manual for ACSL, an acronym for "ANSI C Specification Language". This is a Behavioral Interface Specification Language (aka BISL) implemented in the FRAMA-C framework. As suggested by its name, it aims at specifying behavioral properties of C source code. The main inspiration for this language comes from the specification language of the CADUCEUS tool [7, 8] for deductive verification of behavioral properties of C programs. It is itself inspired from the *Java Modeling Language* (JML [16]) which aims at similar goals for Java source code: indeed it aims both at *runtime assertion checking* and *static verification* using the ESC/JAVA2 tool [12], where we aim at *static verification* and *deductive verification* (see Appendix A.2 for a detailed comparison between ACSL and JML).

Going back further in history, JML design was guided by the general *design-by-contract* principle proposed by Bertrand Meyer, who took his own inspiration from the concepts of preconditions and postconditions on a routine, going back at least to Dijkstra, Floyd and Hoare in the late 60's and early 70's, and originally implemented in the EIFFEL language.

In this document, we assume that the reader has a good knowledge of the ANSI C programming language [11, 10].

1.1 Organization of this document

In this preliminary chapter we introduce some definitions and vocabulary, and discuss generalities about this specification language. Chapter 2 presents the specification language itself. Chapter 3 presents additional informations about *libraries* of specifications. Finally, chapter A provides a few additional information. A detailed table of contents is given on page 2. A glossary is given in Appendix A.1.

1.2 Generalities about Annotations

In this document, we consider that specifications are given as annotations in comments written directly in C source files, so that source files remain compilable. Those comments must start by `/*@` or `//@` and end as usual in C.

For some particular context, it might not be possible to modify the source code. It is strongly recommended that a tool which implements ACSL specifications provides a technical means to store annotations in separate files. This is not the purpose of this document to describe such technical means. Nevertheless, some of the specifications, namely those at a global level, can be given in separate files: logical specifications can be imported (see Section 2.6.6) and a function contract can be attached to a copy of the function profile (see Section 2.3.5).

1.2.1 Kinds of annotations

- Global annotations:
  - *function contract* : such an annotation is inserted just before the declaration or the definition of a function. See section 2.3.
  - *global invariant* : this is allowed at the level of global declarations. See section 2.10.
  - *type invariant* : this allows both structure or union invariants, and invariants on type names introduced by `typedef`. See section 2.10.
  - *logic specifications* : logic type introduction, introduction or definition of logic functions or predicates, axioms. Such an annotation is placed at the level of global declarations.
- Statement annotations:
  - `assert` *clause* : these are allowed everywhere a C label is allowed, or just before a block closing brace.
  - *loop annotation* (invariant, variant, assign clauses): is allowed immediately before a loop statement: `for` , `while` , `do` ... `while` . See Section 2.4.2.
  - *statement contract* : very similar to a function contract, and placed before a statement or a block. Semantical condition must be checked (normal termination only, no goto going inside, no goto going outside). See Section 2.4.4.
  - *ghost code* : regular C code, only visible from the specification, that is only allowed to modify ghost variables. See section 2.11. This includes ghost braces for enclosing blocks.

1.2.2 Parsing annotations in practice

In JML, parsing is done by simply ignoring `//@`, `/*@` and `*/` at the lexical analysis level. This technique could modify the semantics of the code, for example:

```
return x /*@ +1 */ ;
```

In our language, this is forbidden. Technically, the current implementation of Frama-C isolates the comments in a first step of syntax analysis, and then parses a second time. Nevertheless, the grammar and the corresponding parser must be carefully designed to avoid interaction of annotations with the code. For example, in code such as

```
if (c) //@ assert P;
   c=1;
```

the statement `c=1` must be understood as the branch of the `if`. This is ensured by the grammar below, saying that `assert` annotations are not statement themselves, but attached to the statement that follows, like C labels.

1.2.3 About preprocessing

This document considers C source *after* preprocessing. Tools must decide what to do for preprocessing phase: what to do with annotations, should macro substitution be performed or not, etc.

1.2.4 About keywords

Additional keywords of the specification language start with a backslash, if they are used in position of a term or a predicate (which are defined in the following). Otherwise they do not start with a backslash (like `ensures`) and they remain valid identifiers.

1.3 Notations for grammars

In this document, grammar rules are given in BNF form. In grammar rules, we use extra notations e^* to denote repetition of zero, one or more occurrences of e, e^+ for repetition of one or more occurrences of e, and $e^?$ for zero or one occurrence of e.

Acknowledgements

We gratefully thank all people who contributed to this document: Sylvie Boldo, Jean-Louis Colaço, Pierre Crégut, Pascal Cuoq, David Delmas, Stéphane Duprat, Arnaud Gotlieb, Thierry Hubert, Dillon Pariente, Pierre Rousseau, Julien Signoles, Jean Souyris.

Chapter 2

Specification language

2.1 Lexical rules

Lexical structure mostly follows that of ANSI C. A few differences must be noted:

- the at sign (`@`) is a blank character, thus equivalent to a space character.
- the identifiers may start with the backslash character (`\`).
- Some UTF8 characters may be used in place of some constructs, as shown in the following table

`>=`	$\geq$	0x2265
`<=`	$\leq$	0x2264
`>`	$>$	0x003E
`<`	$<$	0x003C
`!=`	$\neq$	0x2262
`==`	$\equiv$	0x2261
`==>`	$\Longrightarrow$	0x21D2
`<==>`	$\Longleftrightarrow$	0x21D4
`&&`	$\wedge$	0x2227
`||`	$\vee$	0x2228
`^^`	$\oplus$	0x22BB
`!`	$\neg$	0x00AC
`\forall`	$\forall$	0x2200
`\exists`	$\exists$	0x2203

In this document, we use UTF8 characters in the examples, to improve readability.

2.2 Logic expressions

This first section presents the language of expressions one can use in annotations. These are called *logic expressions* in the following. They correspond to pure C expressions, with additional constructs that we will introduce progressively.

Figure 2.1 presents the grammar for the basic constructs of logic expressions. In that grammar, we distinguish between *predicates* and *terms*, following the usual distinction between propositions and terms in classical first-order logic. The grammar for binders and type expressions is given separately in Figure 2.2.

With respect to C pure expressions, the additional constructs are as follows:

bin-op	::=	`+` | `-` | `*` | `/` | `%`	
	|	`==` | `!=` | `<=` | `>=` | `>` | `<`	
	|	`&&` | `||` |	boolean operations
	|	`&` | `|` | `-->` | `<-->` | `^`	bitwise operations
unary-op	::=	`+` | `-`	unary plus and minus
	|	`!`	boolean negation
	|	`~`	bitwise complementation
	|	`*`	pointer dereferencing
	|	`&`	address operator
term	::=	`\true` | `\false`	
	|	*integer*	integer constants
	|	*real*	real constants
	|	*id*	variables
	|	*unary-op term*	
	|	*term bin-op term*	
	|	*term* `[` *term* `]`	array access
	|	*term* `.` *id*	structure field access
	|	*term* `->` *id*	
	|	`(` *type-expr* `)` *term*	cast
	|	*id* `(` *term* (`,` *term*)* `)`	function application
	|	`(` *term* `)`	parentheses
	|	*term* `?` *term* `:` *term*	
	|	`\let` *id* `=` *term* `;` *term*	local binding
	|	`sizeof` `(` *term* `)`	
	|	`sizeof` `(` *C-type-expr* `)`	
	|	*id* `:` *term*	syntactic naming
rel-op	::=	`==` | `!=` | `<=` | `>=` | `>` | `<`	
pred	::=	`\true` | `\false`	
	|	*term* (*rel-op term*)$^+$	comparisons, see remark below
	|	*id* `(` *term* (`,` *term*)* `)`	predicate application
	|	`(` *pred* `)`	parentheses
	|	*pred* `&&` *pred*	conjunction
	|	*pred* `||` *pred*	disjunction
	|	*pred* `==>` *pred*	implication
	|	*pred* `<==>` *pred*	equivalence
	|	`!` *pred*	negation
	|	*pred* `^^` *pred*	exclusive or
	|	*term* `?` *pred* `:` *pred*	
	|	*pred* `?` *pred* `:` *pred*	
	|	`\let` *id* `=` *term* `;` *pred*	local binding
	|	`\forall` *binders* `;` *pred*	universal quantification
	|	`\exists` *binders* `;` *pred*	existential quantification
	|	*id* `:` *pred*	syntactic naming

Figure 2.1: Grammar of terms and predicates

binders	::=	*type-expr variable-ident* (`,` *type-expr*$^?$ *variable-ident*)*	
type-expr	::=	*logic-type-expr* | *C-type-expr*	
logic-type-expr	::=	*built-in-logic-type*	
	|	*id*	type identifier
built-in-logic-type	::=	`boolean` | `integer` | `real`	
variable-ident	::=	*id* | $\star$ *variable-ident* | *variable-ident* `[]`	

Figure 2.2: Grammar of binders and type expressions

Additional connectives C operators `&&` (UTF8: $\wedge$), `||` (UTF8: $\vee$) and `!` (UTF8: $\neg$) are used as logical connectives. There are additional connectives `==>` (UTF8: $\Longrightarrow$) for implication, `<==>` (UTF8: $\Longleftrightarrow$) for equivalence and `^^` (UTF8: $\oplus$) for exclusive or. These logical connectives all have a bitwise counterpart, either C ones like `&`, `|`, `~` and `^`, or additional ones like the bitwise implication `-->` and the bitwise equivalence `<-->`.

Quantification Universal quantification is denoted by `\forall` τ $x_1, \ldots, x_n$; e and existential quantification by `\exists` τ $x_1, \ldots, x_n$; e.

Local binding `\let` x = e_1 `;` e_2 introduces the name x for expression e_1 which can be used in expression e_2.

Conditional c `?` e_1 `:` e_2. There is a subtlety here: the condition may be either a boolean term or a predicate. In case of a predicate, the two branches must be also predicates, so that this construct acts as a connective with the following semantics: c`?`e_1`:`e_2 is equivalent to (c`==>`e_1)`&&`(`!`c`==>`e_2).

Syntactic naming id`:`e is a term or a predicate equivalent to e. It is different from local naming with `\let` : the name cannot be reused in other terms or predicates. It is only for readibility purposes.

Logic functions Applications in terms and in propositions are not applications of C functions, but of logic functions or predicates; see Section 2.6 for detail.

Consecutive comparison operators The construct t_1 $relop_1$ t_2 $relop_2$ $t_3 \cdots t_k$ with several consecutive comparison operators is a shortcut for t_1 $relop_1$ t_2 `&&` t_2 $relop_2$ t_3 `&&` $\cdots$. It is required that the $relop_i$ operators must be in the same "direction", *i.e.* they must all belong either to $\{<, <=, ==\}$ or to $\{>, >=, ==\}$. Expressions such as `x < y > z` or `x != y != z` are not allowed.

To enforce the same interpretation as in C expressions, one may need to add extra parentheses: a`==`b`<`c is equivalent to a`==`b`&&`b`<`c, whereas a`==(`b`<`c`)` is equivalent to `\let` x`=`b`<`c`;` a`==`x. This situation raises some issues, see example below.

There is a subtlety with respect to comparison operators: they are predicates when used in predicate position, and boolean functions when used in term position.

Example 2.1 *Let's consider the following example:*

```
int f(int a, int b) { return a < b; }
```

The following post-conditions are wrong:

- *the obvious post-condition* `\result == a < b` *is not the right one because it is actually a shortcut for* `\result == a && a < b`.

class	associativity	operators
selection	left	`[ ··· ] -> .`
unary	right	`! ~ + - * &` (cast) `sizeof`
multiplicative	left	`* / %`
additive	left	`+ -`
shift	left	`<< >>`
comparison	left	`< <= > >=`
comparison	left	`== !=`
bitwise and	left	`&`
bitwise xor	left	`^`
bitwise or	left	`|`
bitwise implies	left	`-->`
bitwise equiv	left	`<-->`
connective and	left	`&&`
connective xor	left	`^^`
connective or	left	`||`
connective implies	right	`==>`
connective equiv	left	`<==>`
ternary connective	right	`··· ? ··· : ···`
binding	left	`\forall \exists \let`
naming	right	`:`

Figure 2.3: Operator precedence

- *adding parentheses results in an ill-typed post-condition* `\result == (a < b)`*, because it tests equality of* `\result` *which has type* `int` *with* `a < b` *which has type* `boolean`*.*

- *similarly,* `\result <==> a < b` *is not well-typed, because it makes an equivalence between an* `int` *and a predicate.*

The following are correct post-conditions:

- `\result != 0 <==> a < b` *is acceptable because it is an equivalence between two predicates.*

- `\result == (integer)(a<b)` *is also acceptable because it compares two integers. The cast towards* `integer` *enforces* `a<b` *to be understood as a boolean term. Notice that a cast towards* `int` *would also be acceptable.*

2.2.1 Operators precedence

The precedence of C operators is conservatively extended with additional operators, as shown Figure 2.3. In this table, operators are sorted from highest to lowest priority. Operators of same priority are presented on the same line.

There is a remaining ambiguity between the connective `..?..:..` and the labelling operator `:`. Consider for instance the expression `x?y:z:t`. The precedence table does not indicate whether this should be understood as `x?(y:z):t` or `x?y:(z:t)`. Such a case must be considered as a syntax error, and should be fixed by explicitly adding parentheses.

2.2.2 Semantics

The semantics of logic expressions in ACSL is based on mathematical first-order logic (http://en.wikipedia.org/wiki/First_order_logic). In particular, it is a 2-valued logic with only total functions. Consequently, expressions are never "undefined". This is an important design choice and the specification writer should be aware of that. (For a discussion about the issues raised by such design choices, in similar specification languages such as JML, see the comprehensive list compiled by Patrice Chalin [1, 2].)

Having only total functions implies than one can write terms such as `1/0`, or `*p` when `p` is null (or more generally when it points to a non-properly allocated memory cell). In particular, the predicates

```
1/0  ==  1/0
 *p  ==  *p
```

are valid, since they are instances of the axiom $\forall x, x = x$ of first-order logic. Of course, there is no way to deduce anything useful about such terms. As usual, it is up to the specification designer to write consistent assertions. For example, when introducing the following lemma (see Section 2.6):

```
/*@ lemma div_mul_identity:
  @   ∀ real x, real y; y ≠ 0.0 ⟹ y*(x/y) ≡ x;
  @*/
```

a premise is added to require `y` to be non zero.

2.2.3 Typing

The language of logic expressions is typed (as in *multi-sorted* first-order logic). Types are either C types or *logic types* defined as follows:

- "mathematical" types: `integer` for unbounded, mathematical integers, `real` for real numbers, `boolean` for booleans (with values written `\true` and `\false`);
- logic types introduced by the specification writer (see Section 2.6).

There are implicit coercions for numeric types:

- C integral types `char`, `short`, `int` and `long`, signed or unsigned, are all subtypes of type `integer`;
- `integer` is itself a subtype of type `real`;
- C types `float` and `double` are subtypes of type `real`.

Notes:

- There is a distinction between booleans and predicates. The expression $x < y$ in term position is a boolean, and the same expression is also allowed in predicate position.
- Unlike in C, there is a distinction between booleans and integers. There is an implicit promotion from integers to booleans, thus one may write `x && y` instead of `x != 0 && y != 0`. If the reverse conversion is needed, an explicit cast is required, *e.g.* `(int)(x>0)+1`, where `\false` becomes 0 and `\true` becomes 1.
- Quantification can be made over any type: logic types and C types. Quantification over pointers must be carefully used, since it depends on the memory state where dereferencing is done (see Section 2.2.4 and Section 2.6.5).

Formal typing rules for terms are given in appendix A.3.

2.2.4 Integer arithmetic and machine integers

The following integer arithmetic operations apply to *mathematical integers*: addition, subtraction, multiplication, unary minus. The value of a C variable of an integral type is promoted to a mathematical integer. As a consequence, there is no such thing as "overflow" in logic expressions.

Division and modulo are also mathematical operations, which coincide with the corresponding C operations on C machine integers, thus following the ANSI C99 conventions. In particular, these are not the usual mathematical Euclidean division and remainder. Generally speaking, division rounds the result towards zero. The results are not specified if divisor is zero; otherwise if q and r are the quotient and the remainder of n divided by d then:

- $|d \times q| \leq |n|$, and $|q|$ is maximal for this property;
- q is zero if $|n| < |d|$;
- q is positive if $|n| \geq |d|$ and n and d have the same sign;
- q is negative if $|n| \geq |d|$ and n and d have the opposite signs;
- $q \times d + r = n$;
- $|r| < |d|$;
- r is zero or has the same sign as d.

Example 2.2 *The following examples illustrate the results of division and modulo depending on the sign of their arguments:*

- `5/3` *is 1 and* `5%3` *is 2;*
- `(-5)/3` *is -1 and* `(-5)%3` *is -2;*
- `5/(-3)` *is -1 and* `5%(-3)` *is 2;*
- `(-5)/(-3)` *is 1 and* `(-5)%(-3)` *is -2.*

Hexadecimal and octal constants

Hexadecimal and octal constants are always non-negative. Suffixes `u` and `l` for C constants are allowed but meaningless.

Casts and overflows

In logic expressions, casting from mathematical integers to an integral C type t (such as `char`, `short`, `int`, etc.) is allowed and is interpreted as follows: the result is the unique value of the corresponding type that is congruent to the mathematical result modulo the cardinal of this type, that is $2^{8\times\texttt{sizeof}(t)}$.

Example 2.3 `(unsigned char)1000` *is* $1000 \bmod 256$ *i.e.* 232.

To express in the logic the value of a C expression, one has to add all the necessary casts. For example, the logic expression denoting the value of the C expression $x*y+z$ is `(int)((int)(x*y)+z)`. Note that there is no implicit cast from integers to C integral types.

Example 2.4 *The declaration*

```
//@ logic int f(int x) = x+1 ;
```

is not allowed because `x+1`*, which is a mathematical integer, must be casted to* `int`*. One should write either*

```
//@ logic integer f(int x) = x+1 ;
```

or

```
//@ logic int f(int x) = (int)(x+1) ;
```

Quantification on C integral types

Quantification over a C integral type corresponds to integer quantification over the corresponding interval.

Example 2.5 *Thus the formula*

```
\forall char c; c <= 1000
```

is equivalent to

```
\forall integer c; -128 <= c <= 127 ==> c <= 1000
```

Size of C integer types

Remark that the size of C types is architecture-dependent. ACSL does not enforce these sizes either, hence the semantics of terms involving such types is also architecture-dependent (but of course in a consistent way as the C code). The sizeof operator may be used in the logic.

Constants giving maximum and minimum values of those types may be given in a library.

Enum types

Enum types are also interpreted as mathematical integers. Casting of an integer to a enum in the logic must give the same result as if it was in the C code.

Bitwise operations

Like arithmetic operations, bitwise operations apply to any mathematical integers: any mathematical integer as a unique infinite 2-complement binary representation with infinitely many 0 (for non-negative numbers) or 1 (for negative numbers) on the left. Then bitwise operations apply to these representation.

Example 2.6

- `7 & 12` = ···00111`&`···001100 = ···00100 = *4*
- `-8 | 5` = ···11000`|`···00101 = ···11101 = *-3*
- `~5` = `~`···00101 = ···111010 = *-6*
- `-5 << 2` = ···11011`<<`2 = ···11101100 = *-20*
- `5 >> 2` = ···00101`>>`2 = ···0001 = *1*
- `-5 >> 2` = ···11011`>>`2 = ···1110 = *-2*

2.2.5 Real numbers and floating point numbers

Floating-point constants and operations are interpreted as mathematical real numbers. A C variable of type float or double is implicitly promoted to a real. Integers are promoted to reals if necessary.

Example 2.7 $2 * 3.5$ *is equal to the real number 7*

Comparisons operators are interpreted as real operators too.

Casts, infinity and NaNs

Casting from an integer type or a float type to a float or a double is as in C: same conversion operations applies.

Casting a real number x to a float (resp. a double) results in a float value y with the nearest-even rounding mode (see IEEE 754 norm), which is the closest to x real number which is representable in the float (resp. double) type. If the source real number is too large, this may also result into one of the special values +infinity and -infinity. This semantics ensures that the float result of a C operation $e_1 ope_2$ on floats, if the default rounding mode is not changed in the program, might be obtained in the logic under the form $(float)(e_1 ope_2)$.

Example 2.8 *We have* `(float)0.1` = 13421773×2^{-27} *which is equal to* 0.10000000149006473916

Rounding of real number towards a float with other rounding modes might be obtained via specific logic functions given in external libraries. As well, classical mathematical operations like exponential, sine, cosine, etc. are supposed to be available in some library (see Section 3.3.1).

```
Il faut pouvoir dire, comme
pour les operations entieres, que le resultat d'une operation C sur
des float est le cast de l'operation logique:

on peut aussi obtenir Nan, + inf, - inf, et -0

mode d'arrondi ?



on veut des fonctions predef :

isNan,

isFinite,

\finite_float: float -> prop
\finite_double: double -> prop

\nearest_even_float: real -> float
\nearest_even_double: real -> double

\to_zero

\round_up_float
```

fun-contract ::= *simple-behavior named-behavior** *decreases-clause*$^?$
simple-behavior ::= (*requires-clause* | *assigns-clause* | *ensures-clause*)*
named-behavior ::= `behavior` *ident* `:` *behavior-body*
behavior-body ::= (*assumes-clause* | *requires-clause* | *assigns-clause* | *ensures-clause*)*
assumes-clause ::= `assumes` *predicate* `;`
requires-clause ::= `requires` *predicate* `;`
assigns-clause ::= `assigns` *locations* `;`
locations ::= *location* (`,` *location*)* | `\nothing`
ensures-clause ::= `ensures` *predicate* `;`
decreases-clause ::= `decreases` *term* (`for` *ident*)$^?$ `;`

Figure 2.4: Grammar of function contracts

```
\round_up_double

\round_down_float

\round_up

\nearest_away


option: type rounding_mode = Nearest_even | ...


\round_float : rounding_mode , real -> float

\round_double
```

Quantification

Quantification over a variable of type `real` is of course usual quantification over real numbers.

Quantification over float (resp. double) types is allowed too, and are supposed to range over all real numbers representable as floats (resp doubles). In particular, this does not include NaN, +infinity and -infinity in the considered range.

2.3 Function contracts

The Figure 2.4 shows a grammar for function contracts in general. The grammar for non-terminal *locations* is given below in Section 2.3.4, informally they denote C l-values.

We present below first the role of *simple contracts* (Section 2.3.2), then more generally contracts with *named behaviors* (Section 2.3.3). The role of `decreases` clause is presented later in Section 2.5. Prior to this, in next section we introduce two additional constructs in the grammer for terms.

term	::=	`\old (` *term* `)`	old value
	|	`\result`	result of a function
pred	::=	`\old (` *pred* `)`	

Figure 2.5: `\old` and `\result` in terms

2.3.1 Built-in constructs `\old` and `\result`

Function contracts usually need additional constructs for terms as given on figure 2.5. These are the following:

- `\old`(e) denotes the value of e in the pre-state of the function.
- `\result` denotes the returned value of the function.

`\old`(e) and `\result` can be used only in `assigns` clauses and `ensures` clauses, since `requires` clauses and `assumes` refer to the pre-state. See Section 2.4.3 for more details on scoping rules for labels.

2.3.2 Simple function contracts

A simple function contract, having no named behavior, has the following form:

```
/*@ requires P1;
  @ requires P2;
  @ assigns L1;
  @ assigns L2;
  @ ensures E1;
  @ ensures E2;
  @*/
```

The semantics of such a contract is as follows:

- The caller of the function must guarantee that it is called in a state where the property $P_1 \&\& P_2$ holds.
- The called function returns a state where the property $E_1 \&\& E_2$ holds.
- All memory locations of the pre-state that do not belong to the set $L_1 \cup L_2$ remain allocated and are left unchanged in the post-state.

Notice that the multiplicity of clauses are proposed mainly to improve readibility since the contract above is equivalent to the following simplified one:

```
/*@ requires P1&&P2;
  @ assigns L1, L2;
  @ ensures E1&&E2;
  @*/
```

Also, if no clause `requires` is given, it defaults to requiring 'true', and similarly for `ensures` clause. Giving no `assigns` clause means that locations assigned by the function are not specified, so the caller has no information at all on this function's side effects. See Section 2.3.5 for more details on default status of clauses.

Example 2.9 *The following function is given a simple contract for computation of integer square root, rounded to the floor.*

```
/*@ requires x ≥ 0;
  @ ensures \result ≥ 0;
  @ ensures \result * \result ≤ x;
  @ ensures x < (\result + 1) * (\result + 1);
  @*/
int isqrt(int x);
```

The contract means that the function must not be called with a negative argument, and in return the result satisfies the conjunction of the three predicates given in `ensures` *clauses.*

Example 2.10 *The following function is given a contract to specify it increments the value pointed by the pointer given as argument.*

```
/*@ requires \valid(p);
  @ assigns *p;
  @ ensures *p ≡ \old(*p) + 1;
  @*/
void incrstar(int *p);
```

The contract means that the function must be called with a pointer `p` *that points to a safely allocated memory location (See Section 2.7 for details on* `\valid` *built-in predicate). It does not modify any memory location except the value pointed by* `p`*, and more precisely this value is incremented by one.*

2.3.3 Contracts with named behaviors

More generally, a function contract has the following form with named behaviors (restricted to two behaviors for readability):

```
/*@ requires P;
  @ behavior b1:
  @   assumes A1;
  @   requires R1;
  @   assigns L1;
  @   ensures E1;
  @ behavior b2:
  @   assumes A2;
  @   requires R2;
  @   assigns L2;
  @   ensures E2;
  @*/
```

The semantics of such a contract is as follows:

- The caller of the function must guarantee that it is called in a state where the property P`&&`$(A_1$`==>`$R_1)$`&&`$(A_2$`==>`$R_2)$ holds.
- The called function returns a state where the properties `\old`(A_i)`==>`E_i hold for each i.
- For each i, if the function is called in a pre-state where A_i holds, then each memory location of that pre-state that do not belong to the set L_i remain allocated and is left unchanged in the post-state.

Notice that the `requires` clauses in the behaviors are proposed mainly to improve readibility (to avoid some duplication of formulas), since the contract above is equivalent to the following simplified one:

```
/*@ requires P&&(A1==>R1)&&(A2==>R2);
  @ behavior b1:
  @   assumes A1;
  @   assigns L1;
  @   ensures E1;
  @ behavior b2:
  @   assumes A2;
  @   assigns L2;
  @   ensures E2;
  @*/
```

Also, a simple contract like in Section 2.3.2 is indeed equivalent to

```
/*@ requires P;
  @ behavior <any name>:
  @   assumes \true;
  @   assigns L;
  @   ensures E;
  @*/
```

thus simple contracts are just contracts with an anonymous behavior.

Example 2.11 *In the following,* `bsearch` (t, n, v) *searches for element* v *in array* t *between indices* 0 *and* $n - 1$.

```
/*@ requires n ≥ 0 ∧ \valid(t+(0..n−1));
  @ assigns \nothing;
  @ ensures −1 ≤ \result ≤ n−1;
  @ behavior success:
  @   ensures \result ≥ 0 ⟹ t[\result] ≡ v;
  @ behavior failure:
  @   assumes t_is_sorted : ∀ integer k1, integer k2;
  @             0 ≤ k1 ≤ k2 ≤ n−1 ⟹ t[k1] ≤ t[k2];
  @   ensures \result ≡ −1 ⟹
  @     ∀ integer k; 0 ≤ k < n ⟹ t[k] ≢ v;
  @*/
int bsearch(double t[], int n, double v);
```

The precondition is to require that the array is safely allocated for at least the index from 0 *to* $n - 1$. *The two named behaviors correspond respectively to the succesful behavior and the failing behavior.*

The intention is to perform a binary search, which requires that the array t *is sorted in increasing order: this is the purpose of the predicate named* `t_is_sorted` *in the assumes clause of the behavior named* `failure`.

See 2.4.2 for a continuation of this example.

Example 2.12 *The following function illustrates the importance of different* `assigns` *clauses for each behavior.*

```
/*@ behavior p_changed:
  @   assumes n > 0;
  @   requires \valid(p);
  @   assigns *p;
  @   ensures *p ≡ n;
  @ behavior q_changed:
  @   assumes n ≤ 0;
  @   requires \valid(q);
  @   assigns *q;
  @   ensures *q ≡ n;
  @*/
void f(int n, int *p, int *q) {
  if (n > 0) *p = n; else *q = n;
}
```

Its contract means that it assigns values pointed by p or by q, conditionally on the sign of n.

Completeness of behaviors

Notice that in a contract with named behaviors, it is not required that the disjunction of the A_i is true, *i.e.* is it is not mandatory to provide a "complete" set of behaviors. Analogously, it is not required that two distinct behaviors are disjoint.

If such conditions are seeked, it is possible to add specific clauses to a contract:

```
/*@ ...
  @ complete behaviors b1,...,bn;
  @*/
```

specifies that the set of behaviors is complete *i.e.*

$$R \Rightarrow \bigvee_{1 \leq i \leq n} A_i$$

is true, where R is the pre-condition of the contract. The simplified version of that clause

```
/*@ ...
  @ complete behaviors;
  @*/
```

means that all behaviors given in the contract should be taken into account.

Analogously,

```
/*@ ...
  @ disjoint behaviors b1,...,bn;
  @*/
```

specifies that the given behaviors are pairwise disjoint *i.e.* for all distinct i and j:

$$R \Rightarrow \neg(A_i \wedge A_j)$$

is true. The simplified version of that clause

```
/*@ ...
  @ disjoint behaviors;
  @*/
```

means that all behaviors given in the contract should be taken into account.

tset	::=	`\empty`	empty set
	|	*term*	singleton
	|	*tset* `->` *id*	
	|	*tset* `.` *id*	
	|	`*` *tset*	
	|	*tset* `[` *tset* `]`	
	|	*term*$^{?}$ `..` *term*$^{?}$	range
	|	`\union (` *tset* (`,` *tset*)* `)`	union of locations
	|	`\inter (` *tset* (`,` *tset*)* `)`	intersection
	|	*tset* `+` *tset*	
	|	`(` *tset* `)`	
	|	`{` *tset* `|` *binders* (`;` *pred*)$^{?}$ `}`	set comprehension
pred	::=	`\subset (` *tset* `,` *tset* `)`	set inclusion

Figure 2.6: Grammar for sets of terms

Error cases

See Section A.2.1 for a discussion about behaviors related to error cases.

2.3.4 Memory locations

There are several places where one needs to describe a set of memory locations: in the above `assigns` clauses of functions' contracts, or in the `reads` clauses of Section 2.6. More generally, we introduce syntactic constructs to denote *sets of terms*. A *memory location* is then any set of terms denoting a set of lvalues. Moreover, a location given as argument to an `assigns` clause must be a set of modifiable lvalues, as described in Section A.1.

The grammar for denoting sets of terms is given in Figure 2.6. The semantics is given as follows, where s denotes any tset:

- `\empty` denotes the empty set
- a simple term denotes a singleton
- $s \rightarrow id$ denotes the set of $x \rightarrow id$ for each $x \in s$
- $s.id$ denotes the set of $x.id$ for each $x \in s$
- $*s$ denotes the set of $*x$ for each $x \in s$
- $s_1[s_2]$ denotes the set of $x_1[x_2]$ for each $x_1 \in s_1$ and $x_2 \in s_2$
- $t_1..t_2$ denotes the set of integers between t_1 and t_2, included.
- `union`$(s_1, \ldots, s_n)$ denotes the union of $s_1, s_2, \ldots$ and s_n
- `inter`$(s_1, \ldots, s_n)$ denotes the intersection of $s_1, s_2, \ldots$ and s_n
- $s_1 + s_2$ denotes the set of $x_1 + x_2$ for each $x_1 \in s_1$ and $x_2 \in s_2$
- (s) denotes the same set as s

- $\{s \mid b; P\}$ denotes set comprehension: set of terms denoted by s for each values of binders satisfying predicate P. Binders b are bound in s and P

Remark: assigns `\nothing` is equivalent to assigns `\empty`, it is left for convenience.

Example 2.13 *The following function sets to 0 each cell of an array.*

```
/*@ assigns t[0..n-1];
  @ assigns *(t+(0..n-1));
  @ assigns *(t+{ i | integer i ; 0 ≤ i < n });
  @*/
void reset_array(int t[],int n) {
  int i;
  for (i=0; i < n; i++) t[i] = 0;
}
```

It is annotated with three equivalent `assigns` *clauses, each one specifying that only the set of cells* $\{t[0], \ldots, t[n-1]\}$ *are modified.*

Example 2.14 *The following function increments each values stored in a linked list.*

```
struct list {
  int hd;
  struct list *next;
};

// reachability in NULL-terminated lists
/*@ predicate reachable(struct list *root, struct list *to) =
  @   root ≡ to ∨ root ≢ \null ∧ reachable(root->next,to) ;
  @*/

//@ assigns { q->hd | struct list *q ; reachable(p,q) } ;
void incr_list(struct list *p) {
  while (p) { p->hd++ ; p = p->next; }
}
```

The `assigns` *clause specifies that the set of modified memory locations is the set of fields* `q->hd` *for each pointer* `q` *reachable from* `p` *following* `next` *fields. See Section 2.6 for details about the declaration of the logic predicate* `reachable`.

2.3.5 Default contracts, multiple contracts

In C, a function can be defined only once but declared several times. It is allowed to annotate each of these declarations with contracts. Those contracts are seen as a single contract with the union of the requires clauses and the behaviors.

On the other hand a function may have no contract at all, or a contract with not all clauses. The `requires` and the `ensures` clauses default to true when none is given.

If no `assigns` clause is given, it remains unspecified. If the function under consideration has only a declaration but no body, then that means it potentially modifies "everything", hence in practice it will be impossible to verify anything about programs calling that function; in other words giving it a contract is in practice mandatory. On the other hand, if that function has a body, giving no `assigns` clause means in practice that it is left to tools to compute an over-approximation of the sets of assigned locations.

compound-statement ::= { *declaration** *statement** *assertion*$^+$ }

statement ::= *assertion statement*

assertion ::= `/*@ assert` *pred* `; */`
| `/*@ for` *id* `(, id)`* `: assert` *pred* `; */`

Figure 2.7: Grammar for assertions

2.4 Statement annotations

Annotations on C statements are of three kinds:

- Loop annotations: invariant, assigns clause, variant; allowed before any loop statement: `while`, `for`, and `do...while`.
- Assertions and logic labels; allowed before any C statement or at end of blocks.
- Statement contracts; allowed before any C statement, specifying their behavior in a similar manner as C functions.

2.4.1 Assertions

The syntax of those annotations is given Figure 2.7 under the form of a addition to the grammar of C statements.

- The `assert` p clause means that p must hold in the state before the statement.
- The form `for` $id_1, \ldots, id_k$ `:` `assert` p variant associates the assertion to the named behaviors id_i, each of them being a behavior identifier for the current function (or a behavior of an enclosing block as defined later in Section 2.4.4). It means that this assertion is meant to be valid only for the considered behaviors.

2.4.2 Loop annotations

The syntax of loop annotations is given Figure 2.8 under the form of an addition to the grammar of C statements.

Loop invariants

The semantics of loop invariants is defined as follows: a loop annotation of the form

```
/*@ loop invariant I;
  @ loop assigns L;
  @*/
...
```

specifies that

- The predicate I is initially true, that is true in the state before entering the loop (more precisely, in the case of a `for` loop: after the initialization expression).

statement ::= `/*@` *loop-annot* `*/`
`while (` *expr* `)` *statement*
| `/*@` *loop-annot* `*/`
`for (` *expr* `;` *expr* `;` *expr* `)` *statement*
| `/*@` *loop-annot* `*/`
`do` *statement* `while (` *expr* `) ;`

loop-annot ::= (*loop-invariant* | *loop-assigns*)*
loop-variant?

loop-invariant ::= `loop invariant` *pred* `;`
| `for` *id* (`,` *id*)* `: loop invariant` *pred* `;` invariant for behavior id

loop-assigns ::= `loop assigns` *locations* `;`
| `for` *id* (`,` *id*)* `: loop assigns` *locations* `;` assigns for behavior id

loop-variant ::= `loop variant` *term* `;`
| `loop variant` *term* `for` *id* `;` variant for relation id

Figure 2.8: Grammar for loop annotations

- The predicate I is an inductive invariant, that is if I is assumed true in some state where the condition c is also true, and if execution of the loop body in that state ends normally at the end of the body or with a `continue` statement, I is true in the resulting state. Notice that if the loop condition has side effects, these are included in the loop body in a suitable way:
  - for a `while` (c) s loop, I must be preserved by the side-effects of c followed by s
  - for a `for`$(init; c; step)$ s loop, I must be preserved by the side-effects of c followed by s followed by step
  - for a `do` s `while` (c); loop, I must be preserved by s followed by the side-effects of c
- At any loop iteration, any location that was allocated before entering the loop, and is not member of L in the current state, must remain allocated and have the same value as before entering the loop.

Remarks:

- The `\old` construct is not allowed in loop annotations. The `at` form should be used to refer to another state.
- When a loop exits with `break` or `return` or `goto`, it is not required that the loop invariant hold.

Example 2.15 *Here is a continuation of example 2.11. Notice the use of a loop invariant associated to a function behavior.*

```
/*@ requires n ≥ 0 ∧ \valid(t+(0..n−1));
  @ assigns \nothing;
```

```
  @ ensures −1 ≤ \result ≤ n−1;
  @ behavior success:
  @   ensures \result ≥ 0 ⟹ t[\result] ≡ v;
  @ behavior failure:
  @   assumes t_is_sorted : ∀ integer k1, int k2;
  @       0 ≤ k1 ≤ k2 ≤ n−1 ⟹ t[k1] ≤ t[k2];
  @   ensures \result ≡ −1 ⟹
  @     ∀ integer k; 0 ≤ k < n => t[k] ≢ v;
  @*/
int bsearch(double t[], int n, double v) {
  int l = 0, u = n−1;
  /*@ loop invariant 0 ≤ l ∧ u ≤ n−1;
    @ for failure: loop invariant
    @   ∀ integer k; 0 ≤ k < n ∧ t[k] ≡ v ⟹ l ≤ k ≤ u;
    @*/
  while (l ≤ u ) {
    int m = l + (u−l)/2; // better than (u+l)/2
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m − 1;
    else return m;
  }
  return −1;
}
```

Loop variants

Optionally, a loop annotation may end with a loop variant of the form

```
/*@ loop variant m;
  @*/
...
```

where m is a term of type `integer`.

The semantics is as follows: for each loop iteration that terminates normally or with `continue`, the value of m at end of the iteration must be smaller that its value at the beginning of the iteration. Moreover, its value at the beginning must be non-negative.

Notice that it means that the value of m at loop exit might be negative. This is OK because it does not compromise termination of the loop. A toy example is as follows.

Example 2.16

```
void f(int x) {
  //@ loop variant x;
  while (x ≥ 0) {
    x −= 2;
  }
}
```

It also possible to specify termination orderings other than the less-than relation on integers, using the additional `for` modifier. This is explained in Section 2.5.

```
assertion ::= /*@ invariant pred ; */
            | /*@ for id (, id)* : invariant pred ; */
```

Figure 2.9: Grammar for general inductive invariants

General inductive invariants

It is allowed to pose a inductive invariant not only at the beginning of a loop iteration. For example, it makes sense for a `do` s `while` (c)`;` loop to pose an invariant just after the loop body s. Such an invariant is just a variant of assertions, as shown in the grammar of Figure 2.9

Example 2.17 *In the following, the invariant is correct, but it would not be correct if placed as a loop invariant at the beginning of the loop.*

```
/*@ requires n > 0 ∧ \valid(t+(0..n−1));
  @ ensures \result ≡ \max(0,n−1,(\lambda integer k ; t[k]));
  @*/
double max(double t[], int n) {
  int i = 0; double m,v;
  do {
    v = t[i++];
    m = v > m ? v : m;
  }
  /*@ invariant m ≡ \max(0,i−1,(\lambda integer k ; t[k]));
    @*/
  while (i < n);
  return m;
}
```

More generally, such an invariant may occur anywhere inside a function's body. The meaning is just that the invariant is true at that point, much like an `assert`, but moreover impose that it is inductive, i.e. is preserved by a loop iteration. Several invariants are allowed at different places in the loop body. These advanced extensions can be useful for complex control flows, in particular with `goto`s.

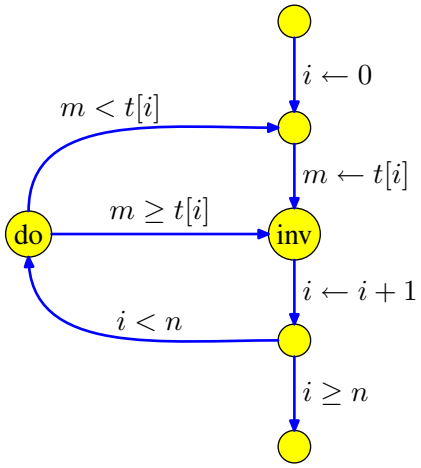
Syntactically, such an advanced loop invariant can be placed anywhere where an `assert` clause is allowed. The difference is in the semantics: a loop invariant must be *inductive*. This means that a proposition given as an invariant must not only be true at the corresponding program point for any execution of the program, but it must be also preserved independently of the context.

Example 2.18 *Here is a program annotated with an invariant inside the loop body:*

```
/*@ requires n > 0;
  @ ensures \result ≡ \max(0,n−1,\lambda integer k; t[k]);
  @*/
double max_array(double t[], int n) {
  double m; int i=0;
  goto L;
  do {
    if (t[i] > m) { L: m = t[i]; }
    /*@ invariant
      @   0 ≤ i < n ∧ m ≡ \max(0,i,\lambda integer k; t[k]);
```

```
      @*/
    i++;
  }
  while (i < n);
  return m;
}
```

The control-flow graph of the code is as follows



The invariant is inductively preserved by the two paths that goes from node inv to itself.

Example 2.19 *The program*

```
int x = 0;
int y = 10;

/*@ loop invariant 0 ≤ x < 11;
  @*/
while (y > 0) {
  x++;
  y--;
}
```

is not correctly annotated, even if it is true that x remains less than 11 during the execution. This is because it is not true that the property $x < 11$ *is preserved by the execution of* `x++ ; y--;`*. A correct loop invariant could be* `0 <= x < 11 && x+y == 10`*, true at loop entrance and preserved (under the assumption of the loop condition* `y>0`*).*

Similarly with general invariants: in

```
int x = 0;
int y = 10;

while (y > 0) {
  x++;
  //@ invariant 0 < x < 11;
  y--;
  //@ invariant 0 ≤ y < 10;
}
```

the propositions given as invariants are correct assertions but not inductive invariants: `0 <= y < 10` *is not a consequence of hypothesis* `0 < x < 11` *after executing* `y--` *; and* `0 < x < 11` *cannot be deduced from* `0 <= y < 10` *after looping back through the condition* `y>0` *and executing* `x++`*. Correct invariants could be:*

```
while (y > 0) {
  x++;
  //@ invariant 0 < x < 11 ∧ x+y ≡ 11;
  y--;
  //@ invariant 0 ≤ y < 10 ∧ x+y ≡ 10;
}
```

2.4.3 Built-in construct `\at`

Statement annotations usually need another additional construct $\texttt{\at}(e, id)$ refering to the value of the expression e in the state at label id. The label id can be either a regular C label, or a label added within a ghost statement as described in Section 2.11. This label must be declared in the same function as the occurrence of $\texttt{\at}(e, id)$, but unlike `goto`s, more restrictive scoping rules must be respected:

- the label id must occur before the occurrence of $\texttt{\at}(e, id)$ in the source.
- the label id must not be inside a inner block.

These rules are exactly the same rules as the visibility of some local variable within a statement (see [11], Section A11.1)

Default logic labels

There are three predefined logic labels: `Pre`, `Here` and `Old`. $\texttt{\old}(e)$ is in fact a syntactic sugar for $\texttt{\at}(e, Old)$.

- The label `Pre` is visible in all statement annotations, and refers to the pre-state of the function it occurs in.
- The label `Here` is visible in all statement annotations, where it refers to the state where the annotation appears; and in all contracts, and refers to the pre-state for the `requires` clause and the post-state for other clauses.
- The label `Old` is visible in `assigns` and `ensures` clauses of all contracts (both for functions and for statement contracts described below in Section 2.4.4), and refers to the pre-state of this contract.

Remark that no logic label are visible in global logic declaration such as lemmas, axioms, definition of predicate or logic functions. When such an annotation needs to refer to a given memory state, it has to be given a label binder: this is described in Section 2.6.5.

Example 2.20 *The code below implements the famous extended Euclide's algorithm for computation of the greatest common divisor of two integers x and y, computing at the same time two* Bézout coefficients *p and q such that $p \times x + q \times y = gcd(x, y)$.*

The loop invariant for the Bézout property needs to refers to the value of x and y in the pre-state of the function

statement ::= `/*@` *statement-contract* `*/` *statement*

statement-contract ::= `(for` *id* `(,` *id*`)*)?` *simple-behavior named-behavior* abrupt-clause**

abrupt-clause ::= `breaks` *predicate* `;`
| `continues` *predicate* `;`
| `returns` *predicate* `;`

Figure 2.10: Grammar for statement contracts

```
/*@ requires x ≥ 0 ∧ y ≥ 0;
  @ behavior bezoutProperty:
  @   ensures (*p)*x+(*q)*y ≡ \result;
  @*/
int extended_Euclide(int x, int y, int *p, int *q) {
  int a = 1, b = 0, c = 0, d = 1;
  /*@ loop invariant x ≥ 0 ∧ y ≥ 0 ;
    @ for bezoutProperty: loop invariant
    @    a*\at (x,Pre)+b*\at (y,Pre) ≡ x ∧
    @    c*\at (x,Pre)+d*\at (y,Pre) ≡ y ;
    @ loop variant y;
    @*/
  while (y > 0) {
    int r = x % y;
    int q = x / y;
    int ta = a, tb = b;
    x = y; y = r;
    a = c; b = d;
    c = ta − c * q; d = tb − d * q;
  }
  *p = a; *q = b;
  return x;
}
```

2.4.4 Statement contracts

The grammar for statement contracts is given in Figure 2.10. It is very similar to function behaviors, but without `decreases` clause. Additionally, behavior can refer to enclosing named behaviors, with the form `for` *id* `:` `...`. Such behaviors are meant to be valid only for the corresponding function behavior, in particular only under the corresponding `assumes` clause.

The clauses `breaks`, `continues` and `returns` are specific way for stating a property on the program state when the annotated statement terminates abruptly with the corresponding C statements, `break`, `continue` and `return`. For the `returns` case, the `\result` construct is allowed and bound to the value returned (if not a void function).

Example 2.21 *Here is a toy example which illustrates each of these special clauses for statement contracts.*

```
int f(int x) {

  while (x > 0) {

    /*@ breaks x % 11 ≡ 0 ;
      @ continues (x+1) % 11 ≢ 0 ∧ x % 7 ≡ 0;
      @ returns (\result+2) % 11 ≢ 0 ∧ (\result+1) % 7 ≢ 0
      @         ∧ \result % 5 ≡ 0;
      @ ensures (x+3) % 11 ≢ 0 ∧ (x+2) % 7 ≢ 0 ∧ (x+1) % 5 ≢ 0;
      @*/
    {
      if (x % 11 ≡ 0) break;
      x--;
      if (x % 7 ≡ 0) continue;
      x--;
      if (x % 5 ≡ 0) return x;
      x--;
    }
  }
  return x;
}
```

2.5 Termination

The property of termination concerns both loops and recursive function calls. Termination is guaranteed by attaching a measure function to each loop and each recursive functions.

By default, a measure is a function into integers, and measures are compared using standard less-than ordering (Section 2.5.1). It is also possible to define measures into other domains and/or using a different ordering relation (Section 2.5.2).

2.5.1 Integer measures

Functions are annotated with integer measures with the form

```
//@ decreases e ;
```

and loops are annotated similarly with the form

```
//@ loop variant e;
```

where the logic expression e has type `integer`. For recursive calls, or for loops, this expression must decrease for the relation R defined by

$$R(x, y) ::= x > y \;\&\&\; x \geq 0$$

In other words, the measure must be a decreasing sequence of integers which remain non-negative, except possibly for the last value of the sequence (See example 2.16).

Example 2.22 *In Example 2.15, a loop variant* `u-l` *decreases at each iteration, and remains non-negative, except at the last iteration where it may become negative.*

2.5.2 General measures

More general measures on other types can be provided, using the keyword `for`. For functions it becomes

`//@ decreases` e `for` R`;`

and for loops:

`//@ loop variant` e `for` R`;`

In those cases, the logic expression e has some type τ and R must be relation on τ, that is a logic binary predicate declared by the form

`//@ predicate` $R(\tau\ x, \tau\ y)\cdots$

(see Section 2.6 for details). Of course, to guarantee termination, it must be proved that R is a well-founded relation.

Example 2.23 *The following is a toy example illustrating a variant annotation using a pair of integers, ordered lexicographically.*

```
//@ ensures \result ≥ 0;
int dummy();

//@ type intpair = (integer,integer)

/*@ predicate lexico(intpair p1, intpair p2) =
  @    \let (x1,y1) = p1 ;
  @    \let (x2,y2) = p2 ;
  @       x1 < x2 ∧ 0 ≤ x2 ∨
  @       x1 ≡ x2 ∧ 0 ≤ y2 ∧ y1 < y2;
  @*/

//@ requires x ≥ 0 ∧ y ≥ 0;
void f(int x,int y) {
  /*@ loop invariant x ≥ 0 ∧ y ≥ 0;
    @ loop variant (x,y) for lexico;
    @*/
  while (x > 0 ∧ y > 0) {

    if (dummy()) {
      x--; y = dummy();
    }
    else y--;
  }
}
```

2.5.3 Recursive function calls

The precise semantics of measures on recursive calls, especially in the general case of mutually recursive functions, is given as follows. Each *cluster* of mutually recursive functions must be identified: each of them corresponds to a strongly connected component of the call graph.

For a given cluster, each function must be annotated with a decreases clause with the same relation R (syntactically). Then, in the body of any function f of that cluster, any recursive call to a function g must occur in a state where the measure attached to g is smaller (w.r.t R) than the measure of f in the pre-state of f. This also applies when g is f itself.

Example 2.24 *This are the classical factorial and Fibonacci functions*

```
/*@ requires n ≤ 12;
  @ decreases n;
  @*/
int fact(int n) {
  if (n ≤ 1) return 1;
  return n * fact(n−1);
}

//@ decreases n;
int fib(int n) {
  if (n ≤ 1) return 1;
  return fib(n−1) + fib(n−2);
}
```

Example 2.25 *This example illustrate mutual recursion*

```
//@ decreases n;
int even(int n) {
  if (n ≡ 0) return 1;
  return odd(n−1);
}

//@ decreases x;
int odd(int x) {
  if (x ≡ 0) return 0;
  return even(x−1);
}
```

2.6 Logic specifications

The language of logic expressions used in annotations can be extended by declaration of new logic types, and new constants, logic functions and predicates. These declarations follows the classical setting of *algebraic specifications*, in particular new functions and predicates may be either *defined* by explicit expressions, or *axiomatized* by a set of axioms. The grammer for these declarations is given Figure 2.11.

The difference between a lemma and an axiom is that a lemma is supposed to be deductible from axioms: a redundant axiom in some sense.

2.6.1 Polymorphic logic types

EXPERIMENTAL

We consider here an algebraic specification setting based on multi-sorted logic, where types can be *polymorphic* that is parameterized by other types. For example, one may declare the type of polymorphic lists as

```
//@ type list<A>;
```

and then can consider for example list of integers (`list <integer>`), list of pointers (e.g. `list <char*>`), list of list of reals (`list <list <real> >`), etc.

C-global-decl	::=	`/*@` *logic-decl*$^+$ `*/`	
logic-decl	::=	*logic-type-decl*	
	|	*logic-const-decl* | *logic-const-def*	
	|	*logic-predicate-decl* | *logic-predicate-def*	
	|	*logic-function-decl* | *logic-function-def*	
	|	*lemma-def*	
type-var	::=	*id*	
type-var-binders	::=	< *type-var* (, *type-var*)* >	
logic-type-decl	::=	`type` *logic-type* `;`	
logic-type	::=	*id*	
	|	*id type-var-binders*	polymorphic type
logic-type-expr	::=	*type-var*	type variable
	|	*id* < *type-expr* (, *type-expr*)* >	polymorphic type
poly-id	::=	*id*	normal identifier
	|	*id type-var-binders*	identifier for polymorphic object
logic-const-def	::=	`logic` *type-expr poly-id* = *term* `;`	
logic-function-def	::=	`logic` *type-expr poly-id parameters* = *term* `;`	
logic-predicate-def	::=	`predicate` *poly-id parameters* = *pred* `;`	
logic-const-decl	::=	`logic` *type-expr poly-id* `;`	
logic-function-decl	::=	`logic` *type-expr poly-id parameters* `;`	
logic-predicate-decl	::=	`predicate` *poly-id parameters*$^?$ `;`	
parameters	::=	`(` *parameter* (, *parameter*)* `)`	
parameter	::=	*type-expr variable-ident*	
lemma-def	::=	`axiom` *poly-id* `:` *pred* `;`	
	|	`lemma` *poly-id* `:` *pred* `;`	

Figure 2.11: Grammar for logic declarations

The grammar of Figure 2.11 contains the additional rules for declaring polymorphic types, and for polymorphic type expressions.

2.6.2 Recursive logic definitions

The explicit definitions of logic functions and predicates can be recursive. Declarations in the same bunch of logic declarations are implicitely mutually recursive, so mutually recursive functions are possible too.

Example 2.26 *The following logic declaration*

term	::=	`\lambda` *binders* `;` *term*	abstraction
	|	*extended-quantifier* `(` *term* `;` *term* `;` *term* `)`	
extended-quantifier	::=	`\max` | `\min` | `\sum` | `\product` | `\numof`	

Figure 2.12: Grammar for higher-order constructs

```
/*@ logic int max_index{L}(int t[],int n) =
  @   (n≡0) ? 0 :
  @   (t[n−1]≡0) ? n : max_index(t, n−1);
  @*/
```

defines a logic function which returns the maximal index i between 0 and $n-1$ such that $t[i]=0$.

Notice that there is no syntactic condition on such recursive definitions, such as limitation to primitive recursion like e.g. in Coq [6]. In essence, a recursive definition of the form $f(args)e$ where f occurs in expression e is just a shorcut for a declaration of f followed by an axiom `\forall` $args, f(args) = e$. In other words, recursive definitions are not guaranteed to be consistent, in the same way that axioms may introduce inconsistency. Of course, tools might provide a way to check consistency.

2.6.3 Higher-order logic constructions

EXPERIMENTAL

Figure 2.12 introduces new term constructs for higher-order logic.

Abstraction The term `\lambda` $\tau_1\ x_1, \ldots, \tau_n\ x_n$ `;` t denotes the n-ary logic function which maps $x_1, \ldots, x_n$ to t. It has the same precedence as `\forall` and `\exists`

Extended quantifiers Terms $\backslash quant(t_1, t_2, t_3)$ where *quant* is `max`, `min`, `sum`, `product`, or `numof`, are extended quantifications. t_1 and t_2 must have type `integer`, and t_3 must be a unary function with an integer argument, and a numeric value (integer or real) except for `\num_of` for which it should have a boolean value. Their meanings are given as follows:

$$\begin{aligned}
\backslash\texttt{max}(i,j,f) &= \max\{f(i), f(i+1), \ldots, f(j)\} \\
\backslash\texttt{min}(i,j,f) &= \min\{f(i), f(i+1), \ldots, f(j)\} \\
\backslash\texttt{sum}(i,j,f) &= f(i) + f(i+1) + \cdots + f(j) \\
\backslash\texttt{product}(i,j,f) &= f(i) \times f(i+1) \times \cdots \times f(j) \\
\backslash\texttt{numof}(i,j,f) &= \#\{k \mid i \le k \le j \ \&\& \ f(k)\} \\
&= \backslash\texttt{sum}(i,j,\backslash\texttt{lambda integer } k; f(k)?1:0)
\end{aligned}$$

If $i > j$ then `\sum` and `\numof` above are 0, `\product` is 1, and `\max` and `\min` are unspecified (see Section 2.2.2).

Example 2.27 *Function that sums the element of an array of doubles.*

```
/*@ requires n ≥ 0 ∧ \valid(t+(0..n−1)) ;
  @ ensures \result ≡ \sum(0,n−1,\lambda int k; t[k]);
  @*/
double array_sum(double t[],int n) {
  int i;
```

logic-type-decl	::=	`type` *logic-type* = *logic-type-def* `;`	
logic-type-def	::=	*record-type* | *sum-type*	
	|	*type-expr*	type abbreviation
record-type	::=	`{` *type-expr id* (`;` *type-expr id*)* `;`$^?$ `}`	
sum-type	::=	`|`$^?$ *constructor* (`|` *constructor*)*	
constructor	::=	*id*	constant constructor
	|	*id* `(` *type-expr* (`,` *type-expr*)* `)`	non-constant constructor
type-expr	::=	`(` *type-expr* (`,` *type-expr*)$^+$ `)`	product type
term	::=	*term* `.` *id*	record field access
	|	`\match` *term* `{` *match-cases* `}`	pattern-matching
	|	`(` *term* (`,` *term*)$^+$ `)`	tuples
	|	`{` (`.` *id* = *term* `;`)$^+$ `}`	records
	|	`\let` `(` *id* (, *id*)$^+$ `)` = *term* `;` *term*	
match-cases	::=	*match-case*$^+$	
match-case	::=	`case` *pat* `:` *term*	
pat	::=	*id*	constant constructor
	|	*id* `(` *pat* (`,` *pat*)* `)`	non-constant constructor
	|	*pat* `|` *pat*	or pattern
	|	_	any pattern
	|	*cte*	numeric constant
	|	`{` (`.` *id* = *pat*)* `}`	record pattern
	|	`(` *pat* (`,` *pat*)* `)`	tuple pattern
	|	*pat* `as` *id*	pattern binding

Figure 2.13: Grammar for concrete logic types and pattern-matching

```
  double s = 0.0;
  /*@ loop invariant 0 ≤ i ≤ n;
    @ loop invariant s ≡ \sum(0,i−1,\lambda int k; t[k]);
    @ loop variant n−i;
  */
  for(i=0; i < n; i++) s += t[i];
  return s;
}
```

2.6.4 Concrete logic types

EXPERIMENTAL

Logic types may not only be declared but also defined concretely, either under thr form of record types, or of sum types. These definitions may be recursive. With record types, the field access notation

$t.id$ can be used, and for sum types, a pattern-matching construction is available. The grammar rules for these additional constructions are given Figure 2.13

Example 2.28 *The declaration*

```
//@ type list<A> = Nil | Cons(A,list<A>);
```

introduce a concrete definition of finite lists. The logic definition

```
/*@ logic integer list_length<A>(list<A> l) =
  @     \match l ;
  @       | Nil : 0
  @       | Cons(h,t) : 1+list_length(t) ;
  @*/
```

defines the length of a list by recursion and pattern-matching.

2.6.5 Hybrid functions and predicates

These are functions or predicates which take both C types and logic types as arguments. Such a predicate can either be defined with the same syntax as before, or simply declared, but in the latter case the declaration should usually be augmented with a `reads` clause, with the syntax given on Figure 2.14 which extend the one of Figure 2.11. This feature is useful when a function or a predicate is not easily definable, but can be more easily axiomatized.

Both with direct logic definitions or with logic declarations and axioms, an hybrid function will usually depend on one or more program points, because it depends upon memory states, via expressions such as:

- pointer dereferencing: `*p`, `p->f`
- array accesses: `t[i]`
- address operator: `&x`
- built-in predicates depending on memory: `\valid`

To make such a definition safe, it is mandatory to add after the declared identifier a set of labels, between curly braces. Expressions as above must then be enclosed into the `\at` construct to refer to a given label. Nevertheless, to ease reading such logic expressions, it is allowed to omit a label whenever there is only one label in the context.

Example 2.29 *The following declares a predicate which tells the number of positive elements in an array of doubles, between given indexes*

```
/*@ logic integer num_of_pos{L}(int i,int j, double t[])
  @   reads t[..]
  @*/

/*@ axiom num_of_pos_empty{L} :
  @   ∀ int i, j, double t[];
  @       i > j ⟹ num_of_pos(i,j,t) ≡ 0
  @*/

/*@ axiom num_of_pos_true_case{L} :
```

logic-function-decl	::=	`logic` *type-expr poly-id parameters* *reads-clause* `;`	
logic-predicate-decl	::=	`predicate` *poly-id parameters* *reads-clause* `;`	
reads-clause	::=	`reads` *locations*	
logic-function-def	::=	`logic` *type-expr poly-id parameters* *reads-clause* = *term* `;`	
logic-predicate-def	::=	`predicate` *poly-id parameters* *reads-clause* = *pred* `;`	
poly-id	::=	*id*	normal identifier
	|	*id type-var-binders*	identifier for polymorphic object
	|	*id label-binders*	normal identifier with labels
	|	*id label-binders type-var-binders*	polymorphic identifier with labels
label-binders	::=	{ *id* $(, id)^*$ }	

Figure 2.14: Grammar for logic declarations with reads clauses

```
  @   ∀ int i, j, double t[];
  @       i ≤ j ∧ t[j] > 0.0 ⟹
  @         num_of_pos(i,j,t) ≡ num_of_pos(i,j−1,t) + 1;
  @*/

/*@ axiom num_of_pos_false_case{L} :
  @   ∀ int i, j, double t[];
  @       i ≤ j ∧ ! t[j] > 0) ⟹
  @         num_of_pos(i,j,t) ≡ num_of_pos(i,j−1,t);
  @*/
```

It should be noticed that without the reads clauses, this axiomatization would be inconsistent, since the predicate would not depend on the values stored in t, hence the two last axioms would say both that $a = b + 1$ *and* $a = b$ *for some* a *and* b*.*

Example 2.30

```
/* nb_occ(t,i,j,e) gives the number of occurrences of e in t[i..j]
 * (in a given memory state labelled L)
 */

/*@ logic integer nb_occ{L}(double t[], integer i, integer j,
  @                          double e)
  @   reads t[..];
  @*/
/* Notice that without label {L}, t[..] would be rejected.
 * With {L}, it is indeed a shortcut for \at (t[..],L).
 */
```

```
/*@ axiom nb_occ_empty{L} :
  @   ∀ double t[], integer i, integer j, double e;
  @     i > j ⟹ nb_occ(t,i,j,e) ≡ 0;
  @*/
// without {L}, term nb_occ(t,i,j,e) would be rejected

/*@ axiom nb_occ_true{L} :
  @   ∀ double t[], integer i, integer j, double e;
  @     i ≤ j ∧ t[i] ≡ e ⟹
  @       nb_occ(t,i,j,e) ≡ nb_occ(t,i,j−1,e) + 1;
  @*/
// without {L}, term ti would be rejected, here it is \at (ti,L)

/*@ axiom nb_occ_false{L} :
  @   ∀ double t[], integer i, integer j, double e;
  @     i ≤ j ∧ t[i] ≢ e ⟹
  @       nb_occ(t,i,j,e) ≡ nb_occ(t,i,j−1,e);
  @*/




/* permut{L1,L2}(t1,t2,n) is true whenever t1[0..n−1] in state L1
 * is a permutation of  t2[0..n−1] in state L2
 */

/*@ predicate permut{L1,L2}(double t1[], double t2[], integer n)
  @   reads \at (t1[..],L1), \at (t2[..],L2);
  @*/


/*@ axiom permut_refl{L} :
  @   ∀ double t[], integer n; permut{L,L}(t,t,n);
  @*/

/*@ axiom permut_sym{L1,L2} :
  @   ∀ double t1[], double t2[], integer n;
  @     permut{L1,L2}(t1,t2,n) ⟹ permut{L2,L1}(t2,t1,n) ;
  @*/

/*@ axiom permut_trans{L1,L2,L3} :
  @   ∀ double t1[], double t2[], double t3[], integer n;
  @     permut{L1,L2}(t1,t2,n) ∧ permut{L2,L3}(t2,t3,n)
  @     ⟹ permut{L1,L3}(t1,t3,n) ;
  @*/

/*@ axiom permut_exchange{L1,L2} :
  @   ∀ double t1[], double t2[], integer i, integer j, integer n;
  @       \at (t1[i],L1) ≡ \at (t2[j],L2) ∧
```

```
@        \at (t1[j],L1) ≡ \at (t2[i],L2) ∧
@        (∀ integer k; 0 ≤ k < n ∧ k ≠ i ∧ k ≠ j ⟹
@             \at (t1[k],L1) ≡ \at (t2[k],L2))
@        ⟹ permut{L1,L2}(t1,t2,n);
@*/
```

2.6.6 Specification Modules

Specification modules can be provided to encapsulate several logic definitions, for example

```
/*@ module List {
  @
  @   type list<A> = Nil | Cons(A , list<A>);
  @
  @   logic integer length<A>(list<A> l)  =
  @     \match l ;
  @       | Nil : 0
  @       | Cons(h,t) : 1+length(t) ;
  @
  @   logic A fold_right<A,B>((A -> B -> B) f, list<A> l, B acc) =
  @     \match l ;
  @       | Nil : acc
  @       | Cons(h,t) : f(h,fold(f,t,acc)) ;
  @
  @   logic list<A> filter<A>((A -> boolean) f, list<A> l) =
  @     fold_right((\lambda A x, list<A> acc;
  @       f(x) ? Cons(x,acc) : acc), Nil) ;
  @
  @ }
  @*/
```

Module components are then accessible using a qualified notation like `List::length`.

Predefined algebraic specifications can be provided as libraries (see section 3), and imported using a construct like

```
//@ open List;
```

where the file `list.acsl` contains logic definitions, like the `List` module above.

2.7 Pointers and physical adressing

2.7.1 Memory blocks and pointer dereferencing

- `\base_addr` base address of an allocated pointer

  `\base_addr : 'a * → char*`

- `\block_length` length of the allocated block of a pointer

  `\block_length : 'a * → size_t`

- `\valid` is a built-in predicate which applies to a set of terms of some pointer type. `\valid`(s) is true whenever dereferencing any $p \in s$ is safe.

Some shortcuts are provided:

- `\null` is an extra notation for the null pointer (i.e. a shortcut for `(void*)0`)
- `\offset`(p) returns the offset between p and its base address

$$\begin{array}{rcl} \texttt{\textbackslash offset} & : & \texttt{'a} * \rightarrow \texttt{size\_t} \\ \texttt{\textbackslash offset}(p) & = & (\texttt{char}*)p - \texttt{\textbackslash base\_addr}(p) \end{array}$$

the following property holds: for any set of pointers s, `\valid`(s) if and only if for all $p \in s$:

$$\texttt{\textbackslash offset}(p) \geq 0 \wedge \texttt{\textbackslash offset}(p) + \texttt{sizeof}(*p) \leq \texttt{\textbackslash block\_length}(p)$$

2.7.2 Separation

EXPERIMENTAL

`\separated`$(loc_1, .., loc_n)$: means that for each $i \neq j$, the intersection of loc_i and loc_j is empty. Each loc_i is a set of terms as defined in Section 2.3.4.

Locations as first-class values

Locations, as defined in Section 2.3.4, can be used as first-class values in annotations. They have the built-in type $loc < A >$ where A is the type of values in that locations.

Example 2.31 *Here is an example where we defined the* footprint *of a structure.*

```
struct S {
  char *x;
  int *y;
};

//@ logic loc<void*> footprint(struct S s) = union (s.x,s.y) ;
```

This logic function can be used as argument of `\separated`.

Note: in other words, a clause assigns l1,..,ln could be transformed into an post-condition

$$\forall char * p; \texttt{\textbackslash separated}(\texttt{union}(l1, .., ln), *p) ==> *p == \texttt{\textbackslash old}(*p)$$

2.7.3 Allocation and deallocation

EXPERIMENTAL

- built-in predicate `\fresh`, specifying in a post-condition that a pointer was not allocated in the pre-state.
- built-in predicate `\freed`, specifying in a post-condition that a pointer was allocated in the pre-state but not anymore.

declaration ::= `/*@` *data-inv-decl* `*`

data-inv-decl ::= *data-invariant* | *type-invariant*

data-invariant ::= *inv-strength*? `data invariant` *id* `:` *pred* `;`

type-invariant ::= *inv-strength*? `type invariant` *id* `(` *C-type-expr id* `)` `=` *pred* `;`

inv-strength ::= `weak` | `strong`

Figure 2.15: Grammar for declarations of data invariants

2.8 Abnormal termination

EXPERIMENTAL

It is used to give behavioral properties to the `main` function or to any function that may exit the program, e.g. calling `exit` function.

Such a behavior can be written with the form

```
exit_behavior
assumes
assigns
ensures ...
```

where in ensures clause, `\result` is bound to the return code, e.g the value returned by `main` or the argument passed to `exit`.

2.9 Dependencies information

EXPERIMENTAL

An extended syntax of `assigns` clauses allows to give dependencies:

//@ assigns $loc_1, \ldots, loc_n$ \from $loc'_1, \ldots, loc'_k$;

Moreover, dependencies can be made precise using *functional expressions*:

//@ assigns loc \from $loc'_1, \ldots, loc'_k$ = term ;

2.10 Data invariants

Data invariants are properties on data that are supposed to hold permanently during the life time of these data. In ACSL, we distinguish between:

- *global* invariants and *type* invariants: the former apply on global variables, whereas the latter are associated to static types, and apply on any variables of this type.
- *strong* invariants and *weak* invariants: strong invariants must be valid really at any any time of the program execution (more precisely at any *sequence point* as defined in the norm), whereas weak invariants are valid at *function boundaries* (function entrance and exit) but can be violated in between.

The syntax for declaring data invariants is given in Figure 2.15. The strength modifier defaults to `weak`.

Example 2.32 *In the following example, we declare*

1. *a weak global invariant* `a_is_positive` *which specifies that the global variable* `a` *should remains positive (weakly, so this property might be violated temporarily between functions calls)*
2. *a strong type invariant for type* `temperature`
3. *a weak type invariant on structures* `S`.

```
int a;
//@ data invariant a_is_positive: a > 0 ;

typedef double temperature;
/*@ strong type invariant temp_in_celsius(temperature t) =
  @   t ≥ −273.15  ;
  @*/

struct S {
  int f;
};
//@ type invariant S_f_is_positive(struct S s) = s.f > 0 ;
```

2.10.1 Semantics

The distinction between strong and weak invariants is on the sequence points were the property is supposed to hold. The distinction between the global invariants and the type invariants is the set of values on which they are supposed to hold.

- Weak global invariants are properties which applies to global data and hold at any function entrance and function exit.
- Strong global invariants are properties which applies to global data and hold at any step of execution (starting after initialisation of these data).
- A weak type invariant on a type τ is supposed to hold at any function entrance and exit, and applies to any global variable or formal parameter which has static type τ. If the result of the function is of type τ, the result is also supposed to satisfy its invariant at function exit.
- A strong type invariant on a type τ is supposed to hold at any state of execution, and applies to any global variable, local variable, or formal parameter which has static type τ. If the result of the function is of type τ, the result is also supposed to satisfy its strong invariant at function exit

Example 2.33 *The following example illustrates the use of a data invariant on a local static variable*

```
void out_char(char c) {

  static int col = 0;
  //@ data invariant I : 0 ≤ col ≤ 79;

  col++;
  if (col ≥ 80) col = 0;

}
```

Example 2.34 *Here is a longer example, the famous Dijkstra's Dutch flag algorithm.*

```
typedef enum { BLUE, WHITE, RED } color;
/*@ type invariant isColor(color c) =
  @   c ≡ BLUE ∨ c ≡ WHITE ∨ c ≡ RED ;
  @*/

/*@ predicate permut{L1,L2}(color t1[], color t2[], integer n)
  @   reads \at (t1[..],L1), \at (t2[..],L2);
  @*/

/*@ requires \valid(t+i) ∧ \valid(t+j);
  @ assigns t[i],t[j];
  @ ensures t[i] ≡ \old(t[j]) ∧ t[j] ≡ \old(t[i]);
  @*/
void swap(color t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}

typedef struct flag {
  int n;
  color *colors;
} flag;
/*@ type invariant is_colored(flag f) =
  @   f.n ≥ 0 ∧ \valid(f.colors+(0..f.n-1)) ∧
  @   ∀ integer k; 0 ≤ k < f.n ⟹ isColor(f.colors[k]) ;
  @*/

/*@ predicate isMonochrome{L}(color t[], integer i, integer j,
  @                           color c) =
  @   ∀ integer k; i ≤ k ≤ j ⟹ t[k] ≡ c ;
  @*/

/*@ assigns f.colors[0..f.n-1];
  @ ensures
  @   ∃ integer b, integer r;
  @     isMonochrome(f.colors,0,b-1,BLUE) ∧
  @     isMonochrome(f.colors,b,r-1,WHITE) ∧
  @     isMonochrome(f.colors,r,f.n-1,RED) ∧
  @     permut{Old,Here}(f.colors,f.colors,f.n-1);
  @*/
void dutch_flag(flag f) {
  color *t = f.colors;
  int b = 0;
  int i = 0;
  int r = f.n;
  /*@ loop invariant
    @   (∀ integer k; 0 ≤ k < f.n ⟹ isColor(t[k])) ∧
    @   0 ≤ b ≤ i ≤ r ≤ f.n ∧
```

declaration	::=	*C-declaration*	
	|	`/*@ model` *C-declaration* `*/`	model variable
struct-declaration	::=	*C-struct-declaration*	
	|	`/*@ model` *C-struct-declaration* `*/`	model field

Figure 2.16: Grammar for declarations of model variables and fields

```
    @   isMonochrome(t,0,b-1,BLUE) ∧
    @   isMonochrome(t,b,i-1,WHITE) ∧
    @   isMonochrome(t,r,f.n-1,RED) ∧
    @   permut{Pre,Here}(t,t,f.n-1);
    @ loop assigns b,i,r,t[0 .. f.n-1];
    @ loop variant r - i;
    @*/
  while (i < r) {
    switch (t[i]) {
    case BLUE:
      swap(t, b++, i++);
      break;
    case WHITE:
      i++;
      break;
    case RED:
      swap(t, --r, i);
      break;
    }
  }
}
```

2.10.2 Model variables and model fields

A model variable is a variable introduced in the specification, whose type is a logic type. Analogously, structures may have *model fields*. These are used to provide abstract specifications to functions whose concrete implementation must remain private.

Syntax of declaration of model variables and fields is given on Figure 2.16. It is presented as additions to regular C grammar to variable declarations and struct field declarations.

Informal semantics of model variables is as follows:

- Model variables can only appear in specifications. They are not lvalues, thus cannot be assigned directly (unlike ghost variables, see below).
- Nevertheless a function contract might say that a model variable is assigned.
- When a function contract mentions model variables:
  - the precondition is implicitely existentially quantified over those variables
  - the post-conditions are universally quantified over the old values of model variables, and existentially quantified over the new values.

Thus, in practice the only way of proving that a function body satisfies a contract with model variables is to provide an invariant relating model variables and concrete variables, as in the example below.

Example 2.35 *Here is an example of a specification of a function which generates fresh integers. The contract is given in term of a model variable which is intended to represent the set of "forbidden" values, e.g. that already have been generated.*

```
/* public interface */

//@ open Set;

//@ model set<integer> forbidden = emptyset;

/*@ assigns forbidden;
  @ ensures  in(\result,\old(forbidden)
  @   ∧ in(\result,forbidden) ∧ subset(\old(forbidden),forbidden);
  @*/
int gen();
```

The contract is expressed abstractly, telling that

- *the forbidden set of values is modified*
- *the result number is not in the set of forbidden values, thus is "fresh"*
- *the new set of forbidden contains both the returned value and the previously forbidden values.*

An implementation of this function might be as follows, where a decision has been made to generate values in increasing order, so that it suffices to record the last value generated. This decision is made explicit by an invariant.

```
/* implementation */

int gen() {
  static int x = 0;
  /*@ data invariant I: ∀ integer k;
    @    Set::mem(k,forbidden) ⟹ x > k;
    @*/
  return x++;
}
```

Remarks Although the syntax of model variables is close to JML model variables, they differ in the sense that the type of a model variable is a logic type, not a program type. Also, the semantics above is closer to the one of the B machines. It has to be noticed that program verification with model variables does not have a well-establish theoretical background [17, 15], so we deliberately don't provide a precise semantics in this document .

2.11 Ghost variables and statements

Ghost variables and statements are like C variables and statements, but visible only in the specifications. They are introduced by the `ghost` keyword at the beginning of the annotation (i.e. `/*@ ghost ... */` or `//@ ghost ...` for a one-line ghost code, as mentionned in section 1.2). The grammar is given in Figure 2.17, in which only the first form of annotation is used. In this figure, the *C-** non-terminals refer to the corresponding grammar rules of the ISO standard, without any ACSL extension. Any non terminal of the form *ghost-non-term* for which no definition is given in the figure represents the corresponding *C-non-term* entry, in which any *entry* is substituted by *ghost-entry*.

The variations with respect to the C grammar are the following:

- Comments must be introduced by `//` and extend until the end of the line (the ghost code itself is placed inside a C comment. `/* ... */` would thus lead to incorrect C code).
- It is however possible to write multi-line annotations for ghost code. These annotations are enclosed between `/@` and `@/`. As in normal annotations, `@`s at the beginning of a line and at the end of the comment (before the final `@/`) are considered as blank.
- Logical types, such as `integer` or `real` are authorized in ghost code.
- A non-ghost function can take ghost parameters. If such a ghost clause is present in the declarator, then the list of ghost parameters must be non-empty and fixed (no vararg ghost). The call to the function must then provide the appropriate number of ghost parameters.
- Any non-ghost *if-statement* which does not have a non-ghost `else` clause can be augmented with a ghost one. Similarly, a non-ghost `switch` can have a ghost `default:` clause if it does not have a non-ghost one (there are however semantical restrictions for valid ghost labelled statements in a switch, see next paragraph for details).

Semantics of Ghost Code The question of semantics is essential for ghost code. Informally, the semantics requires that ghost statements do not change the regular program execution. This implies several conditions, including e.g:

- Ghost code cannot modify a non-ghost C variable.
- Ghost code cannot modify a non-ghost structure field.
- If p is a ghost pointer pointing to a non-ghost memory location, then it is forbidden to assign $*p$.
- Body of a ghost function is ghost code, hence do not modify non-ghost variables or fields.
- If a non-ghost C function is called in ghost code, it must not modify non-ghost variables or fields.
- If a structure has ghost fields, the `sizeof` of the structure is the same has the structure without ghost fields. Also, alignment of fields remains unchanged.
- The control-flow graph of a function must not be altered by ghost statements. In particular, no ghost `return` can appear in the body of a non-ghost function. Similarly, ghost `goto`, `break`, and continue `continue` cannot jump outside of the innermost enclosing non-ghost block.

Semantics is specified as follows. First, one has to think that program execution with ghost code involves a *ghost memory heap* and a *ghost stack*, disjoint from the regular heap and stack. Ghost variables lie in the ghost heap, so as the ghost field of structures. Thus, every memory side-effect can be classified

ghost-type-specifier ::= *C-type-specifier*
| *logic-type-name*

declaration ::= *C-declaration*
| `/*@ ghost` *ghost-declaration* `*/`

direct-declarator ::= *C-direct-declarator*
| *direct-declarator* `(` *parameter-type-list*$^{?}$ `)`
`/*@ ghost (` *parameter-list* `) */` declare fun with ghost args

postfix-expression ::= *C-postfix-expression*
| *postfix-expression* `(` *argument-expression-list*$^{?}$ `)`
`/*@ ghost (` *argument-expression-list* `) */` call fun with ghost args

statement ::= *C-statement* | *statements-ghost C-statement*

statements-ghost ::= `/*@ ghost` *ghost-statement*$^{+}$ `*/`

ghost-selection-statement ::= *C-selection-statement*
| `if (` *expression* `)` *statement*
`/*@ ghost else` *C-statement*$^{+}$ `*/`

Figure 2.17: Grammar for ghost statements

as ghost or non-ghost. Then, the semantics is that memory side-effects of ghost code must always be in the ghost heap or the ghost stack.

Notice that this semantics is not statically decidable. It is left to tools to provide approximations, correct in the sense that any code statically detected as ghost must be semantically ghost.

Example 2.36 *The following example shows some invalid assignments of ghost pointers:*

```
void f(int x, int *q) {
  //@ ghost int *p = q;
  //@ ghost *p = 0;
  // above assignment is wrong: it modifies *q which lies
  // in regular memory heap

  //@ ghost p = &x;
  //@ ghost *p = 0;
  // above assignment is wrong: it modifies x which lies
  // in regular memory stack

}
```

Example 2.37 *The following example shows some invalid ghost statements:*

```
int f (int x, int y) {
  //@ ghost int z = x + y;
```

declaration ::= `//@ volatile` *tset* `reads` *id* `writes` *id*

Figure 2.18: Grammar for volatile constructs

```
  switch (x) {
  case 0: return y;
  //@ ghost 1: z=y;
  // above statement is correct.
  //@ ghost 2: { z++; break; }
  // invalid, would bypass the non−ghost default
  default: y++;
  }
  return y;
}

int g(int x) {
  //@ ghost int z = x;
  if (x > 0) { return x; }
  //@ ghost else { z++; return x; }
  // invalid, would bypass the non−ghost return
  return x+1;
}
```

Differences between model variables and ghost variables A ghost variable is an additional specification variable which is assigned in ghost code like any C variable. On the other hand, a model variable cannot be assigned, but one can state it is modified and can express properties about the new value, in a non-deterministic way, using logic assertions and invariants.

In other words, one can say that the specifications stating a ghost variable modification are executable.

Example 2.38 *The example 2.35 can also be specified with a ghost variable instead of a model variable, by adding before the return :* `//@ ghost forbidden = union(single(\result),forbidden;`

2.11.1 Volatile variables

EXPERIMENTAL

Volatile variables can not be used in logic terms, since reading such a variable may have a side effect, in particular two successive reads may return different values.

Specifying properties of a volatile variable may be done via a specific construct to attach two ghost functions to it. This construct, described by the grammar of Figure 2.18, has the following shape:

```
volatile τ x;
//@ volatile x reads f writes g;
```

where `f` and `g` are ghost functions with the following prototypes:

```
τ f(volatile τ* p);
τ g(volatile τ* p, τ v);
```

This must be understood as a special construct to instrument the C code, where each access to the variable `x` is replaced by a call to `f(&x)`, and each assignment to `x` of a value `v` is replaced by `g(&x,v)`.

Example 2.39 *The following code is instrumented in order to inject fixed values at each read of variable* x*, and collect written values.*

```
volatile int x;

/*@ ghost //@ requires p ≡ &x;
  @ int reads_x(volatile int *p) {
  @   static int injector_x[] = { 1, 2, 3 };
  @   static int injector_count = 0;
  @   if (p ≡ &x)
  @     return injector_x[injector_count++];
  @   else
  @     return 0; // should not happen
  @ }
  @*/

//@ ghost int collector_x[3];
//@ ghost int collector_count = 0;

/*@ ghost //@ requires p ≡ &x;
  @ int writes_x(volatile int *p, int v) {
  @   if (p ≡ &x)
  @     return collector_x[collector_count++] = v;
  @   else
  @    return 0; // should not happen
  @ }
  @*/

//@ volatile x reads reads_x writes writes_x;

/*@ ensures collector_count ≡ 3 ∧ collector_x[2] ≡ 2;
  @ ensures \result ≡ 6;
  @*/
int main () {
  int i, sum = 0;
  for (i=0 ; i < 3; i++) {
    sum += x;
    x = i;
  }
  return sum;
}
```

Chapter 3

Libraries

This chapter is devoted to librairies of specification, built upon the ACSL specification language.

Section 3.1 describes additional predicates propose by the Jessie plugin of Frama-C, to propose a slightly higher level of annotation.

3.1 Jessie library: logical adressing of memory blocks

The Jessie library is a collection of logic specifications whose semantics is well-defined only on source codes free from architecture-dependent features. In particular it is currently incompatible with pointer casts or unions (although there is ongoing work to support some of them [18]). As a consequence, in this particular setting, a valid pointer of some type $\tau*$ necessarily points to a memory block which contains values of type τ.

3.1.1 Abstract level of pointer validity

The Jessie plugin currently assumes the input source code free from architecture-dependent features. In particular it currently completely disallows pointer casts or unions (although there is ongoing work to support some of them [18]). As a consequence a valid pointer of some type $\tau*$ necessarily points to a memory block which contains values of type τ. To model that, the jessie library introduce new logic functions:

```
integer \offset_min(‘a *p);
integer \offset_max(‘a *p);
```

- `\offset_min`(p) is the minimum integer i such that $(p+i)$ is a valid pointer.
- `\offset_max`(p): the maximum integer i such that $(p+i)$ is a valid pointer

The following properties hold:

$$\begin{aligned} \texttt{\textbackslash offset\_min}(p+i) &= \texttt{\textbackslash offset\_min}(p) - i \\ \texttt{\textbackslash offset\_max}(p+i) &= \texttt{\textbackslash offset\_max}(p) - i \end{aligned}$$

It also introduce syntactic sugar:

$$\texttt{\textbackslash valid\_range}(p,i,j) \;:=\; \texttt{\textbackslash offset\_min}(p) <= i \wedge \texttt{\textbackslash offset\_max}(p) >= j$$

and the ACSL built-in predicate `\valid`(p) is now equivalent to `\valid_range`$(p,0,0)$.

3.1.2 Strings

EXPERIMENTAL
The predicate

$$\texttt{integer \strlen}(char * p)$$

denotes the length of a 0-terminated C string. It is total function, whose value is non-negative if and only if the pointer in argument is really a string.

Example 3.1 *Here is a contract for the* `strcpy` *function:*

```
/*@ // src and dest cannot overlap
  @ requires \base_addr(src) ≢ \base_addr(dest);
  @ // src is a valid C string
  @ requires \strlen(src) ≥ 0 ;
  @ // dest is large enough to store a copy of src up to the 0
  @ requires \valid_range(dest,0,\strlen(src));
  @ ensures
  @   ∀ integer k; 0 ≤ k ≤ \strlen(src) ⟹ dest[k] ≡ src[k]
  @*/
void str_cpy(char *dest, const char *src);
```

3.1.3 Field structures

"offsetof" "(" ... ")" ; EXPERIMENTAL
"alignof" "(" C-type-expr ")" ; EXPERIMENTAL

3.2 Memory leaks

EXPERIMENTAL
Verification of absence of memory leak is outside the scope of the specification language. On the other hand, various models could be set up, using for example ghost variables.

3.3 Libraries of logic specifications

A standard library is provided, in the spirit of the List module of Section 2.6.6

3.3.1 Real numbers

A library of general purpose functions and predicate over real numbers, floats and doubles.
Includes

- abs, exp, power, log, sin, cos, atan, etc. over reals
- isFinite predicate over floats and doubles (means not NaN nor infinity)
- rounding reals to floats or doubles with specific rounding modes.

3.3.2 Finite lists

- pure functions nil, cons, append, fold, etc.
- Path, Reachable, isFiniteList, isCyclic, etc. on C linked-lists.

3.3.3 Sets and Maps

Finite sets, finite maps, in ZB-style.

Chapter 4

Conclusion

This document presents a Behavioral Interface Specification Language for ANSI C source code. It provides a common basis that could be shared among several tools.

The specification language described here is intended to evolve in the future, and remain open to additional constructions.

One possible interesting extension is regarding “temporal” properties in a large sense, such as liveness properties, which sometimes can be simulated by regular specifications with ghost variables [9], or properties on evolution of data over the time, such as the history constraints of JML, or in the Lustre assertion language.

Appendix A

Appendices

A.1 Glossary

pure expressions In ACSL setting, a *pure* expression is a C expression which contains no assignments, no incrementation operator ++ or −−, no function call, and no access to a volatile object. The set of pure expression is a subset of the set of C expressions without side effect (C standard [11, 10], §5.1.2.3, alinea 2).

left-values A *left-value* (*lvalue* for short) is an expression which denotes some place in the memory during program execution, either on the stack, on the heap, or in the static data segment. It can be either a variable identifier or an expression of the form $*e$, $e[e]$, $e\,.\,id$ or $e{-}{>}id$, where e is any expression and id a field name. See C standard, §6.3.2.1 for a more detailed description of lvalues.

A *modifiable lvalue* is an lvalue allowed in the left part of an assignment. In essence, all lvalues are modifiable except variables declared as `const` or of some array type with explicit length.

pre-state and post-state For a given function call, the *pre-state* denotes the program state at the beginning of the call, including the current values for the function parameters. The *post-state* denotes the program state at the return of the call.

function behavior A *function behavior* (*behavior* for short) is a set of properties relating the pre-state and the post-state for a possibly restricted set of pre-states (behavior *assumptions*).

function contract A *function contract* (*contract* for short) forms a specification of a function, consisting of the combination of a precondition (a requirement on the pre-state for any caller to that function), a collection of behaviors, and possibly a measure in case of a recursive function.

A.2 Comparison with JML

Although we took our inspiration in the Java Modeling Language (aka JML [13]), our Behavioral Interface Specification Language (BISL for short) ACSL is notably different from JML in crucial ways. ACSL and JML differ mostly in two aspects:

- ACSL is a BISL for C, a low-level structured language, while JML is a BISL for Java, an object-oriented inheritance-based language. Not only the language features are not the same but the programming styles and idioms are very different, which makes for different ways of specifying behaviors. In particular, C has no inheritance nor exceptions, and no language support for the simplest properties on memory (e.g., the size of an allocated memory block).

- JML relies on runtime assertion checking (RAC) when typing, static analysis and automatic deductive verification fail. The example of CCured [19, 4], that adds strong typing to C by relying on RAC too, shows it is not possible to do it in a modular way. Indeed, it is necessary to modify the layout of C data structures for RAC, which is not modular. The follow-up project Deputy [5] thus reduces the checking power of annotations in order to preserve modularity. On the contrary, we choose not to restrein the power of annotations (e.g., all first order logic formulas are allowed). To that end, we rely on manual deductive verification using an interactive theorem prover (e.g., Coq) when every other technique failed.

In the following, we further describe these differences.

A.2.1 Low-level language vs. inheritance-based one

No inherited specifications

JML has a core notion of inheritance of specifications, that duplicates in specifications the inheritance feature of Java. Inheritance combined with visibility and modularity account for a number of complex features in JML (e.g., `spec_public` modifier, data groups, represents clauses, etc), that are necessary to express the desired inheritance-related specifications while respecting visibility and modularity. Since C has no inheritance, these intricacies are avoided in ACSL.

Error handling without exceptions

The usual way of signaling errors in Java is through exceptions. Therefore, JML specifications are tailored to express exceptional postconditions, depending on the exception raised. Since C has no exceptions, ACSL does not use exceptional specifications. Instead, C programmers are used to signal errors by returning special values, like mandated in various ways in the C standard.

Example A.1 *In §7.12.1 of the standard, it is said that functions in <math.h> signal errors as follows: "On a domain error, [...] the integer expression errno acquires the value EDOM."*

Example A.2 *In §7.19.5.1 of the standard, it is said that function fclose signals errors as follows: "The fclose function returns [...] EOF if any errors were detected."*

Example A.3 *In §7.19.6.1 of the standard, it is said that function fprintf signals errors as follows: "The fprintf function returns [...] a negative value if an output or encoding error occured."*

Example A.4 *In §7.20.3 of the standard, it is said that memory management functions signal errors as follows: "If the space cannot be allocated, a null pointer is returned."*

As shown by these few examples, there is no unique way to signal errors in the C standard library, not mentioning user-defined functions. But since errors are signaled by returning special values, it is sufficient to write an appropriate postcondition:

`/*@ ensures \result ==` *error_value* `||` *normal_postcondition*`; */`

C contracts are not Java ones

In Java, the precondition of the following function that nullifies an array of characters is always true. Even if there was a precondition on the length of array `a`, it could easily be expressed using the Java expression `a.length` that gives the dynamic length of array `a`.

```
public static void Java_nullify(char[] a) {
  if (a == null) return;
  for (int i = 0; i < a.length; ++i) {
    a[i] = 0;
  }
}
```

On the contrary, the precondition of the same function in C, whose definition follows, is more involved. One must remark that the C programmer has to add an extra argument for the size of the array, or rather a lower bound on this array size.

```
void C_nullify(char* a, unsigned int n) {
  int i;
  if (n == 0) return;
  for (i = 0; i < n; ++i) {
    a[i] = 0;
  }
}
```

A correct precondition for this function is the following:

```
/*@ requires \valid(a + 0..(n - 1)); */
```

It uses predicate `\valid` defined in Section 2.7.1, with the meaning that $\texttt{\valid}(a + 0..(-1))$ is always true. When `n` is null, `a` does not need to be valid at all, and when `n` is strictly positive, `a` must point to an array of size at least `n`. To make it more obvious, the C programmer adopted a defensive programming style, which returns immediately when `n` is null. We can duplicate this in the specification:

```
/*@ requires n == 0 || \valid(a + 0..(n - 1)); */
```

Usually, many memory requirements are only necessary for some paths through the function, which correspond to some particular behaviors, selected according to some tests performed along the corresponding paths. Since C has no memory primitives, these tests involve other variables that the C programmer added to track additional information, like `n` in our example.

To make it easier, it is possible in ACSL to distinguish between the `assume` part of a behavior, that specifies the tests that need to succeed for this behavior to apply, and the `requires` part that specifies the additional preconditions that must be true when a behavior applies. The specification for our example can then be translated into:

```
/*@ behavior n_is_null:
  @   assumes n == 0;
  @ behavior n_is_not_null:
  @   assumes n != 0;
  @   requires \valid(a + 0..(n - 1));
  @*/
```

This is equivalent to the previous requirement, except here behaviors can be completed with post-conditions that belong to one behavior only. Contrary to JML, the set of behaviors for a function do not necessarily cover all cases of use for this function, as mentioned in Section 2.3.3. This allows for partial specifications, whereas JML behaviors cannot offer such flexibility.

ACSL contracts vs. JML ones

To fully understand the difference between specifications in ACSL and JML, we detail in the following the requirement on the pre-state and the guarantee on the post-state, given behaviors in JML and ACSL.

A JML contract is either a lightweight or an heavyweight one. For the purpose of our comparison, it is sufficient to know that a lightweight contract has `requires` and `ensures` clauses all at the same level, while an heavyweight contract has multiple behaviors, each consisting of `requires` and `ensures` clauses. Although it is not possible in JML to mix both styles, we can define here what it would mean to have both, by conjoining the conditions on the pre- and the post-state. Here is an hypothetical JML contract mixing lightweight and heavyweight styles:

```
/*@ requires P1;
  @ requires P2;
  @ ensures Q1;
  @ ensures Q2;
  @ behavior x1:
  @   requires R1;
  @   ensures E1;
  @ behavior x2:
  @   requires R2;
  @   ensures E2;
  @*/
```

It assumes from the pre-state the condition:

$$P_1 \ \&\& \ P_2 \ \&\& \ (R_1 \ || \ R_2)$$

and guarantees that the following condition holds in post-state:

$$Q_1 \ \&\& \ Q_2 \ \&\& \ (\backslash\mathtt{old}(R_1) ==> E_1) \ \&\& \ (\backslash\mathtt{old}(R_2) ==> E_2)$$

Here is now an ACSL specification:

```
/*@ requires P1;
  @ requires P2;
  @ ensures Q1;
  @ ensures Q2;
  @ behavior x1:
  @   assumes A1;
  @   requires R1;
  @   ensures E1;
  @ behavior x2:
  @   assumes A2;
  @   requires R2;
  @   ensures E2;
  @*/
```

Syntactically, the only difference with the JML specification is the addition of the `assumes` clauses. Its translation to assume-guarantee is however quite different. It assumes from the pre-state the condition:

$$P_1 \ \&\& \ P_2 \ \&\& \ (A_1 ==> R_1) \ \&\& \ (A_2 ==> R_2)$$

and guarantees that the following condition holds in the post-state:

$$Q_1 \ \&\& \ Q_2 \ \&\& \ (\backslash\mathtt{old}(A_1) ==> E_1) \ \&\& \ (\backslash\mathtt{old}(A_2) ==> E_2)$$

ACSL `assigns` clause

Beware that in `assigns` clause, the expressions given refer to the post-state, whereas in JML, they refer to the pre-state. The JML semantics can be obtained using `\old` construct.

A.2.2 Deductive verification vs. RAC

Sugar-free behaviors

As explained in details in [20], JML heavyweight behaviors can be viewed as syntactic sugar (however complex it is) that can be translated automatically into more basic contracts consisting mostly of pre- and postconditions and frame conditions. This allows complex nesting of behaviors from the user point of view, while tools only have to deal with basic contracts. In particular, the major tools on JML use this desugaring process, like the Common JML tools to do assertion checking, unit testing, etc. (seen in [16]) and the tool ESC/Java2 for automatic deductive verification of JML specifications (seen in [3]).

One issue with such a desugaring approach is the complexity of the transformations involved, e.g., see the desugaring of assignable clauses between multiple *spec-cases* in JML [20]. Another issue is precisely that tools only see one global contract, instead of multiple independent behaviors, that could be analyzed in more details separately.

Instead, we favor the view that a function implements multiple behaviors, that can be analyzed separately if a tool feels like it. Therefore, we do not intend to provide a desugaring process.

Axiomatized functions in specifications

JML only allows pure Java methods to be called in specifications [14]. This is certainly understandable when relying on RAC: methods called should be defined so that the runtime can call them, and they should not have side-effects in order not to pollute the program they are supposed to annotate.

In our setting, it is desirable to allow calls to logical functions in specifications. These functions may be defined, like program ones, but they may also be only declared (with a suitable declaration of `reads` clause) and defined through an axiomatization. This makes for richer specifications that may be useful either in automatic or in manual deductive verification.

A.2.3 Syntactic differences

The following table summarizes the difference between JML keywords and ACSL ones, when the intent is the same, although minor differences might exist.

JML	ACSL
modifiable,assignable	assigns
measured_by	decreases
loop_invariant	loop invariant
decreases	loop variant
(`\forall` $\tau\ x; P; Q$)	(`\forall` $\tau\ x; P$`==>`Q)
(`\exists` $\tau\ x; P; Q$)	(`\exists` $\tau\ x; P$`&&`Q)
(`\max` $\tau\ x; a$`<=`x`<=`$$b; f$)	`\max`(a, b, `\lambda` $\tau\ x; f$)

A.3 Typing rules

Disclaimer: this section is unfinished, it is left here just to give an idea of what it will look like in the final document.

A.3.1 Rules for terms

Integer promotion:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \texttt{integer}}$$

if τ is any C integer type `char`, `short`, `int`, or `long`, whatever attribute they have, in particular signed or unsigned

Variables:

$$\frac{}{\Gamma \vdash id : \tau} \text{ if } id : \tau \in \Gamma$$

Unary integer operations:

$$\frac{\Gamma \vdash t : \texttt{integer}}{\Gamma \vdash op\, t : \texttt{integer}} \text{ if } op \in \{+, -, \tilde{}\}$$

Boolean negation:

$$\frac{\Gamma \vdash t : \texttt{boolean}}{\Gamma \vdash ! \, t : \texttt{boolean}}$$

Pointer dereferencing:

$$\frac{\Gamma \vdash t : \tau *}{\Gamma \vdash *t : \tau}$$

Address operator:

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \&t : \tau *}$$

Binary

$$\frac{\Gamma \vdash t_1 : \texttt{integer} \qquad \Gamma \vdash t_2 : \texttt{integer}}{\Gamma \vdash t_1\ op\ t_2 : \texttt{integer}} \text{ if } op \in \{+, -, *, /, \%\}$$

$$\frac{\Gamma \vdash t_1 : \texttt{real} \qquad \Gamma \vdash t_2 : \texttt{real}}{\Gamma \vdash t_1\ op\ t_2 : \texttt{real}} \text{ if } op \in \{+, -, *, /\}$$

$$\frac{\Gamma \vdash t_1 : \texttt{integer} \qquad \Gamma \vdash t_2 : \texttt{integer}}{\Gamma \vdash t_1\ op\ t_2 : \texttt{boolean}} \text{ if } op \in \{==, ! =, <=, <, >=, >\}$$

$$\frac{\Gamma \vdash t_1 : \texttt{real} \qquad \Gamma \vdash t_2 : \texttt{real}}{\Gamma \vdash t_1\ op\ t_2 : \texttt{boolean}} \text{ if } op \in \{==, ! =, <=, <, >=, >\}$$

$$\frac{\Gamma \vdash t_1 : \tau * \qquad \Gamma \vdash t_2 : \tau *}{\Gamma \vdash t_1\ op\ t_2 : \texttt{boolean}} \text{ if } op \in \{==, ! =, <=, <, >=, >\}$$

(to be continued)

A.3.2 Typing rules for locations

We consider the typing judgement $\Gamma, \texttt{\textbackslash lambda} \vdash s : \tau, b$ meaning that s is a set of terms of type tau, which is moreover a set of locations if the boolean b is true. Γ is the C environment and `\lambda` is the logic environment.

Rules:

$$\frac{}{\Gamma \vdash id : \tau, true} \text{ if } id : \tau in \Gamma$$

$$\frac{\Gamma \vdash s : \tau *, b}{\vdash *s : \tau, true}$$

$$\frac{e : structS * \qquad e_2 : tset\tau}{\vdash s- > id : \tau}$$

$$\frac{b \cup \texttt{\textbackslash lambda} \vdash e : tset\tau}{\vdash \{e \mid b; P\} : tset\tau}$$

$$\frac{\Gamma \vdash e_1 : \tau, b \quad e_2 : \tau, b}{\vdash e_1, e_2 : \tau, b}$$

A.4 Illustrative example

This is an attempt to define an example for ACSL, much as the Purse example in JML description papers. It is a memory allocator, whose main functions are `memory_alloc` and `memory_free`, to respectively allocate and deallocate memory. The goal is to exercise as much as possible of ACSL.

```
#include <stdlib.h>

#define DEFAULT_BLOCK_SIZE 1000

typedef enum _bool { false = 0, true = 1 } bool;

/*@ predicate finite_list<A>((A* -> A*) next_elem, A* ptr) =
  @   ptr ≡ \null ∨
  @   (\valid(ptr) ∧ finite_list(next_elem,next_elem(ptr))) ;
  @
  @ logic integer list_length<A>((A* -> A*) next_elem, A* ptr) =
  @   (ptr ≡ \null) ? 0 :
  @   1 + list_length(next_elem,next_elem(ptr)) ;
  @
  @
  @ predicate lower_length<A>((A* -> A*) next_elem,
  @                           A* ptr1, A* ptr2) =
  @   finite_list(next_elem, ptr1) ∧ finite_list(next_elem, ptr2)
  @   ∧ list_length(next_elem, ptr1) < list_length(next_elem, ptr2) ;
  @*/

// forward reference
struct _memory_slice;

/* A memory block holds a pointer to a raw block of memory allocated by
 * calling malloc. It is sliced into chunks, which are maintained by
 * the slice structure. It maintains additional information such as
 * the size of the memory block, the number of bytes used and the next
 * index at which to put a chunk.
 */
typedef struct _memory_block {
  //@ ghost bool         packed;
    // ghost field packed is meant to be used as a guard that tells when
    // the invariant of a structure of type memory_block holds
  unsigned int           size;
    // size of the array data
  unsigned int           next;
    // next index in data at which to put a chunk
  unsigned int           used;
    // how many bytes are used in data, not necessarily contiguous ones
  char*                  data;
    // raw memory block allocated by malloc
  struct _memory_slice* slice;
    // structure that describes the slicing of a block into chunks
```

```
} memory_block;

/*@ type invariant inv_memory_block(memory_block mb) =
  @   mb.packed ⟹
  @     (0 < mb.size ∧ mb.used ≤ mb.next ≤ mb.size
  @     ∧ \offset (mb.data) ≡ 0
  @     ∧ \block_length(mb.data) ≡ mb.size) ;
  @
  @ predicate valid_memory_block(memory_block* mb) =
  @   \valid(mb) ∧ mb->packed ;
  @*/

/* A memory chunk holds a pointer data to some part of a memory block
 * block. It maintains the offset at which it points in the block, as well
 * as the size of the block it is allowed to access. A field free tells
 * whether the chunk is used or not.
 */
typedef struct _memory_chunk {
  //@ ghost bool packed;
    // ghost field packed is meant to be used as a guard that tells when
    // the invariant of a structure of type memory_chunk holds
  unsigned int   offset;
    // offset at which data points into [block->data]
  unsigned int   size;
    // size of the chunk
  bool           free;
    // true if the chunk is not used, false otherwise
  memory_block*  block;
    // block of memory into which the chunk points
  char*          data;
    // shortcut for [block->data + offset]
} memory_chunk;

/*@ type invariant inv_memory_chunk(memory_chunk mc) =
  @   mc.packed ⟹
  @     (0 < mc.size ∧ valid_memory_block(mc.block)
  @     ∧ mc.offset + mc.size ≤ mc.block->next) ;
  @
  @ predicate valid_memory_chunk(memory_chunk* mc, int s) =
  @   \valid(mc) ∧ mc->packed ∧ mc->size ≡ s ;
  @
  @ predicate used_memory_chunk(memory_chunk mc) =
  @   mc.free ≡ false ;
  @
  @ predicate freed_memory_chunk(memory_chunk mc) =
  @   mc.free ≡ true ;
  @*/

/* A memory chunk list links memory chunks in the same memory block.
 * Newly allocated chunks are put first, so that the offset of chunks
```

```
 * decreases when following the next pointer. Allocated chunks should
 * fill the memory block up to its own next index.
 */
typedef struct _memory_chunk_list {
  memory_chunk*              chunk;
    // current list element
  struct _memory_chunk_list* next;
    // tail of the list
} memory_chunk_list;

/*@ logic memory_chunk_list* next_chunk(memory_chunk_list* ptr) = ptr->next ;
  @
  @ predicate valid_memory_chunk_list
  @                   (memory_chunk_list* mcl, memory_block* mb) =
  @   \valid(mcl) ∧ valid_memory_chunk(mcl->chunk,mcl->chunk->size)
  @   ∧ mcl->chunk->block ≡ mb
  @   ∧ (mcl->next ≡ \null ∨
  @        valid_memory_chunk_list(mcl->next, mb))
  @   ∧ mcl->offset ≡ mcl->chunk->offset
  @   ∧ (
  @         // it is the last chunk in the list
  @         (mcl->next ≡ \null ∧ mcl->chunk->offset ≡ 0)
  @      ∨
  @         // it is a chunk in the middle of the list
  @         (mcl->next ≢ \null
  @         ∧ mcl->next->chunk->offset + mcl->next->chunk->size
  @            ≡ mcl->chunk->offset)
  @      )
  @   ∧ finite_list(next_chunk, mcl) ;
  @
  @ predicate valid_complete_chunk_list
  @                   (memory_chunk_list* mcl, memory_block* mb) =
  @   valid_memory_chunk_list(mcl,mb)
  @   ∧ mcl->next->chunk->offset +
  @      mcl->next->chunk->size ≡ mb->next ;
  @
  @ predicate chunk_lower_length(memory_chunk_list* ptr1,
  @                              memory_chunk_list* ptr2) =
  @   lower_length(next_chunk, ptr1, ptr2) ;
  @*/

/* A memory slice holds together a memory block block and a list of chunks
 * chunks on this memory block.
 */
typedef struct _memory_slice {
  //@ ghost bool     packed;
    // ghost field packed is meant to be used as a guard that tells when
    // the invariant of a structure of type memory_slice holds
  memory_block*      block;
  memory_chunk_list* chunks;
```

```
} memory_slice;

/*@ type invariant inv_memory_slice(memory_slice* ms) =
  @   ms.packed ⟹
  @     (valid_memory_block(ms->block) ∧ ms->block->slice ≡ ms
  @     ∧ (ms->chunks ≡ \null
  @         ∨ valid_complete_chunk_list(ms->chunks, ms->block))) ;
  @
  @ predicate valid_memory_slice(memory_slice* ms) =
  @   \valid(ms) ∧ ms->packed ;
  @*/

/* A memory slice list links memory slices, to form a memory pool.
 */
typedef struct _memory_slice_list {
  //@ ghost bool     packed;
    // ghost field packed is meant to be used as a guard that tells when
    // the invariant of a structure of type memory_slice_list holds
  memory_slice*              slice;
    // current list element
  struct _memory_slice_list* next;
    // tail of the list
} memory_slice_list;

/*@ logic memory_slice_list* next_slice(memory_slice_list* ptr) = ptr->next ;
  @
  @ type invariant inv_memory_slice_list(memory_slice_list* msl) =
  @   msl.packed ⟹
  @     (valid_memory_slice(msl->slice)
  @     ∧ (msl->next ≡ \null ∨
  @         valid_memory_slice_list(msl->next))
  @     ∧ finite_list(next_slice, msl)) ;
  @
  @ predicate valid_memory_slice_list(memory_slice_list* msl) =
  @   \valid(msl) ∧ msl->packed ;
  @
  @ predicate slice_lower_length(memory_slice_list* ptr1,
  @                              memory_slice_list* ptr2) =
  @      lower_length(next_slice, ptr1, ptr2)
  @ } */

typedef memory_slice_list* memory_pool;

/*@ type invariant valid_memory_pool(memory_pool *mp) =
  @   \valid(mp) ∧ valid_memory_slice_list(*mp) ;
  @*/

/*@ behavior zero_size:
  @   assumes s ≡ 0;
  @   assigns \nothing;
```

```
  @   ensures \result ≡ 0;
  @
  @ behavior positive_size:
  @   assumes s > 0;
  @   requires valid_memory_pool(arena);
  @   ensures \result ≡ 0
  @     ∨ (valid_memory_chunk(\result,s) ∧
  @         used_memory_chunk(*\result));
  @ */
memory_chunk* memory_alloc(memory_pool* arena, unsigned int s) {
  memory_slice_list *msl = *arena;
  memory_chunk_list *mcl;
  memory_slice *ms;
  memory_block *mb;
  memory_chunk *mc;
  unsigned int mb_size;
  //@ ghost unsigned int mcl_offset;
  char *mb_data;
  // guard condition
  if (s ≡ 0) return 0;
  // iterate through memory blocks (or slices)
  /*@
    @ loop invariant valid_memory_slice_list(msl);
    @ loop variant msl for slice_lower_length;
    @ */
  while (msl ≢ 0) {
    ms = msl->slice;
    mb = ms->block;
    mcl = ms->chunks;
    // does mb contain enough free space?
    if (s ≤ mb->size - mb->next) {
      //@ ghost ms->ghost = false;    // unpack the slice
      // allocate a new chunk
      mc = (memory_chunk*)malloc(sizeof(memory_chunk));
      if (mc ≡ 0) return 0;
      mc->offset = mb->next;
      mc->size = s;
      mc->free = false;
      mc->block = mb;
      //@ ghost mc->ghost = true;    // pack the chunk
      // update block accordingly
      //@ ghost mb->ghost = false;    // unpack the block
      mb->next += s;
      mb->used += s;
      //@ ghost mb->ghost = true;    // pack the block
      // add the new chunk to the list
      mcl = (memory_chunk_list*)malloc(sizeof(memory_chunk_list));
      if (mcl ≡ 0) return 0;
      mcl->chunk = mc;
      mcl->next = ms->chunks;
```

```
      ms->chunks = mcl;
      //@ ghost ms->ghost = true;    // pack the slice
      return mc;
    }
    // iterate through memory chunks
    /*@
      @ loop invariant valid_memory_chunk_list(mcl,mb);
      @ loop variant mcl for chunk_lower_length;
      @ */
    while (mcl ≢ 0) {
      mc = mcl->chunk;
      // is mc free and large enough?
      if (mc->free ∧ s ≤ mc->size) {
        mc->free = false;
        mb->used += mc->size;
        return mc;
      }
      // try next chunk
      mcl = mcl->next;
    }
    msl = msl->next;
  }
  // allocate a new block
  mb_size = (DEFAULT_BLOCK_SIZE < s) ? s : DEFAULT_BLOCK_SIZE;
  mb_data = (char*)malloc(mb_size);
  if (mb_data ≡ 0) return 0;
  mb = (memory_block*)malloc(sizeof(memory_block));
  if (mb ≡ 0) return 0;
  mb->size = mb_size;
  mb->next = s;
  mb->used = s;
  mb->data = mb_data;
  //@ ghost mb->ghost = true;    // pack the block
  // allocate a new chunk
  mc = (memory_chunk*)malloc(sizeof(memory_chunk));
  if (mc ≡ 0) return 0;
  mc->offset = 0;
  mc->size = s;
  mc->free = false;
  mc->block = mb;
  //@ ghost mc->ghost = true;    // pack the chunk
  // allocate a new chunk list
  mcl = (memory_chunk_list*)malloc(sizeof(memory_chunk_list));
  if (mcl ≡ 0) return 0;
  //@ ghost mcl->offset = 0;
  mcl->chunk = mc;
  mcl->next = 0;
  // allocate a new slice
  ms = (memory_slice*)malloc(sizeof(memory_slice));
  if (ms ≡ 0) return 0;
```

```
  ms->block = mb;
  ms->chunks = mcl;
  //@ ghost ms->ghost = true;     // pack the slice
  // update the block accordingly
  mb->slice = ms;
  // add the new slice to the list
  msl = (memory_slice_list*)malloc(sizeof(memory_slice_list));
  if (msl ≡ 0) return 0;
  msl->slice = ms;
  msl->next = *arena;
  //@ ghost msl->ghost = true;     // pack the slice list
  *arena = msl;
  return mc;
}

/*@ behavior null_chunk:
  @   assumes chunk ≡ \null;
  @   assigns \nothing;
  @
  @ behavior valid_chunk:
  @   assumes chunk ≢ \null;
  @   requires valid_memory_pool(arena);
  @   requires valid_memory_chunk(chunk,chunk->size);
  @   requires used_memory_chunk(chunk);
  @   ensures
  @       // if it is not the last chunk in the block, mark it as free
  @       (valid_memory_chunk(chunk,chunk->size)
  @       ∧ freed_memory_chunk(chunk))
  @     ∨
  @       // if it is the last chunk in the block, deallocate the block
  @       ! \valid(chunk);
  @ */
void memory_free(memory_pool* arena, memory_chunk* chunk) {
  memory_slice_list *msl = *arena;
  memory_block *mb = chunk->block;
  memory_slice *ms = mb->slice;
  memory_chunk_list *mcl;
  memory_chunk *mc;
  // is it the last chunk in use in the block?
  if (mb->used ≡ chunk->size) {
    // remove the corresponding slice from the memory pool
    // case it is the first slice
    if (msl->slice ≡ ms) {
      *arena = msl->next;
      //@ ghost msl->ghost = false;     // unpack the slice list
      free(msl);
    }
    // case it is not the first slice
    while (msl ≢ 0) {
      if (msl->next ≢ 0 ∧ msl->next->slice ≡ ms) {
```

```
        memory_slice_list* msl_next = msl->next;
        msl->next = msl->next->next;
        // unpack the slice list
        //@ ghost msl_next->ghost = false;
        free(msl_next);
        break;
      }
      msl = msl->next;
    }
    //@ ghost ms->ghost = false;     // unpack the slice
    // deallocate all chunks in the block
    mcl = ms->chunks;
    // iterate through memory chunks
    /*@
      @ loop invariant valid_memory_chunk_list(mcl,mb);
      @ loop variant mcl for chunk_lower_length;
      @ */
    while (mcl ≠ 0) {
      memory_chunk_list *mcl_next = mcl->next;
      mc = mcl->chunk;
      //@ ghost mc->ghost = false;     // unpack the chunk
      free(mc);
      free(mcl);
      mcl = mcl_next;
    }
    mb->next = 0;
    mb->used = 0;
    // deallocate the memory block and its data
    //@ ghost mb->ghost = false;      // unpack the block
    free(mb->data);
    free(mb);
    // deallocate the corresponding slice
    free(ms);
    return;
  }
  // mark the chunk as freed
  chunk->free = true;
  // update the block accordingly
  mb->used -= chunk->size;
  return;
}
```

Bibliography

[1] Patrice Chalin. Reassessing JML's logical foundation. In *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05)*, Glasgow, Scotland, July 2005.

[2] Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*, pages 23–33, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[3] David R. Cok and Joseph R. Kiniry. ESC/Java2 implementation notes. Technical report, may 2007. http://secure.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/Escjava2-ImplementationNotes.pdf.

[4] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244, 2003.

[5] Jeremy Paul Condit, Matthew Thomas Harren, Zachary Ryan Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *ESOP '07: Proceedings of the 16th European Symposium on Programming*, Oct 2006.

[6] The Coq Proof Assistant. http://coq.inria.fr/.

[7] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, November 2004. Springer.

[8] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer.

[9] A. Giorgetti and J. Groslambert. JAG: JML Annotation Generation for verifying temporal properties. In *FASE'2006, Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*, pages 373–376, Vienna, Austria, March 2006. Springer.

[10] International Organization for Standardization (ISO). *The ANSI C standard (C99)*. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

[11] Brian Kernighan and Dennis Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988.

[12] Joseph Kiniry. ESC/Java2. http://kind.ucd.ie/products/opensource/ESCJava2/.

[13] Gary Leavens. Jml. http://www.cs.iastate.edu/~leavens/JML.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[15] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, 2007.

[16] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[17] Claude Marché. Towards modular algebraic specifications for pointer programs: a case study. In H. Comon-Lundh, C. Kirchner, and H. Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 235–258. Springer-Verlag, 2007.

[18] Yannick Moy. Union and cast in deductive verification. Technical Report ICIS-R07015, Radboud University Nijmegen, jul 2007. http://www.lri.fr/~moy/union_and_cast/union_and_cast.pdf.

[19] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[20] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03a, Iowa State University, 2000.

List of Figures

Index