ACSL: ANSI/ISO C Specification Language

Preliminary design (version 1.4, December 12, 2008)

Patrick Baudin[1], Jean-Christophe 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://frama-c.cea.fr`).

This is the version 1.4 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.

Acknowledgements

We gratefully thank all the 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 1

Introduction

This document is a reference manual for the ACSL implementation provided by the Frama-C framework [9]. ACSL is an acronym for "ANSI C Specification Language". This is a Behavioral Interface Specification Language (a.k.a. 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 [17]) 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 [13], 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 [12, 11].

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, Appendix A provides a few additional information. A detailed table of contents is given on page 2. A glossary is given in Appendix A.1.

Not all of the features mentioned in this document are currently implemented in the Frama-C kernel. Those who aren't yet are signaled as in the following line:

This feature is not currently supported by Frama-C [1]

As a summary, the features that are not currently implemented into Frama-C include in particular:

- some built-in predicates and logical functions;
- abrupt termination clauses in statement contracts (section 2.4.4);
- definition of logical types (section 2.6);
- specification modules (section 2.6.10);
- model variables and Model fields (section 2.44);

[1] Additional remarks on the feature may appear as footnote

- only basic support for ghost code is provided (section 2.12);
- verification of non interference of ghost code (p. 56);
- specification of volatile variables (section 2.12.1);

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 2.6.10) 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 2.11.
  - *type invariant* : this allows to declare both structure or union invariants, and invariants on type names introduced by `typedef`. See section 2.11.
  - *logic specifications* : definitions of logic functions or predicates, lemmas, axiomatizations by declaration of new logic types, logic functions, predicates with axioms they satisfy. 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 conditions must be checked (no goto going inside, no goto going outside). See Section 2.4.4.
  - *ghost code* : regular C code, only visible from the specifications, that is only allowed to modify ghost variables. See section 2.12. 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. For the sake of simplicity, we only describe annotations in the usual `/*@ ... */` style of comments. One-line annotations in `//@` comments are alike.

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.
- 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
`!=`	$\not\equiv$	0x2262
`==`	$\equiv$	0x2261
`==>`	$\Longrightarrow$	0x21D2
`<==>`	$\Longleftrightarrow$	0x21D4
`&&`	$\wedge$	0x2227
`||`	$\vee$	0x2228
`^^`	$\oplus$	0x22BB
`!`	$\neg$	0x00AC
`\forall`	$\forall$	0x2200
`\exists`	$\exists$	0x2203

- Comments can be put inside ACSL annotations. They use the C++ format, *i.e.* begin with `//` and extend to the end of current line.

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

```
bin-op   ::= + | - | * | / | %
           | == | != | <= | >= | > | <
           | && | || |                           boolean operations
           | & | | | --> | <--> | ^              bitwise operations

unary-op ::= + | -                               unary plus and minus
           | !                                   boolean negation
           | ~                                   bitwise complementation
           | *                                   pointer dereferencing
           | &                                   address-of operator

term     ::= \true | \false
           | integer                             integer constants
           | real                                real constants
           | id                                  variables
           | unary-op term
           | term bin-op term
           | term [ term ]                       array access
           | { term for [ term ] = term }        array functional modifier
           | term . id                           structure field access
           | { term for id = term }              structure field functional modifier
           | 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)
           | 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	::=	*binder* (, *binder*)*	
binder	::=	*type-expr variable-ident* (, *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* | `*` *variable-ident* | *variable-ident* `[]`	
	|	`(` *variable-ident* `)`	

Figure 2.2: Grammar of binders and type expressions

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:

Additional connectives C operators `&&` (UTF8: ∧), `||` (UTF8: ∨) and `!` (UTF8: ¬) are used as logical connectives. There are additional connectives `==>` (UTF8: ⟹) for implication, `<==>` (UTF8: ⟺) for equivalence and `^^` (UTF8: ⊕) for exclusive or. These logical connectives all have a bitwise counterpart, either C ones like `&`, `|`, `~` and `^`, or additional ones like bitwise implication `-->` and bitwise equivalence `<-->`.

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

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.

Functional modifier The composite element modifier is an additional operator in relation to the C structure field and array accessors. The expression `{` s `for` $id = v$ `}` denotes the same structure than s, except for the field id to be modified by v. The equivalent expression for an array is `{` t `for [` i `]` $= v$ `}` which returns the same array than t, except for the i^{th} element those value is updated to v. See Section 2.10 for an example of use of these operators.

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 obvious postcondition* `\result == a < b` *is not the right one because it is actually a shortcut for* `\result == a && a < b`.
- *adding parentheses results in a correct post-condition* `\result == (a < b)`. *Note however that there is an implicit conversion (see Sec. 2.2.3) from the* `int` *(the type of* `\result`*) to* `boolean` *(the type of* `(a<b)`*)*
- *an equivalent post-condition, which does not rely on implicit conversion, is* `(\result != 0) == (a<b)`. *Both pairs of parentheses are mandatory.*
- `\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.*
- `\result != 0 <==> a < b` *is acceptable because it is an equivalence between two predicates.*

2.2.1 Operators precedence

The precedence of C operators is conservatively extended with additional operators, as shown in 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 [23]. 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 [2, 3].)

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.

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

Figure 2.3: Operator precedence

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 used carefully, since it depends on the memory state where dereferencing is done (see Section 2.2.4 and Section 2.6.9).

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 "arithmetic 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 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 $2^{8 \times \mathtt{sizeof}(t)}$.

Example 2.3 `(unsigned char)1000` *is* 1000 mod 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. The `sizeof` operator may be used in the logic and is assumed to be consistent with the C code. For instance, it should be possible to verify the following code:

```
/*@ ensures \result ≤ sizeof(int); */
int f() { return sizeof(char); }
```

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 an integer into an 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 integer: any mathematical integer has a unique infinite 2-complement binary representation with infinitely many zeros (for non-negative numbers) or ones (for negative numbers) on the left. Then bitwise operations apply to this representation.

Example 2.6

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

Transcendental functions

Classical mathematical operations like exponential, sine, cosine, etc. are supposed to be available in some library (see Section 3.1.1).

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*

Unlike the promotion of C integer types to mathematical integers, there are special float values which do not naturally map to a real number, namely NaN, +Infinity and -Infinity. Since we want a logic with total functions, we consider that there are implicit promotion functions `real_of_float` and `real_of_double` whose results on the 3 values above is left unspecified.

Usual operators for comparison are interpreted as real operators too. In particular, equality operation $\equiv$ of float (or double) expressions means equality of the real numbers they represent respectively. Or equivalently, $x \equiv y$ for x, y two float variables means `real_of_float`$(x) \equiv$ `real_of_float`(y) with the mathematical equality of real numbers.

Special predicates are also available to express the comparison operators of float (resp. double) numbers as in C: `\eq_float`, `\gt_float`, `\ge_float`, `\le_float`, `\lt_float`, `\ne_float` (resp. for double).

Casts, infinity and NaNs

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

Conversion of real numbers to float or double values indeed depends on various possible rounding modes defined by the IEEE 754 standard [22, 24]. These modes are defined by a logic type (see section 2.6.8):

```
/*@ type rounding_mode =
  Nearest_even | To_zero | Round_up | Round_down | Nearest_away
/
```

Then rounding a real number can be done explicitly using functions

```
logic float \round_float(rounding_mode m, real x);
logic double \round_double(rounding_mode m, real x);
```

Cast operators `(float)` and `(double)` applied to a mathematical integer or real number x are equivalent to apply rounding functions above with the nearest-even rounding mode (which is the default rounding mode in C programs) If the source real number is too large, this may also result into one of the special values +infinity and -infinity.

Example 2.8 *We have* `(float)0.1` $\equiv 13421773 \times 2^{-27}$ *which is equal to* 0.10000000149006473916

Notice also that unlike for integers, suffixes f and l are meaningful, because they implicitly add a cast operator as above.

This semantics of casts ensures that the float result r of a C operation $e_1 op e_2$ on floats, if there is no overflow and if the default rounding mode is not changed in the program, has the same real value as the logic expression $(float)(e_1 op e_2)$. Notice that this is not true for the equality `\eq_float` of floats: `-0.0 + -0.0` in C is equal to the float number `-0.0`, which is not `\eq_float` to `0.0`, which is the value of the logic expression `(float)(-0.0 + -0.0)`.

Finally, two additional predicates are provided on float and double numbers,which check that their argument is NaN or a finite number respectively:

```
predicate \is_nan_float(float x);
predicate \is_nan_double(double x);
predicate \is_finite_float(float x);
predicate \is_finite_double(double x);
```

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 is 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

Figure 2.4 shows a grammar for function contracts. Non-terminal *location* is defined later in Section 2.3.4 (informally, *location* denotes a set of C l-values).

This section is organized as follows. First, the grammar for terms is extended with two new constructs. Then Section 2.3.2 introduces *simple contracts*. Finally, Section 2.3.3 defines more general contracts involving *named behaviors* The `decreases` and `terminates` clauses are presented later in Section 2.5.

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

Post-conditions usually require to refer to function results and values in the pre-state. Thus terms are extended with the following new constructs (shown in Figure 2.5).

- `\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 `ensures` clauses, since the other clauses already refer to the pre-state.

fun-contract	::=	*simple-behavior named-behavior** *decreases-clause*$^?$
simple-behavior	::=	*simple-clauses** *terminates-clause*$^?$ *simple-clauses**
simple-clauses	::=	*requires-clause* | *assigns-clause* | *ensures-clause*
named-behavior	::=	`behavior` *ident* `:` *behavior-body*
behavior-body	::=	(*assumes-clause* | *requires-clause*[a] | *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*)$^?$ `;`
terminates-clause	::=	`terminates` *pred* `;`

[a] not in a named behavior

Figure 2.4: Grammar of function contracts

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

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

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 $L_1 \cup L_2 \cup \ldots$ remain allocated and are left unchanged in the post-state. The set $L_1 \cup L_2 \cup \ldots$ itself is interpreted in the pre-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 && ...;
  @*/
```

If no clause `requires` is given, it defaults to `\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 the integer square root.*

```
/*@ 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 be called with a nonnegative argument, and returns a value satisfying the conjunction of the three `ensures` *clauses. Inside these* `ensures` *clauses, the use of the construct* `\old(x)` *is not necessary, even if the function modifies the formal operand* x*, because function calls modify a copy of the effective operands, and the effective operands remain unaltered. In fact,* x *denotes the effective operand of* `isqrt` *calls which has the same value interpreted in the pre-state than in the post-state.*

Example 2.10 *The following function is given a contract to specify that it increments the value pointed to 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 the* `\valid` *built-in predicate). It does not modify any memory location but the one pointed to by* `p`*. Finally, the* `ensures` *clause specifies that the value* `*p` *is incremented by one.*

2.3.3 Contracts with named behaviors

The general form of a function contract contains several named behaviors (restricted to two behaviors, in the following, 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 the call is performed 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 does not belong to the set L_i remains 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;
  @*/
```

Note that a simple contract such as

```
/*@ requires P; assigns L; ensures E; */
```

is actually equivalent to a single named behavior as follows:

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

Similarly, global assigns and ensures clauses are equivalent to a single named behavior. More precisely,

```
/*@ requires P;
  @ assigns L;
  @ ensures E;
  @ behavior b1:  ...
  @ behavior b2:  ...
  @ ⋮
  @*/
```

is equivalent to

```
/*@ requires P;
  @ behavior <any name>:
  @   assumes \true;
  @   assigns L;
  @   ensures E;
  @ behavior b1:  ...
  @ behavior b2:  ...
  @ ⋮
  @*/
```

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 requires array t *to be allocated at least from indices* 0 *to* $n-1$. *The two named behaviors correspond respectively to the successful behavior and the failing behavior.*

Since the function is performing a binary search, it requires the array t *to be 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 to 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.* it is not mandatory to provide a "complete" set of behaviors. If such a condition is seeked, it is possible to add the following clause to a contract:

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

It specifies that the set of behaviors $b_1, \ldots, b_n$ is complete *i.e.* that

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

holds, where R is the precondition of the contract. The simplified version of that clause

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

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

Similarly, it is not required that two distinct behaviors are disjoint. If desired, this can be specified with the following clause:

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

It means that the given behaviors are pairwise disjoint *i.e.* that, for all distinct i and j,

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

holds. The simplified version of that clause

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

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

2.3.4 Memory locations

There are several places where one needs to describe a set of memory locations: in `assigns` clauses of function contracts, or in `\separated` predicate (see Section 2.7.2). More generally, we introduce syntactic constructs to denote *sets of terms*. A *memory location* is then any set of terms denoting a set of l-values. Moreover, a location given as argument to an `assigns` clause must be a set of modifiable l-values, as described in Section A.1.

The grammar for sets of terms is given in Figure 2.6. The semantics is given below, where s denotes any *tset*.

- `\empty` denotes the empty set.
- a simple term denotes a singleton set.

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

- $s \rightarrow id$ 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 $x \in s$.
- $\&s$ 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. If $t_1 > t_2$, this the same as `\empty`
- `\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, that is the union of the sets denoted by s for each value b of binders satisfying predicate P (binders b are bound in both s and P).

Note that `\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.*

```
/*@ requires \valid(t+(0..n-1));
  @ 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* $\{t[0], \ldots, t[n-1]\}$ *is modified.*

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

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

// reachability in linked lists
/*@ inductive reachable{L}(struct list *root, struct list *to) {
  @   case empty: ∀ struct list *l; reachable(l,l) ;
  @   case non_empty: ∀ struct list *l1,*l2;
  @       \valid(l1) ∧ reachable(l1->next,l2) ⟹ reachable(l1,l2) ;
  @ }
*/

// The requires clause forbids to give a circular list
/*@ requires reachable(p,\null);
  @ 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 2.6.3 for details about the declaration of the predicate* `reachable`*.*

2.3.5 Default contracts, multiple contracts

A C 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 behaviors.

On the other hand a function may have no contract at all, or a contract with missing clauses. Missing `requires` and `ensures` clauses default to `\true`. If no `assigns` clause is given, it remains unspecified. If the function under consideration has only a declaration but no body, then it means that 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.

2.4 Statement annotations

Annotations on C statements are of three kinds:

- Assertions: allowed before any C statement or at end of blocks.
- Loop annotations: invariant, assigns clause, variant; allowed before any loop statement: `while`, `for`, and `do...while`.

compound-statement	::=	`{` *declaration** *statement** *assertion*$^+$ `}`
statement	::=	*assertion statement*
assertion	::=	`/*@ assert` *pred* `; */`
	|	`/*@ for` *id* (`,` *id*)* `: assert` *pred* `; */`

Figure 2.7: Grammar for assertions

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

- Statement contracts: allowed before any C statement, specifying their behavior in a similar manner to C function contracts.

2.4.1 Assertions

The syntax of assertions is given in Figure 2.7, as an extension of the grammar of C statements.

- `assert` p means that p must hold in the current state (the sequence point where the assertion occurs).
- The variant `for` $id_1, \ldots, id_k$ `: assert` p 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 must hold only for the considered behaviors.

2.4.2 Loop annotations

The syntax of loop annotations is given in Figure 2.8, as an extension of 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 following conditions hold.

- The predicate I holds before entering the loop (in the case of a `for` loop, this means right after the initialization expression).
- 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. 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 (interpreted in the current state), must remain allocated and has the same value as before entering the loop. In fact, the `loop assigns` clause specifies an inductive invariant for the locations wich are not member of L.

Remarks

- The `\old` construct is not allowed [1] in loop annotations. The `\at` form should be used to refer to another state (see Section 2.4.3).
- When a loop exits with `break` or `return` or `goto`, it is not required that the loop invariant holds. In such cases, locations that are not member of L can be disallocated or assigned between the end of the previous iteration and the exit statement.

Example 2.15 *Here is a continuation of example 2.11. Note 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;
  @*/
```

[1] This is not checked

```
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 (l+u)/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 include 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 nonnegative. Note that the value of m at loop exit might be negative. It does not compromise termination of the loop. Here is an example:

Example 2.16

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

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

General inductive invariants

It is actually allowed to pose an inductive invariant anywhere inside a loop body. For example, it makes sense for a `do` s `while` (c)`;` loop to contain an invariant right after statement s. Such an invariant is a kind of assertions, as shown in Figure 2.9.

Example 2.17 *In the following example, the natural invariant holds at this point (*`\max` *and* `\lambda` *are introduced later in Section 2.6.7). It would be less convenient to set an 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]));
```

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

Figure 2.9: Grammar for general inductive invariants

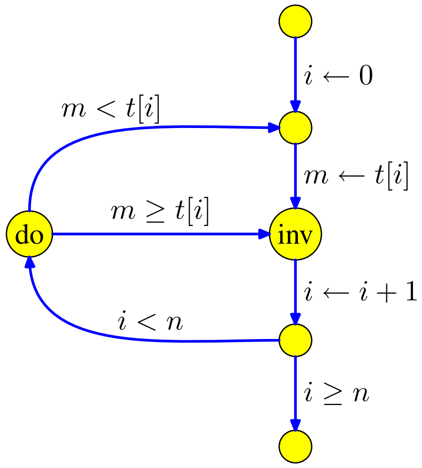
```
  @*/
double max(double t[], int n) {
  int i = 0; double m,v;
  do {
    v = t[i++];
    m = v > m ? v : m;
    /*@ invariant m ≡ \max(0,i−1,(\lambda integer k ; t[k])); */
  } while (i < n);
  return m;
}
```

More generally, loops can be introduced by `goto`s. As a consequence, such invariants may occur anywhere inside a function's body. The meaning is that the invariant holds at that point, much like an `assert`. Moreover, the invariant must be inductive, *i.e.* it must be preserved across a loop iteration. Several invariants are allowed at different places in a loop body. These extensions are useful when dealing with complex control flows.

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 go 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 smaller 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`*. It holds at loop entrance and is preserved (under the assumption of the loop condition* `y>0`*).*

Similarly, the following general invariants are not inductive:

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

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

since `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 `\at`(e, id) referring 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 2.12. This label must be declared in the same function as the occurrence of `\at`(e, id), but unlike `goto`s, more restrictive scoping rules must be respected:

- the label id must occur before the occurrence of `\at`(e, id) in the source;
- the label id must not be inside an inner block.

These rules are exactly the same rules as for the visibility of local variables within C statements (see [12], Section A11.1)

Default logic labels

There are four predefined logic labels: `Pre`, `Here`, `Old` and `Post`. `\old`(e) is in fact syntactic sugar for `\at`(e, Old).

- The label `Here` is visible in all statement annotations, where it refers to the state where the annotation appears; and in all contracts, where it refers to the pre-state for the `requires` and the `assumes, variant, terminates, ...` clause and the post-state for other clauses. It is also visible in data invariants, presented in Section 2.11.
- 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.
- The label `Pre` is visible in all statement annotations, and refers to the pre-state of the function it occurs in.
- The label `Post` is visible in `assigns` and `ensures` clauses.

Note that no logic label is visible in global logic declarations 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 2.6.9.

Example 2.20 *The code below implements the famous extended Euclid's algorithm for computing the greatest common divisor of two integers x and y, while 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 refer to the value of x and y in the pre-state of the function.*

```
/*@ requires x ≥ 0 ∧ y ≥ 0;
  @ behavior bezoutProperty:
  @   ensures (*p)*x+(*q)*y ≡ \result;
  @*/
int extended_Euclid(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;
}
```

Example 2.21 *Here is a toy example illustrating tricky issues with* `\at` *and labels:*

```
int i;
int t[10];

//@ ensures 0 ≤ \result ≤ 9;
int any();

/*@ assigns i,t[\at(i,Post)]
  @ ensures
  @   t[i] ≡ \old(t[\at(i,Here)]) + 1;
  @ ensures
  @   \let j = i; t[j] ≡ \old(t[j]) + 1;
  @*/
void f() {
  i = any();
  t[i]++;
}
```

The two `ensures` *clauses are equivalent. The simpler clause* `t[i] == \old(t[i]) + 1` *would be wrong because in* `\old(t[i])`*,* `i` *denotes the value of* i *n the pre-state.*

Also, the `assigns` *clause* `i,t[i]` *would be wrong too because again in* `t[i]`*, the value of* `i` *in the pre-state is considered.*

2.4.4 Statement contracts

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

The clauses `breaks`, `continues` and `returns` state properties on the program state which hold when the annotated statement terminates abruptly with the corresponding statement (`break`, `continue` or `return`). Inside these clauses, the construct `\old`(e) is allowed and denotes, like for statement contracts `assigns` and `ensures`, the value of e in the pre-state of the statement. For the `returns` case, the `\result` construct is allowed and bounds to the returned value (if not a `void` function).

Example 2.22 *Here is an example which illustrates each of these special clauses for statement contracts.*

$$
\begin{array}{rcl}
\textit{statement} & ::= & \texttt{/*@}\ \textit{statement-contract}\ \texttt{*/}\ \textit{statement} \\
\textit{statement-contract} & ::= & \underline{(\texttt{for}\ \textit{id}\ (\texttt{,}\ \textit{id})^*)^?}\ \textit{simple-behavior}\ \textit{named-behavior}^*\ \underline{\textit{abrupt-clause}^*} \\
\textit{abrupt-clause} & ::= & \underline{\texttt{breaks}\ \textit{predicate}\ \texttt{;}} \\
& | & \underline{\texttt{continues}\ \textit{predicate}\ \texttt{;}} \\
& | & \underline{\texttt{returns}\ \textit{predicate}\ \texttt{;}}
\end{array}
$$

Figure 2.10: Grammar for statement contracts

```
int f(int x) {

  while (x > 0) {

    /*@ breaks x % 11 ≡ 0 ∧ x ≡ \old(x);
      @ continues (x+1) % 11 ≢ 0 ∧ x % 7 ≡ 0 ∧ x ≡ \old(x)−1;
      @ returns (\result+2) % 11 ≢ 0 ∧ (\result+1) % 7 ≢ 0
      @          ∧ \result % 5 ≡ 0 ∧ \result ≡ \old(x)−2;
      @ ensures (x+3) % 11 ≢ 0 ∧ (x+2) % 7 ≢ 0 ∧ (x+1) % 5 ≢ 0
      @          ∧ x ≡ \old(x)−3;
      @*/
    {
      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 function. By default, a measure is an integer expression, and measures are compared using the usual ordering over integers (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 syntax

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

and loops are annotated similarly with the syntax

```
//@ 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

$$R(x, y) ::= x > y \texttt{ \&\& } x \geq 0$$

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

Example 2.23 *In Example 2.15, a loop variant* `u-l` *decreases at each iteration, and remains nonnegative, 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 binary predicate declared as

```
//@ predicate R(τ x, τ y)···
```

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

Example 2.24 *The following example illustrates 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. We call *cluster* a set of mutually recursive functions which is a strongly connected component of the call graph. Within each 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 2.25 *Here 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 2.26 *This example illustrates 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);
}
```

2.5.4 Non-terminating functions

EXPERIMENTAL

There are cases where a function is not supposed to terminate. For instance, the `main` function of a reactive program might be a `while(1)` which indefinitely waits for an event to process. More generally, a function can be expected to terminate only if some preconditions are met. In those cases, a `terminates` clause can be added to the contract of the function, under the following form:

```
//@ terminates p;
```

The semantics of such a clause is as follows: if p holds, then the function is guaranteed to terminate (more precisely, its termination must be proved). If such a clause is not present (and in particular if there is no function contract at all), it defaults to `terminates \true;` that is the function is supposed to always terminate, which is the expected behavior of most functions.

Note that nothing is specified for the case where p does not hold: the function may terminate or not. In particular, `terminates \false;` does not imply that the function loops forever. A possible specification for a function that never terminates is the following:

```
/*@ ensures \false;
    terminates \false;
*/
void f() { while(1); }
```

Example 2.27 *A concrete example of a function that may not always terminate is the* `incr_list` *function of example 2.14. In fact, another acceptable contract for this function is the following one:*

```
#include "list.h"
// this time, the specification accepts circular lists, but does not ensure
// that the function terminates on them (as a matter of fact, it does not).
/*@ assigns { q->hd | struct list *q ; reachable(p,q) } ;
  @ terminates reachable(p,\null);
  @*/
void incr_list(struct list *p) {
  while (p) { p->hd++ ; p = p->next; }
}
```

2.6 Logic specifications

The language of logic expressions used in annotations can be extended by declarations of new logic types, and new constants, logic functions and predicates. These declarations follows the classical setting of *algebraic specifications*. The grammar for these declarations is given in Figure 2.11.

2.6.1 Predicate and function definitions

New functions and predicates can be *defined* by explicit expressions, given after an equal sign.

Example 2.28 *The following definitions*

```
//@ predicate is_positive(integer x) = x > 0;
/*@ logic integer sign(real x) =
  @   x > 0.0 ? 1 : ( x < 0.0 ? -1 : 0);
  @*/
```

illustrates the definition of a new predicate `is_positive` *with an integer parameter, and a new logic function* `sign` *with a real parameter returning an integer.*

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

Figure 2.11: Grammar for global logic definitions

2.6.2 Lemmas

Lemmas are user-given propositions, a facility that might help theorem provers to establish validity of ACSL specifications.

Example 2.29 *The following lemma*

```
//@ lemma mean_property: ∀ integer x,y; x ≤ (x+y)/2 ≤ y;
```

is a useful hint for program like binary search.

Of course, a complete verification of an ACSL specification has to provide a proof for each lemma.

2.6.3 Inductive predicates

A predicate may also be defined by an inductive definition. The grammar for those style of definitions is given on Figure 2.12.

In general, an inductive definition of a predicate P has the form

```
/*@ inductive P(x1,...,xn) {
  @ case c1 :  p1;
...
  @ case ck :  pk;
  @ }   @*/
```

logic-def	::=	*inductive-def*
inductive-def	::=	`inductive` *poly-id parameters*$^?$ `{` *indcase** `}`
indcase	::=	`case` *poly-id* `:` *pred* `;`

Figure 2.12: Grammar for inductive definitions

where each c_i is an identifier and each p_i is a proposition.

The semantics of such a definition is that P is the least fixpoint of the cases, i.e. is the smallest predicate (in the sense that it is false the most often) satisfying the propositions $p_1, \ldots, p_k$. With this general form, the existence of a least fixpoint is not guaranteed, so tools might enforce syntactic conditions on the form of inductive definitions. A standard syntactic restriction could be to allow only propositions p_i of the form

`\forall` $y_1, \ldots, y_m$, h_1 `==>` $\cdots$ `==>` h_l `==>` $P(t_1, \ldots, t_n)$

where P occurs only positively in hypotheses $h_1, \ldots, h_l$ (definite Horn clauses, http://en.wikipedia.org/wiki/Horn_clause).

Example 2.30 *The following introduce a predicate* $\mathtt{isgcd}(x, y, d)$ *meaning that* d *is the greatest common divisor of* x *and* y*.*

```
/*@ inductive is_gcd(integer a, integer b, integer d) {
  @  case gcd_zero:
  @    ∀ integer n; is_gcd(n,0,n);
  @  case gcd_succ:
  @   ∀ integer a,b,d; is_gcd(b, a % b, d) ⟹ is_gcd(a,b,d);
  @ }
  @*/
```

This definition uses definite Horn clauses, hence is consistent.

Example 2.14 already introduced an inductive definition of reachability in linked-lists, and was also bases on definite Horn clauses thus consistent.

2.6.4 Axiomatic definitions

Instead of an explicit definition, one may introduce an *axiomatic* definitions for a set of types, predicates and logic functions, which amounts to declare the expected profiles and a set of axioms. The grammar for those constructions is given on Figure 2.13.

Example 2.31 *The following axiomatization introduce a theory of finite lists of integers a la LISP.*

```
/*@ axiomatic IntList {
  @  type int_list;
  @  logic int_list nil;
  @  logic int_list cons(integer n,int_list l);
  @  logic int_list append(int_list l1,int_list l2);
  @  axiom append_nil:
  @    ∀ int_list l; append(nil,l) ≡ l;
  @  axiom append_cons:
```

logic-def	::=	*axiomatic-decl*	
axiomatic-decl	::=	`axiomatic` *id* `{` *logic-decl** `}`	
logic-decl	::=	*logic-def*	
	|	*logic-type-decl*	
	|	*logic-const-decl*	
	|	*logic-predicate-decl*	
	|	*logic-function-decl*	
	|	*axiom-decl*	
logic-type-decl	::=	`type` *logic-type* `;`	
logic-type	::=	*id*	
	|	*id type-var-binders*	polymorphic type
logic-const-decl	::=	`logic` *type-expr poly-id* `;`	
logic-function-decl	::=	`logic` *type-expr* *poly-id parameters* `;`	
logic-predicate-decl	::=	`predicate` *poly-id parameters*$^?$ `;`	
axiom-decl	::=	`axiom` *poly-id* `:` *pred* `;`	

Figure 2.13: Grammar for axiomatic declarations

```
  @   ∀ integer n, int_list l1,l2;
  @      append(cons(n,l1),l2) ≡ cons(n,append(l1,l2));
  @ }
  @*/
```

Like inductive definitions, there is no syntactic conditions which would guarantee axiomatic definitions to be consistent. It is usually up to the user to ensure that the introduction of axioms does not lead to a logical inconsistency.

Example 2.32 *The following axiomatization*

```
/*@ axiomatic sign {
  @   logic integer sign(real x);
  @   axiom sign_pos: ∀ real x; x ≥ 0. ⟹ sign(x) ≡ 1;
  @   axiom sign_neg: ∀ real x; x ≤ 0. ⟹ sign(x) ≡ −1;
  @ }
  @*/
```

is inconsistent since it implies `sign(0.0) == 1` *and* `sign(0.0) == -1`*, hence* `-1 == 1`

2.6.5 Polymorphic logic types

EXPERIMENTAL

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

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

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

Figure 2.14: Grammar for higher-order constructs

One can then consider for instance list of integers (`list <integer>`), list of pointers (e.g. `list <char*>`), list of list of reals (`list <list <real> >`[2]), etc.

The grammar of Figure 2.11 contains rules for declaring polymorphic types and using polymorphic type expressions.

2.6.6 Recursive logic definitions

Explicit definitions of logic functions and predicates can be recursive. Declarations in the same bunch of logic declarations are implicitly mutually recursive, so that mutually recursive functions are possible too.

Example 2.33 *The following logic declaration*

```
/*@ logic integer max_index{L}(int t[],integer 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. In essence, a recursive definition of the form `f(args) = e;` where `f` occurs in expression `e` is just a shortcut for axiomatic declaration of `f` with an axiom `\forall args; f(args) = e`. In other words, recursive definitions are not guaranteed to be consistent, in the same way that axiomatics may introduce inconsistency. Of course, tools might provide a way to check consistency.

2.6.7 Higher-order logic constructions

EXPERIMENTAL

Figure 2.14 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 `\numof` for which it should have a boolean value. Their meanings are given as follows:

$$
\begin{aligned}
\texttt{\textbackslash max}(i,j,f) &= \max\{f(i),f(i+1),\ldots,f(j)\}\\
\texttt{\textbackslash min}(i,j,f) &= \min\{f(i),f(i+1),\ldots,f(j)\}\\
\texttt{\textbackslash sum}(i,j,f) &= f(i)+f(i+1)+\cdots+f(j)
\end{aligned}
$$

[2] In this latter case, note that the two '>' must be separated by a space, to avoid confusion with the shift operator.

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

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

2.6.8 Concrete logic types

EXPERIMENTAL

Logic types may not only be declared but also be given a definition. Defined logic types can be either under record types, or sum types. These definitions may be recursive. For 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 in Figure 2.15

Example 2.35 *The declaration*

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

introduces a concrete definition of finite lists. The logic definition

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

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

2.6.9 Hybrid functions and predicates

Logic functions and predicates may take both (pure) C types and logic types arguments. Such an hybrid predicate (or function) can either be defined with the same syntax as before (or axiomatized).

Be it defined either directly by an expression or through a set of axioms, an hybrid function (or predicate) usually depends on one or more program points, because it depends upon memory states, *via* expressions such as:

logic-def	::=	`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
	|	*cst*	numeric constant
	|	`{` (`.` *id* `=` *pat*)* `}`	record pattern
	|	`(` *pat* (`,` *pat*)* `)`	tuple pattern
	|	*pat* `as` *id*	pattern binding

Figure 2.15: Grammar for concrete logic types and pattern-matching

- pointer dereferencing: `*p`, `p->f`;
- array access: `t[i]`;
- address-of operator: `&x`;
- built-in predicate 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, as shown on Figure 2.16. Expressions as above must then be enclosed into the `\at` construct to refer to a given label. However, to ease reading of such logic expressions, it is allowed to omit a label whenever there is only one label in the context.

Example 2.36 *The following annotations declare a function which returns the number of occurrences of a given double in an array of doubles between the given indexes, together with the related axioms. It should be noted that without labels, this axiomatization would be inconsistent, since the function 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.

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 2.16: Grammar for logic declarations with labels

```
/*@ axiomatic NbOcc {
  @   // 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);
  @  axiom nb_occ_empty{L}:
  @   ∀ double t[], e, integer i, j;
  @     i > j ⟹ nb_occ(t,i,j,e) ≡ 0;
  @  axiom nb_occ_true{L}:
  @   ∀ double t[], e, integer i, j;
  @     i ≤ j ∧ t[j] ≡ e ⟹
  @       nb_occ(t,i,j,e) ≡ nb_occ(t,i,j−1,e) + 1;
  @  axiom nb_occ_false{L}:
  @   ∀ double t[], e, integer i, j;
  @     i ≤ j ∧ t[j] ≢ e ⟹
  @       nb_occ(t,i,j,e) ≡ nb_occ(t,i,j−1,e);
  @ }
  @*/
```

Example 2.37 *This second example defines a predicate which indicates whether two arrays of the same size are a permutation of each other. It illustrates the use of more than a single label. Thus, the* `\at` *operator is mandatory here. Indeed the two arrays may come from two distinct memory states. Typically, one of the post condition of a sorting function would be* `permut{Pre,Post}(t,t)`.

```
/*@ axiomatic Permut {
  @  // 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);
  @  axiom permut_refl{L}:
  @   ∀ double t[], integer n; permut{L,L}(t,t,n);
  @  axiom permut_sym{L1,L2} :
  @   ∀ double t1[], t2[], integer n;
  @     permut{L1,L2}(t1,t2,n) ⟹ permut{L2,L1}(t2,t1,n) ;
  @ axiom permut_trans{L1,L2,L3} :
  @   ∀ double t1[], t2[], 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[], t2[], integer i, j, 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);
@ }
@*/
```

2.6.10 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 {
  @       case Nil : 0
  @       case Cons(h,t) : 1+length(t) } ;
  @
  @   logic A fold_right<A,B>((A -> B -> B) f, list<A> l, B acc) =
  @     \match l {
  @       case Nil : acc
  @       case 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

The following built-in functions and predicate allows to deal with the memory state:

- `\base_addr` returns the base address of an allocated pointer

  `\base_addr : 'a * → char*`

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

$$\texttt{\textbackslash block\_length} : \texttt{'a} \ \star \rightarrow \texttt{size\_t}$$

- `\valid` applies to a set of terms (see Section 2.3.4) of some pointer type. `\valid`(s) holds if and only if dereferencing any $p \in s$ is safe. In particular, `\valid(\empty)` holds.

Some shortcuts are provided:

- `\null` is an extra notation for the null pointer (*i.e.* a shortcut for `(void*)0`). Note that as in C itself (see [11], par. 6.3.2.3), the constant `0` can have any pointer type.
- `\offset`(p) returns the offset between p and its base address

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

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.

2.7.3 Allocation and deallocation

EXPERIMENTAL

The following built-in predicates allow to deal with allocation and deallocation of memory blocks. They can be used in a postcondition

- `\fresh`(p) indicates that p was not allocated in the pre-state.
- `\freed`(p) , indicates that p was allocated in the pre-state but that it is not the case in the post-state.

2.8 Sets as first-class values

Sets of terms, as defined in Section 2.3.4, can be used as first-class values in annotations. All the elements of such a set must share the same type (modulo the usual implicit conversions). Sets have the built-in type $set < A >$ where A is the type of terms contained in the set.

In addition, it is possible to consider sets of pointers to values of different types. In this case, the set is of type $set < char* >$ and each of its elements e is converted to $(char*)e + (0..sizeof(*e) - 1)$ [3].

Example 2.38 *Here is an example where we defined the* footprint *of a structure, that is the set of locations that can be accessed from an object of this type.*

[3] Currently mixing pointers of different types in such a set leads to a `set<void*>` value. The expansion must be done manually.

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

//@ logic set<char*> footprint(struct S s) = \union (s.x,s.y) ;

/*@ logic set<char*> footprint2(struct S s) =
  @     \union (s.x,(char*)s.y+(0..sizeof(s.y)-1)) ;
  @*/

/*@ axiomatic Conv {
    axiom conversion: ∀ struct S s;
      footprint(s) ≡ \union
(s.x,(char*) y + (0 .. sizeof(int) - 1));
    }
*/
```

Notice that in the first definition, since union is made with a set<char> and a set<int*>, the result is a set<char*> (accordingly to typing of union). In other words, the two definitions above are equivalent. This logic function can be used as argument of* `\separated` *or in assigns clause.*

Thus, the `\separated` predicate satisfies the following property:

$$\texttt{\textbackslash separated}(l_1 : set < \tau_1* >, l_2 : set < \tau_2* >) \texttt{<==>}$$
$$\forall \tau_1 * p; \forall \tau_2 * q; p \in l_1 \land q \in l_2 \rightarrow$$
$$\forall \texttt{integer} i, j; 0 \leq i < \texttt{sizeof}(\tau_1) \land 0 \leq j < \texttt{sizeof}(\tau_2) \rightarrow (\texttt{char*})p + i \neq (\texttt{char*})q + j$$

and a clause `assigns` $l_1, .., l_n$ is equivalent to the postcondition

$$\forall char * p; \texttt{\textbackslash separated}(\texttt{\textbackslash union}(\&l_1, .., \&l_n), p) \texttt{==>} *p \texttt{==} \texttt{\textbackslash old}(*p)$$

2.9 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.* by calling the `exit` function. Such a behavior can be written under the following form:

exit-behavior	::=	`exit_behavior :`
		(*assumes-clause* | *assigns-clause* | *ensures-clause*)*

In an *ensures-clause*, `\result` is bound to the return code, *e.g* the value returned by `main` or the argument passed to `exit`.

2.10 Dependencies information

EXPERIMENTAL

An extended syntax of `assigns` clauses, described in Figure 2.17 allows to specify data dependencies and *functional expressions*.

Such a clause indicates that the assigned values can only depend upon the locations mentioned in the `\from` part of the clause. Again, this is an over-approximation: all of the locations involved in the

assigns-clause	::=	`assigns` *locations* (`\from` *locations*)$^?$ `;`
	|	`assigns` *term* `\from` *locations* = *term* `;`

Figure 2.17: Grammar for dependencies information

computation of the modified values must be present, but some of locations might not be used in practice. If the `from` clause is absent, all of the locations reachable at the given point of the program are supposed to be used. Moreover, for a single location, it is possible to give the precise relation between its final value and the value of its dependencies. This expression is evaluated in the pre-state of the corresponding contract.

Example 2.39 *The following example is a variation over the* `array_sum` *function in example 2.34, in which the values of the array are added to a global variable* `total`.

```
double total = 0.0;

/*@ requires n ≥ 0 ∧ \valid(t+(0..n−1)) ;
  @ assigns total
      \from t[0..n−1] = total + \sum(0,n−1,\lambda int k; t[k]);
  @*/
void array_sum(double t[],int n) {
  int i;
  for(i=0; i < n; i++) total += t[i];
  return;
}
```

Example 2.40 *The composite element modifier operators are useful additional constructs for such functional expressions.*

```
struct buffer { int pos ; char buf[80]; } line;

/*@ requires 80 > line.pos ≥ 0 ;
  @ assigns line
      \from line = { line for buf = { line.buf for [line.pos] = '\0' };
  @*/
void add_eol() {
  line.buf[line.pos] = '\0' ;
}
```

2.11 Data invariants

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

- *global* invariants and *type* invariants: the former only apply to specified global variables, whereas the latter are associated to a static type, and apply to any variables of the corresponding type;
- *strong* invariants and *weak* invariants: strong invariants must be valid at any time during program execution (more precisely at any *sequence point* as defined in the C standard), whereas weak invariants must be valid at *function boundaries* (function entrance and exit) but can be violated in between.

declaration	::=	`/*@` *data-inv-decl* `*/`
data-inv-decl	::=	*data-invariant* | *type-invariant*
data-invariant	::=	*inv-strength*? `global invariant` *id* `:` *pred* `;`
type-invariant	::=	*inv-strength*? `type invariant` *id* `(` *C-type-expr id* `)` `=` *pred* `;`
inv-strength	::=	`weak` | `strong`

Figure 2.18: Grammar for declarations of data invariants

The syntax for declaring data invariants is given in Figure 2.18. The strength modifier defaults to `weak`.

Example 2.41 *In the following example, we declare*

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

```
int a;
//@ global 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 ;
```

2.11.1 Semantics

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

- Weak global invariants are properties which apply to global data and hold at any function entrance and function exit.
- Strong global invariants are properties which apply to global data and hold at any step during execution (starting after initialization of these data).
- A weak type invariant on type τ must 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 must also satisfy its weak invariant at function exit. Notice that it says nothing of fields, array elements, memory locations, etc. of type τ.

- A strong type invariant on type τ must hold at any step during 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 must also satisfy its strong invariant at function exit. Again, it says nothing of fields, array elements, memory locations, etc. of type τ.

Example 2.42 *The following example illustrates the use of a data invariant on a local static variable.*

```
void out_char(char c) {

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

  col++;
  if (col ≥ 80) col = 0;

}
```

Example 2.43 *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) =
  @  \at(\valid_range(t1,0,n),L1) ∧ \at(\valid_range(t2,0,n),L2) ∧
  @  \numof(0,n,\lambda integer i; \at(t1[i],L1) ≡ BLUE) ≡
  @  \numof(0,n,\lambda integer i; \at(t2[i],L2) ≡ BLUE)
  @   ∧
  @  \numof(0,n,\lambda integer i; \at(t1[i],L1) ≡ WHITE) ≡
  @  \numof(0,n,\lambda integer i; \at(t2[i],L2) ≡ WHITE)
  @  ∧
  @  \numof(0,n,\lambda integer i; \at(t1[i],L1) ≡ RED) ≡
  @  \numof(0,n,\lambda integer i; \at(t2[i],L2) ≡ RED);
  @*/

/*@ 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 ∧
    @   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;
    }
  }
}
```

2.11.2 Model variables and model fields

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

The precise syntax for declaring model variables and fields is given in Figure 2.19. It is presented as additions to the regular C grammar for variable declarations and structure field declarations.

Informal semantics of model variables is as follows.

- Model variables can only appear in specifications. They are not lvalues, thus they 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 implicitly existentially quantified over those variables;
  - the postconditions are universally quantified over the old values of model variables, and existentially quantified over the new values.

Thus, in practice, the only way to prove 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 2.44 *Here is an example of a specification for 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. the values that have already 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 value returned is not in the set of forbidden values, thus it is "fresh";*
- *the new set of forbidden values contains both the value returned and the previous forbidden values.*

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

Figure 2.19: Grammar for declarations of model variables and fields

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

```
/* implementation */
int gen() {
  static int x = 0;
  /*@ global 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 C type. Also, the semantics above is closer to the one of B machines [1]. It has to be noticed that program verification with model variables does not have a well-established theoretical background [18, 16], so we deliberately do not provide a precise semantics in this document .

2.12 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 2.20, 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).

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*	
statements-ghost	::=	`/*@ ghost` *ghost-statement*+ `*/`	
ghost-selection-statement	::=	*C-selection-statement*	
	|	`if (` *expression* `)` *statement* `/*@ ghost else` *C-statement*+ `*/`	
struct-declaration	::=	*C-struct-declaration*	
	|	`/*@ ghost` *C-struct-declaration* `*/`	ghost field

Figure 2.20: Grammar for ghost statements

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 [4] 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.

[4] Not checked in the current implementation

- 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 non-ghost enclosing 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 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 2.45 *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 2.46 *The following example shows some invalid ghost statements:*

```
int f (int x, int y) {
  //@ ghost int z = x + y;
  switch (x) {
  case 0: return y;
  //@ ghost case 1: z=y;
  // above statement is correct.
  //@ ghost case 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 specifications using ghost variables modifications are executable.

Example 2.47 *The example 2.44 can also be specified with a ghost variable instead of a model variable:*

```
//@ ghost set<integer> forbidden = emptyset;

/*@ assigns forbidden;
  @ ensures ¬ Set::in(\result,\old(forbidden)
  @    ∧ Set::in(\result,forbidden)
       ∧ Set::subset(\old(forbidden),forbidden);
  @*/
int gen() {
  static int x = 0;
  /*@ global invariant I: ∀ integer k;
    @     Set::mem(k,forbidden) ⟹ x > k;
    @*/
  x++;
  //@ ghost forbidden = Set::union(Set::single(x),forbidden);
  return x;
}
```

2.12.1 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.

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

Figure 2.21: Grammar for volatile constructs

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 2.21, 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 2.48 *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;
}
```

2.13 Undefined values, dangling pointers

2.13.1 Initialization

`\initialized` is a predicate taking a set of l-values as argument and means that each l-value in this set is initialized.

Example 2.49 *In the following, the assertion is true.*

```
int f(int n) {
  int x;
```

```
  if (n > 0) x = n ; else x = -n;
  //@ assert \initialized(x);
  return x;
}
```

2.13.2 Unspecified values

`\specified` is a predicate taking a set of l-values as argument and means that each l-value in this set has a *specified value*: its value is not a dangling pointer (that is the address of a local variable which is not in the scope anymore)

Example 2.50 *In the following, the assertion is not true.*

```
int* f() {
  int a;
  return &a;
}

int* g() {
  int* p = f();
  //@ assert \specified(p);
  return p+1;
}
```

Chapter 3

Libraries

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

This chapter is devoted to librairies of specification, built upon the ACSL specification language. Section 3.2 describes additional predicates introduced by the Jessie plugin of Frama-C, to propose a slightly higher level of annotation.

3.1 Libraries of logic specifications

A standard library is provided, in the spirit of the List module of Section 2.6.10

3.1.1 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.

3.1.2 Finite lists

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

3.1.3 Sets and Maps

Finite sets, finite maps, in ZB-style.

3.2 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 [19]). 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.

3.2.1 Abstract level of pointer validity

Int the particular setting described above, it is possible to introduce the following 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`$(p,0,0)$.

3.2.2 Strings

EXPERIMENTAL

The logic function

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

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 strcpy(char *dest, const char *src);
```

3.3 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.

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 interesting possible extension regards "temporal" properties in a large sense, such as liveness properties, which can sometimes be simulated by regular specifications with ghost variables [10], 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 [12, 11], §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 $e\texttt{->}id$, 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 [14]), ACSL is notably different from JML in two crucial aspects:

- ACSL is a BISL for C, a low-level structured language, while JML is a BISL for Java, an object-oriented inheritance-based high-level language. Not only the language features are not the same but the programming styles and idioms are very different, which entails also 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 [20, 5], that adds strong typing to C by relying on

RAC too, shows that 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 [6] thus reduces the checking power of annotations in order to preserve modularity. On the contrary, we choose not to restrain 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 remainder of this chapter, we describe these differences in further details.

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. First, 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)); */
```

where predicate `\valid` is the one defined in Section 2.7.1. (note that `\valid`$(a + 0..(-1))$ is the same as `\valid(\empty)` and thus is true regardless of the validity of a itself). 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 postconditions 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. Here, Our two behaviors are clearly mutually exclusive, and, since `n` is an `unsigned int`, our they cover all the possible cases. We could have specified that as well, by adding the following lines in the contract (see Section 2.3.3).

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

ACSL contracts vs. JML ones

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

A JML contract is either *lightweight* or *heavyweight*. 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 A1;
  @   requires R1;
  @   ensures E1;
  @ behavior x2:
  @   requires A2;
  @   requires R2;
  @   ensures E2;
  @*/
```

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 post-state:

$$Q_1 \;\&\&\; Q_2 \;\&\&\; (\texttt{\textbackslash old}(A_1 \;\&\&\; R_1) \texttt{==>} E_1) \;\&\&\; (\texttt{\textbackslash old}(A_2 \;\&\&\; R_2) \texttt{==>} E_2)$$

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:

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

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

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

Thus, ACSL allows to distinguish between the clauses that control which behavior is active (the `assumes` clauses) and the clauses that are preconditions for a particular behavior (the internal `requires` clauses). In addition, as mentioned above, there is by default no requirement in ACSL for the specification to be complete (The last part of the JML condition on the pre-state). If desired, this has to be precised explicitely with a `complete behaviors` clause as seen in Section 2.3.3.

A.2.2 Deductive verification vs. RAC

Sugar-free behaviors

As explained in details in [21], 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. (see [17]) and the tool ESC/Java2 for automatic deductive verification of JML specifications (see [4]).

One issue with such a desugaring approach is the complexity of the transformations involved, as *e.g.* for desugaring assignable clauses between multiple *spec-cases* in JML [21]. Another issue is precisely that tools only see one global contract, instead of multiple independent behaviors, that could be analyzed separately in more detail. 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 [15]. 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
$(\texttt{\textbackslash forall}\ \tau\ x; P; Q)$	$(\texttt{\textbackslash forall}\ \tau\ x; P\texttt{==>}Q)$
$(\texttt{\textbackslash exists}\ \tau\ x; P; Q)$	$(\texttt{\textbackslash exists}\ \tau\ x; P\texttt{\&\&}Q)$
$(\texttt{\textbackslash max}\ \tau\ x; a\texttt{<=}x\texttt{<=}b; f)$	$\texttt{\textbackslash max}(a, b, \texttt{\textbackslash lambda}\ \tau\ x; f)$

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:

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

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 sets

We consider the typing judgement $\Gamma, \Lambda \vdash s : \tau, b$ meaning that s is a set of terms of type τ, which is moreover a set of locations if the boolean b is true. Γ is the C environment and Λ is the logic environment.

Rules:

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

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

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

$$\frac{id : \tau \quad s : set < struct\ S* >}{\vdash s- > id : set < \tau >}$$

$$\frac{\Gamma, b \cup \Lambda \vdash e : tset\tau}{\Gamma, \Lambda \vdash \{e \mid b; P\} : tset\tau}$$

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

A.4 Specification Templates

This section describes some common issues that may occur when writing an ACSL specification and proposes some solution to overcome them

A.4.1 Accessing to a C variable that is masked

In some situation, it is necessary to use in the specification a C variable that is masked at the given program point. For instance, a function contract may need to refer to a global variable that has the same name as a function parameter, as in the following code:

```
int x;
//@ assigns x;
int g();

int f(int x) {
  // ...
  return g();
}
```

In order to write the `assigns` clause for `f`, we must access the global variable `x`, since `f` calls `g`, which can modify `x`. This is not possible with C scoping rules, as `x` refers to the parameter of `f` in the scope of the function.

A solution is to use a ghost pointer to `x`, giving rise to the following code:

```
int x;

//@ ghost int* const ghost_ptr_x = &x;

//@ assigns x;
int g();

//@ assigns *ghost_ptr_x;
```

```
int f(int x) {
  // ...
  return g();
}
```

A.5 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 boolean        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;

/*@ strong 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 boolean 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;

/*@ strong 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 boolean     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;

/*@ strong 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 boolean     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 ;
  @
  @ strong 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;
}
```

A.6 Changes

A.6.1 Version 1.x

- Fixed typos in the examples corresponding to features implemented in Frama-C

A.6.2 Version 1.4

- Introduction of `axiomatic` to gather predicates, logic functions, and their defining axioms.
- added specification templates appendix for common specification issues
- use of sets as first-class term as been precised
- fixed semantics of predicated `\separated`

A.6.3 Version 1.3

- Functional update of structures
- Terminates clause in function behaviors
- Typos reported by David Mentre

A.6.4 Version 1.2

This is the first public release of this document.

Bibliography

[1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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] The Frama-C framework for analysis of C code. http://frama-c.cea.fr/.

[10] 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.

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

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

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

[14] Gary Leavens. Jml. http://www.eecs.ucf.edu/~leavens/JML/.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] David Stevenson et al. An american national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

[23] Wikipedia. First order logic. http://en.wikipedia.org/wiki/First_order_logic.

[24] Wikipedia. IEEE 754. http://en.wikipedia.org/wiki/IEEE_754-1985.

List of Figures

Index