HAL
open science

Abstract domains for bit-level machine integer and floating-point operations

Antoine Miné

To cite this version:

Antoine Miné. Abstract domains for bit-level machine integer and floating-point operations. WING'12 - 4th International Workshop on invariant Generation, Jun 2012, Manchester, United Kingdom. pp.16. hal-00748094

HAL Id: hal-00748094

https://hal.science/hal-00748094v1

Submitted on 4 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Abstract Domains for Bit-Level Machine Integer and Floating-point Operations

Antoine Miné*

CNRS & École Normale Supérieure
45, rue d'Ulm
75005 Paris, France
mine@di.ens.fr

Abstract

We present a few lightweight numeric abstract domains to analyze C programs that exploit the binary representation of numbers in computers, for instance to perform "compute-through-overflow" on machine integers, or to directly manipulate the exponent and mantissa of floating-point numbers. On integers, we propose an extension of intervals with a modular component, as well as a bitfield domain. On floating-point numbers, we propose a predicate domain to match, infer, and propagate selected expression patterns. These domains are simple, efficient, and extensible. We have included them into the Astrée and AstréeA static analyzers to supplement existing domains. Experimental results show that they can improve the analysis precision at a reasonable cost.

1 Introduction

Semantic-based static analysis is an invaluable tool to help ensuring the correctness of programs as it allows discovering program invariants at compile-time and fully automatically. Abstract interpretation [9] provides a systematic way to design static analyzers that are sound but approximate: they infer invariants which are not necessarily the tightest ones. A central concept is that of abstract domains, which consist of a set of program properties together with a computer representation and algorithms to compute sound approximations in the abstract of the effect of each language instruction. For instance, the interval domain [9] allows inferring variable bounds. Bound properties allow expressing the absence of many run-time errors (such as arithmetic and array overflows) but, due to approximations, the inferred bounds may not be sufficiently precise to imply the desired safety assertions (e.g., in the presence of loops). An effective static analyzer for run-time errors, such as Astrée [7], uses additional domains to infer local and loop invariants of a more complex form (e.g., octagons [20]) and derive tighter bounds.

Most numeric domains naturally abstract an ideal semantics based on perfect integers or rationals, while computers actually use binary numbers with a fixed number of digits. One solution is to adapt the domains to take into account hardware limitations: overflows are detected and treated as errors, while floating-point semantics is simulated by introducing rounding errors [17]. While this works well in many cases, it is not sufficient to analyze programs that perform overflows on purpose (expecting a wrap-around semantics) or that rely on the precise binary representation of numbers. The goal of this article is to propose a set of simple, lightweight numeric abstract domains that are aware of these aspects.

*This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France.

```
char add1(char x, char y) {
  return (char)
    ((unsigned char)x +
     (unsigned char)y);
}
```

(a)

```
char add2(char x, char y) {
  unsigned register r1,r2,r3;
  r1 = x; r2 = y;
  r3 = r1 + r2;
  return r3;
}
```

(b)

Figure 1: Integer "compute-through-overflow" examples. `char` are assumed to be signed.

```
union u { int i[2]; double d; };
double cast(int i) {
  union u x,y;
  x.i[0] = 0x43300000;
  y.i[0] = x.i[0];
  x.i[1] = 0x80000000;
  y.i[1] = i ^ x.i[1];
  return y.d - x.d;
}
```

(a)

```
double sqrt(double d) {
  double r;
  unsigned* p = (unsigned*)&d;
  int e = (*p & 0x7fe00000) >> 20;
  *p = (*p & 0x801fffff) | 0x3fe00000;
  r = ((c1*d+c2)*d+c3)*d+c4;
  *p = (e/2 + 511) << 20;
  p[1] = 0;
  return d * r;
}
```

(b)

Figure 2: Floating-point computations exploiting the IEEE binary representation. On the right, `c1` to `c4` are unspecified constant coefficients of a polynomial approximation. A 32-bit big-endian processor is assumed (e.g., PowerPC).

1.1 Motivating Examples

Figure 1.a presents a small C function that adds two signed bytes (`char`) by casting them to unsigned bytes before the addition and casting the result back to signed bytes. The function systematically triggers an overflow on negative arguments, which is detected by an analyzer such as Astrée. Additionally, on widespread architectures, the return value equals `x+y` due to wrap-around. This programming pattern is used in popular industrial code generators such as TargetLink [10] and known as "compute-through-overflow." An analysis not aware of wrap-around will either report dead code (if overflows are assumed to be fatal) or return $[-128, 127]$ (if overflows produce full-range results). Even a wrapping-aware interval analysis will return an imprecise interval for arguments crossing zero, e.g. $[-1, 0]$, as the first cast maps $\{-1, 0\}$ to $\{255, 0\}$ and intervals cannot represent non-convex properties (see Sec. 2.6).

A variant is shown in Fig. 1.b, where the casts are implicit and caused by copies between variables of different types. This pattern is used to ensure that arithmetic computations are performed in CPU registers only, using a pool of `register` variables with irrelevant signedness (i.e., register allocation is explicit and not entrusted to the compiler).

Figure 2.a presents a C function exploiting the binary representation of floating-point numbers based on the IEEE standard [13]. It implements a conversion from 32-bit integers to 64-bit floats by first constructing the float representation for $\texttt{x.d} = 2^{52} + 2^{31}$ and $\texttt{y.d} = 2^{52} + 2^{31} + \texttt{i}$ using integer operations and then computing $\texttt{y.d} - \texttt{x.d} = \texttt{i}$ as a float subtraction. This code is similar to the assembly code generated by compilers when targeting CPUs missing the conversion instruction (such as PowerPC). Some code generators choose to provide their own C implementation instead of relying on the compiler (for instance, to improve the traceability of

the assembly code). Figure 2.b exploits the binary encoding of floats to implement a square root: the argument is split into an exponent `e` and a mantissa in $[1, 4]$ (computed by masking the exponent in `d`); then the square root of the mantissa is evaluated through a polynomial, while the exponent is simply halved. In both examples, a sound analyzer not aware of the IEEE floating-point encoding will return the full float range.

These examples may seem disputable, yet they are representative of actual industrial codes (the examples have been modified for the sake of exposition). The underlying programming patterns are supported by many compilers and code generators. An industrial-strength static analyzer is expected to accept existing coding practices and handle them precisely.

1.2 Contribution

We introduce a refined concrete semantics taking bit manipulations into account, and present several abstractions to infer precise bounds for the codes in Figs. 1–2.

Section 2 focuses on integers with wrap-around: we propose an interval domain extended with a modular component and a bitfield domain abstracting each bit separately. Handling the `union` type and pointer cast from Fig. 2 requires a specific memory model, which is described in Sec. 3. Section 4 presents a bit-level float domain based on pattern matching enriched with predicate propagation. Section 5 presents experimental results using the Astrée and AstréeA static analyzers. Finally, Sec. 6 concludes.

The domains we present are very simple and lightweight; they have a limited expressiveness. They are intended to supplement, not replace, classic domains, when analyzing programs featuring bit-level manipulations. Moreover, they are often slight variations on existing domains [9, 15, 23, 19, 20]. We stress the fact that these domains and the change of concrete semantics they require have been incorporated into existing industrial analyzers, to enrich the class of programs they can analyze precisely, at low cost and with no precision regression on previously analyzed codes.

1.3 Related Work

The documentation [2] for the PolySpace analyzer suggests removing, prior to an analysis, all computes-through-overflows and provides a source filter based on regular expressions to do so. This solution is fragile and can miss casts (e.g., when they are not explicit, as in Fig. 1.b) or cause unsoundness (in case the pattern is too inclusive and transforms unrelated code parts), while the solution we propose is semantic-based.

Various domains supporting modular arithmetics have been proposed, such as simple [11] and affine congruences [12, 24]. Masdupuy introduced interval congruences [15] to analyze array indices; our modular intervals are a slightly simpler restriction and feature operators adapted to wrap-around. Simon and King propose a wrap-around operator for polyhedra [26]; in addition to being costly, it outputs convex polyhedra while our examples require the inference of non-convex invariants locally. Abstracting each bit of an integer separately is a natural idea that has been used, for instance, by Monniaux [23] and Regehr et al. [25]. Brauer et al. [8] propose a bit-blasting technique to design precise transfer functions for small blocks of integer operations, which can bypass the need for more expressive (e.g., disjunctive) local invariants.

We are not aware of any abstract domain able to handle bit-level operations on floating-point numbers. Unlike classic predicate abstraction [6], our floating-point predicate domain includes its own fast and ad-hoc (but limited) propagation algorithm instead of relying on an external generic tool.

$$\textit{int-type} ::= (\texttt{signed} \mid \texttt{unsigned})?\ (\texttt{char} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long} \mid \texttt{long long})\ \ n? \qquad (n \in \mathbb{N}^*)$$

$$\textit{bitsize} : \textit{int-type} \to \mathbb{N}^*$$
$$\textit{signed} : \textit{int-type} \to \{\text{true}, \text{false}\}$$

$$\textit{range}(t) \stackrel{\text{def}}{=} \begin{cases} [0, 2^{\textit{bitsize}(t)} - 1] & \text{if } \neg \textit{signed}(t) \\ [-2^{\textit{bitsize}(t)-1}, 2^{\textit{bitsize}(t)-1} - 1] & \text{if } \textit{signed}(t) \end{cases}$$

Figure 3: Integer C types and their characteristics.

2 Integer Abstract Domains

2.1 Concrete Integer Semantics

In this section, we focus on integer computations. Before designing a static analysis, we need to provide a precise, mathematical definition of the semantics of programs. We base our semantics on the C standard [5], extended with hypotheses on the representation of data-types necessary to analyze the programs in Fig. 1.

The C language mixes operators based on mathematical integers (addition, etc.) and operators based on the binary representation of numbers (bit-wise operators, shifts). At the hardware level, however, all integer computations are performed in registers of fixed bit-size. Thus, one way to define the semantics is to break it down at the bit level (i.e., "bit-blasting" [8]). We choose another route and express the semantics using classic mathematical integers in $\mathbb{Z}$. Our semantics is higher-level than a bit-based one, which provides some advantages: on the concrete level, it makes the classic arithmetic C operations (`+`, `-`, `*`, `/`, `%`) straightforward to express; on the abstract level, it remains compatible with abstract domains expressed on perfect numbers (such as polyhedra). We show that this choice does not preclude the definition of bit-wise operators nor bit-aware domains.

2.2 Integer Types

Integer types in C come in different sizes and can be signed or unsigned. We present in Fig. 3 the type grammar for integers, *int-type*, including bitfields that can only appear in structures (when a bit size n is specified). The bit size and signedness of types are partly implementation-specific. We assume that they are specified by two maps: $\textit{bitsize} : \textit{int-type} \to \mathbb{N}^*$ and $\textit{signed} : \textit{int-type} \to \{\text{true}, \text{false}\}$. Moreover, we assume that unsigned integers are represented using a pure binary representation: $b_{n-1} \cdots b_0 \in \{0, 1\}^n$ represents $\sum_{i=0}^{n-1} 2^i b_i$, and signed integers use two's complement representation: $b_{n-1} \cdots b_0 \in \{0, 1\}^n$ represents $\sum_{i=0}^{n-2} 2^i b_i - 2^{n-1} b_{n-1}$. Although this is not required by the C standard, it is the case for all the popular architectures.[1] The *range* (i.e., the set of acceptable values) of each type is derived as in Fig. 3.

2.3 Integer Expressions

We consider here only a pure, side-effect-free, integer fragment of C expressions, as depicted in Fig. 4. To stay concise, we include only arithmetic and bit-wise operators and casts. Moreover, statements are reduced to assignments and assertions (which are sufficient to model programs as control-flow graphs). We perform a static transformation that makes all wrap-around effects

[1] The C standard allows some features that we do not handle: padding bits, trap representations, one's complement representations or sign-magnitude representations of negative numbers, and negative zeros.

$$
\begin{array}{rcll}
\mathit{expr} & ::= & n & \textit{(constant } n \in \mathbb{Z}\textit{)} \\
 & | & V & \textit{(variable } V \in \mathcal{V}\textit{)} \\
 & | & \texttt{(}\mathit{int\text{-}type}\texttt{)}\ \mathit{expr} & \textit{(cast)} \\
 & | & \diamond\, \mathit{expr} & \textit{(unary operation, } \diamond \in \{\texttt{-}, \texttt{\sim}\}\textit{)} \\
 & | & \mathit{expr} \circ \mathit{expr} & \textit{(binary operation, } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&}, \texttt{|}, \texttt{\^{}}, \texttt{>>}, \texttt{<<}\}\textit{)} \\
\\
\mathit{stat} & ::= & V = \mathit{expr} & \textit{(assignment)} \\
 & | & \mathit{assert}(\mathit{expr}) & \textit{(assertion)}
\end{array}
$$

Figure 4: Fragment of integer C syntax.

$$
\begin{array}{l}
\tau : \mathit{expr} \to \mathit{int\text{-}type} \\
\quad \tau(n) \in \mathit{int\text{-}type} \quad \textit{(given)} \qquad \tau(V) \in \mathit{int\text{-}type} \quad \textit{(given)} \\
\quad \tau(\diamond\, e) \stackrel{\text{def}}{=} \mathit{promote}(\tau(e)) \qquad \tau(\texttt{(}t\texttt{)}\ e) \stackrel{\text{def}}{=} t \\
\quad \tau(e_1 \circ e_2) \stackrel{\text{def}}{=} \begin{cases} \mathit{lub}(\mathit{promote}(\tau(e_1)), \mathit{promote}(\tau(e_2))) & \text{if } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&}, \texttt{|}, \texttt{\^{}}\} \\ \mathit{promote}(\tau(e_1)) & \text{if } \circ \in \{\texttt{<<}, \texttt{>>}\} \end{cases}
\end{array}
$$

where:

$$
\mathit{promote}(t) \stackrel{\text{def}}{=} \begin{cases} \texttt{int} & \text{if } \mathit{rank}(t) < \mathit{rank}(\texttt{int}) \wedge \mathit{range}(t) \subseteq \mathit{range}(\texttt{int}) \\ \texttt{unsigned} & \text{else if } \mathit{rank}(t) < \mathit{rank}(\texttt{int}) \wedge \mathit{range}(t) \subseteq \mathit{range}(\texttt{unsigned}) \\ \mathit{promote}(t') & \text{if } t \text{ has bitfield type } t'\ n\text{, based on } t' \\ t & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\mathit{lub}(t, t') \stackrel{\text{def}}{=} \\
\quad \text{when } \mathit{rank}(t) \geq \mathit{rank}(t')\text{:} \\
\qquad \begin{cases} t & \text{if } \mathit{signed}(t) = \mathit{signed}(t') \text{ or } \neg\mathit{signed}(t) \wedge \mathit{signed}(t') \\ & \text{or } \mathit{signed}(t) \wedge \neg\mathit{signed}(t') \wedge \mathit{range}(t') \subseteq \mathit{range}(t) \\ \texttt{unsigned}\ t & \text{if } \mathit{signed}(t) \wedge \neg\mathit{signed}(t') \wedge \mathit{range}(t') \not\subseteq \mathit{range}(t) \end{cases} \\
\quad \text{when } \mathit{rank}(t) < \mathit{rank}(t')\text{: } \mathit{lub}(t', t)
\end{array}
$$

$$
\begin{array}{llll}
\mathit{rank}(\texttt{char}) \stackrel{\text{def}}{=} 1 & \mathit{rank}(\texttt{short}) \stackrel{\text{def}}{=} 2 & \mathit{rank}(\texttt{int}) \stackrel{\text{def}}{=} 3 & \mathit{rank}(\texttt{long}) \stackrel{\text{def}}{=} 4 \\
\mathit{rank}(\texttt{long long}) \stackrel{\text{def}}{=} 5 & \mathit{rank}(\texttt{signed}\ t) \stackrel{\text{def}}{=} \mathit{rank}(\texttt{unsigned}\ t) \stackrel{\text{def}}{=} \mathit{rank}(t) & &
\end{array}
$$

Figure 5: Typing of integer expressions.

explicit in expressions by first typing sub-expressions and then inserting casts. These steps are performed in a front-end and generally not discussed, but we present them to highlight of few subtle points.

Typing. The type $\tau(e)$ of an expression e is inferred as in Fig. 5 based on the given type of variables and constants. Firstly, a promotion rule (*promote*) states that values of type t smaller than `int` (where the notion of "smaller" is defined by the *rank* function) are promoted to `int`, if `int` can represent all the values in type t, and to `unsigned` otherwise. Values of bitfield type $t\ n$ are promoted as their corresponding base type t. Secondly, for binary operators, the type of the result is inferred from that of both arguments (*lub*). Integer promotion causes values with the same binary representation but different types to behave differently. For instance,

$$
\begin{array}{lcl}
(\!| V |\!) & \stackrel{\text{def}}{=} & V \\
(\!| n |\!) & \stackrel{\text{def}}{=} & (\tau(n))\ n \\
(\!| (t)\ e |\!) & \stackrel{\text{def}}{=} & (t)\ (\!| e |\!) \\
(\!| \diamond\ e |\!) & \stackrel{\text{def}}{=} & \text{let } t = \tau(\diamond e) \text{ in } (t)\ (\diamond\ (t)\ (\!| e |\!)) \\
(\!| e_1 \circ e_2 |\!) & \stackrel{\text{def}}{=} & \text{if } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&}, \texttt{|}, \texttt{\^{}}\} \text{ then:} \\
 & & \quad \text{let } t = \tau(e_1 \circ e_2) \text{ in } (t)\ ((t)\ (\!| e_1 |\!)\ \circ\ (t)\ (\!| e_2 |\!)) \\
 & & \text{if } \circ \in \{\texttt{<<}, \texttt{>>}\} \text{ then:} \\
 & & \quad \text{let } t = \tau(e_1) \text{ in } (t)\ ((t)\ (\!| e_1 |\!)\ \circ\ (\texttt{unsigned}\ 5)\ (\!| e_2 |\!)) \\
(\!| V = e |\!) & \stackrel{\text{def}}{=} & V = (\tau(V))\ (\!| e |\!) \\
(\!| \mathit{assert}(e) |\!) & \stackrel{\text{def}}{=} & \mathit{assert}((\!| e |\!))
\end{array}
$$

Figure 6: Insertion of implicit casts.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{[\![\mathit{expr}]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{Z})} \\
[\![V]\!]\rho & \stackrel{\text{def}}{=} & \{\, \rho(V) \,\} \qquad\qquad [\![n]\!]\rho \stackrel{\text{def}}{=} \{\, n \,\} \\
[\![(t)\ e]\!]\rho & \stackrel{\text{def}}{=} & \{\, \mathit{wrap}(v, \mathit{range}(t)) \mid v \in [\![e]\!]\rho \,\} \qquad [\![\diamond\ e]\!]\rho \stackrel{\text{def}}{=} \{\, \diamond v \mid v \in [\![e]\!]\rho \,\} \\
[\![e_1 \circ e_2]\!]\rho & \stackrel{\text{def}}{=} & \{\, v_1 \circ v_2 \mid v_1 \in [\![e_1]\!]\rho \wedge v_2 \in [\![e_2]\!]\rho \wedge (v_2 \neq 0 \vee \circ \notin \{\, \texttt{/}, \texttt{\%} \,\}) \,\} \\
\multicolumn{3}{l}{\text{where } \mathit{wrap}(v, [\ell, h]) \stackrel{\text{def}}{=} \min \{\, v' \mid v' \geq \ell \wedge \exists k \in \mathbb{Z} : v = v' + k(h - \ell + 1) \,\}} \\
\multicolumn{3}{l}{[\![\mathit{stat}]\!] : \mathcal{E} \to \mathcal{P}(\mathcal{E})} \\
[\![V = e]\!]\rho & \stackrel{\text{def}}{=} & \{\, \rho[V \mapsto v] \mid v \in [\![e]\!]\rho \,\} \\
[\![\mathit{assert}(e)]\!]\rho & \stackrel{\text{def}}{=} & \{\, \rho \mid \exists v \in [\![e]\!]\rho : v \neq 0 \,\}
\end{array}
$$

Figure 7: Concrete semantics.

`unsigned char a = 255` and `signed char b = -1` have the same representation, but `a >> 1 = 127` while `b >> 1 = -1`. Integer promotion is said to be "value preserving", as opposed to "representation preserving" [4]. This rule comforts us in our decision to focus on the integer value of variables instead of their binary representation.

Cast introduction. The translation of expressions, $(\!| \cdot |\!)$, is presented in Fig. 6. Firstly, before applying an operator, its arguments are converted to the type of the result. This can lead to wrap-around effects. For instance, in `(int)-1 + (unsigned)1`, the left argument is converted to `unsigned`, which gives $2^{32} - 1$ on a 32-bit architecture. The case of bit shift operators is special; as shifting by an amount exceeding the bit-size of the result is undefined in the C standard, we model instead the behavior of intel 32-bit hardware: the right argument is masked to keep only the lower 5 bits (abusing bitfield types). Secondly, casts are introduced to ensure that the value of the result lies within the range of its inferred type. Finally, before storing a value into a variable, it is converted to the type of the variable.

2.4 Operator Semantics

After translation, the semantics of expressions can be defined in terms of integers, without any reference to C types. The arithmetic operators (`+`,`-`,`*`,`/`,`%`) have their classic meaning in $\mathbb{Z}$.[2] To

[2]Note that `/` rounds towards zero and that $a \texttt{\%} b \stackrel{\text{def}}{=} a - (a/b)*b$.

define bit-wise operations (~, &, |, ^, <<, >>) on $\mathbb{Z}$, we first associate an (infinite) bit pattern to each integer in $\mathbb{Z}$. It is an element of the boolean algebra $B = (\{0,1\}^{\mathbb{N}}, \neg, \wedge, \vee)$ with pointwise negation $\neg$, logical and $\wedge$, and logical or $\vee$ operators. The pattern $p(x) \in B$ of an integer $x \in \mathbb{Z}$ is defined using an infinite two's complement representation:

$$p(x) \stackrel{\text{def}}{=} \begin{cases} p(x) = (b_i)_{i\in\mathbb{N}} & \text{where } b_i = \lfloor x/2^i \rfloor \mod 2, \text{ if } x \geq 0 \\ p(x) = (\neg b_i)_{i\in\mathbb{N}} & \text{where } (b_i)_{i\in\mathbb{N}} = p(-x-1), \text{ if } x < 0 \end{cases} \tag{1}$$

The elements in B are reminiscent of 2-adic integers, but we restrict ourselves to those representing regular integers. The function p is injective, and we note p^{-1} its inverse, which is only defined on sequences that are stable after a certain index ($\exists i : \forall j \geq i : b_j = b_i$). The bit-wise C operators are given a semantics in $\mathbb{Z}$, based on their natural semantics in B, as follows:

$$\begin{array}{rcl@{\qquad}rcl} \sim x & \stackrel{\text{def}}{=} & p^{-1}(\neg p(x)) = -x-1 & x \,\&\, y & \stackrel{\text{def}}{=} & p^{-1}(p(x) \wedge p(y)) \\ x \mid y & \stackrel{\text{def}}{=} & p^{-1}(p(x) \vee p(y)) & x \,\hat{}\, y & \stackrel{\text{def}}{=} & p^{-1}(p(x) \oplus p(y)) \\ x \ll y & \stackrel{\text{def}}{=} & \lfloor x \times 2^y \rfloor & x \gg y & \stackrel{\text{def}}{=} & \lfloor x \times 2^{-y} \rfloor \end{array} \tag{2}$$

where $\oplus$ is the exclusive or and $\lfloor \cdot \rfloor$ rounds towards $-\infty$. This semantics is compatible with that of existing arbitrary precision integer libraries [1, 22].

2.5 Expression Semantics

Given environments $\rho \in \mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{Z}$ associating integer values to variables, we can define the semantics $[\![\cdot]\!]$ of expressions and statements as shown in Fig. 7. The semantics of wrap-around is modeled by the *wrap* function. Our semantics is non-deterministic: expressions (resp. statements) return a (possibly empty) set of values (resp. environments). This is necessary to define the semantics of errors that halt the program (e.g., division by zero). Non-determinism is also useful to design analyses that must be sound with respect to several implementations at once. We could for instance relax our semantics and return a full range instead of the modular result in case of an overflow in signed arithmetics (as the result is undefined by the standard).

All it takes to adapt legacy domains to our new semantics is an abstraction of the *wrap* operator. A straightforward but coarse one would state that $\mathit{wrap}^\sharp(v, [\ell, h]) \stackrel{\text{def}}{=} \{v\}$ if $v \in [\ell, h]$, and $[\ell, h]$ otherwise (see also [26] for a more precise abstraction on polyhedra). In the following, we will introduce abstract domains specifically adapted to the wrap-around semantics.

2.6 Integer Interval Domain $\mathcal{D}_i^\sharp$

We recall very briefly the classic interval abstract domain [9]. It maps each variable to an interval of integers:

$$\mathcal{D}_i^\sharp \stackrel{\text{def}}{=} \{\, [\ell, h] \mid \ell, h \in \mathbb{Z} \cup \{\pm\infty\} \,\}$$

As it is a non-relational domain, the abstract semantics of expressions, and so, of assignments, can be defined by structural induction, replacing each operator $\circ$ on $\mathbb{Z}$ with an abstract version $\circ_i^\sharp$ on intervals. Abstract assertions are slightly more complicated and require backward operators; we refer the reader to [18, § 2.4.4] for details. We recall [9] that optimal abstract operators can be systematically designed with the help of a Galois connection (α, γ):

$$\begin{array}{rcl} [\ell_1, h_1] \circ_i^\sharp [\ell_2, h_2] & \stackrel{\text{def}}{=} & \alpha_i(\{\, v_1 \circ v_2 \mid v_1 \in \gamma_i([\ell_1, h_1]),\, v_2 \in \gamma_i([\ell_2, h_2]) \,\}) \\ \gamma_i([\ell, h]) & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{Z} \mid \ell \leq x \leq h \,\} \\ \alpha_i(X) & \stackrel{\text{def}}{=} & [\min X,\, \max X] \end{array}$$

$$
\begin{aligned}
&\texttt{-}^\sharp_m([\ell,h]+k\mathbb{Z}) \stackrel{\text{def}}{=} [-h,-\ell]+k\mathbb{Z}\\
&\texttt{\sim}^\sharp_m([\ell,h]+k\mathbb{Z}) \stackrel{\text{def}}{=} [-h-1,-\ell-1]+k\mathbb{Z}\\
&([\ell_1,h_1]+k_1\mathbb{Z}) \circ^\sharp_m ([\ell_2,h_2]+k_2\mathbb{Z}) \stackrel{\text{def}}{=}\\
&\qquad \begin{cases} ([\ell_1,h_1] \circ^\sharp_i [\ell_2,h_2]) + \gcd(k_1,k_2)\mathbb{Z} & \text{if } \circ \in \{\texttt{+},\texttt{-},\texttt{*},\cup,\triangledown\}\\ ([\ell_1,h_1] \circ^\sharp_i [\ell_2,h_2]) + 0\mathbb{Z} & \text{if } k_1=k_2=0,\ \circ \in \{\texttt{/},\texttt{\%},\texttt{\&},\texttt{|},\texttt{\^{}},\texttt{>>},\texttt{<<}\}\\ [-\infty,+\infty]+0\mathbb{Z} & \text{otherwise}\end{cases}\\
&\mathit{wrap}^\sharp_m([\ell,h]+k\mathbb{Z},[\ell',h']) \stackrel{\text{def}}{=}\\
&\qquad \text{let } k' = \gcd(k,\ h'-\ell'+1) \text{ in}\\
&\qquad \begin{cases} [\mathit{wrap}(\ell,[\ell',h']),\ \mathit{wrap}(h,[\ell',h'])]+0\mathbb{Z} & \text{if } (\ell'+k'\mathbb{Z})\cap[\ell+1,h]=\emptyset\\ [\ell,h]+k'\mathbb{Z} & \text{otherwise}\end{cases}
\end{aligned}
$$

where gcd is the greatest common divisor, and $\gcd(0,x)=\gcd(x,0)=x$.

Figure 8: Abstract operators in the modular interval domain $\mathcal{D}^\sharp_m$.

For instance, $[\ell_1,h_1] +^\sharp_i [\ell_2,h_2] \stackrel{\text{def}}{=} [\ell_1+\ell_2,h_1+h_2]$. Additionally, an abstraction $\cup^\sharp_i$ of the set union $\cup$, as well as a widening $\triangledown^\sharp_i$ [9] enforcing termination are required. We note that the optimal abstraction of *wrap* is:

$$
\begin{aligned}
&\mathit{wrap}^\sharp_i([\ell,h],[\ell',h']) =\\
&\quad \begin{cases} [\mathit{wrap}(\ell,[\ell',h']),\ \mathit{wrap}(h,[\ell',h'])] & \text{if } (\ell'+(h'-\ell'+1)\mathbb{Z})\cap[\ell+1,h]=\emptyset\\ [\ell',h'] & \text{otherwise}\end{cases}
\end{aligned}
$$

which returns the full interval $[\ell',h']$ when $[\ell,h]$ crosses a boundary in $\ell'+(h'-\ell'+1)\mathbb{Z}$. This is the case when $\{\, \mathit{wrap}(v,[\ell',h']) \mid v \in [\ell,h] \,\}$ is not convex.

Example. Consider the function `add1` in Fig. 1.a and assume that, in the abstract, the computed range for both arguments is $[-1,1]$. Then `(unsigned char)`$[-1,1]$ is abstracted as $[0,255]$. This is the best interval abstraction as, in the concrete, the value set is $\{0,1,255\}$. The following interval addition, which is computed in type `int` due to integer promotion, gives $[0,510]$. Once cast back to signed `char`, the interval result is the full range $[-128,127]$. This is much coarser than the concrete result, which is $[-2,2]$.

2.7 Modular Interval Domain $\mathcal{D}^\sharp_m$

We now propose a slight variation on the interval domain that corrects the precision loss in $\mathit{wrap}^\sharp_i$. It is defined as intervals with an extra modular component:

$$
\begin{aligned}
\mathcal{D}^\sharp_m &\stackrel{\text{def}}{=} \{\, [\ell,h]+k\mathbb{Z} \mid \ell,h \in \mathbb{Z}\cup\{\,\pm\infty\,\}, k\in\mathbb{N} \,\}\\
\gamma_m([\ell,h]+k\mathbb{Z}) &\stackrel{\text{def}}{=} \{\, x+ky \mid \ell \le x \le h,\ y\in\mathbb{Z} \,\}
\end{aligned}
$$

The domain was mentioned briefly in [14] but not defined formally. It is also similar to the interval congruences $\theta.[\ell,u]\langle m\rangle$ by Masdupuy [15], but with θ set to 1, which results in simpler abstract operations (abstract values with $\theta \neq 1$ are useful when modeling array accesses, as in [15], but not when modeling wrap-around).

Abstract operators $\circ^\sharp_m$ are defined in Fig. 8. There is no longer a best abstraction in general (e.g., $\{\,0,2,4\,\}$ could be abstracted as either $[0,4]+0\mathbb{Z}$ or $[0,0]+2\mathbb{Z}$ which are both minimal and

yet incomparable), which makes the design of operators with a guaranteed precision difficult. We designed $\texttt{~}^\sharp_m$, $\texttt{-}^\sharp_m$, $\texttt{+}^\sharp_m$, $\texttt{-}^\sharp_m$, and $\cup^\sharp_m$ based on optimal interval and simple congruence operators [11], using basic coset identities to infer modular information. The result may not be minimal (in the sense of minimizing $\gamma_m(x^\sharp \circ^\sharp_m y^\sharp)$) but suffices in practice. For other operators, we simply revert to classic intervals, discarding any modulo information.

We now focus our attention on $wrap^\sharp_m([\ell, h] + k\mathbb{Z}, [\ell', h'])$. Similarly to intervals, wrapping $[\ell, h]+k\mathbb{Z}$ results in a plain interval if no $[\ell, h]+ky$, $y \in \mathbb{Z}$ crosses a boundary in $\ell'+(h'-\ell'+1)\mathbb{Z}$, in which case the operation is exact. Otherwise, it returns the interval argument $[\ell, h]$ modulo both $h'-\ell'+1$ and k. This forgets that the result is bounded by $[\ell', h']$ but keeps an important information: the values ℓ and h. In practice, we maintain the range $[\ell', h']$ by performing a reduced product between $\mathcal{D}_m^\sharp$ and plain intervals $\mathcal{D}_i^\sharp$, which ensures that each operator (except the widening) is at least as precise as in an interval analysis.

Example. Consider again the function `add1` from Fig. 1.a, with abstract arguments $[-1, 1] + 0\mathbb{Z}$. Then, `e = (unsigned char)`[-1,1] is abstracted as $[-1, 1]+256\mathbb{Z}$. Thus, `e+e` gives $[-2, 2]+256\mathbb{Z}$. Finally, `(char)(e+e)` gives back the expected interval: $[-2, 2] + 0\mathbb{Z}$.

2.8 Bitfield Domain $\mathcal{D}_b^\sharp$

The interval domain $\mathcal{D}_i^\sharp$ is not very precise on bit-level operators. On the example of Fig. 2.b, (*range*(`int`) `& 0x801fffff`) `| 0x3fe00000` is abstracted as *range*(`int`), which does not capture the fact that some bits are fixed to 0 and others to 1. This issue can be solved by a simple domain that tracks the value of each bit independently. A similar domain was used by Monniaux when analyzing unsigned 32-bit integer computations in a device driver [23]. Our version, however, abstracts a $\mathbb{Z}-$based concrete semantics, making it independent from bit-size and signedness. The domain associates to each variable two integers, z and o, that represent the bit masks for bits that can be set respectively to 0 and to 1:

$$\begin{array}{rcl} \mathcal{D}_b^\sharp & \stackrel{\text{def}}{=} & \mathbb{Z} \times \mathbb{Z} \\ \gamma_b(z, o) & \stackrel{\text{def}}{=} & \{\, b \mid \forall i \geq 0 : (\neg p(b)_i) \wedge p(z)_i \text{ or } p(b)_i \wedge p(o)_i \,\} \\ \alpha_b(S) & \stackrel{\text{def}}{=} & (\vee \{\, \neg p(b) \mid b \in S \,\}, \vee \{\, p(b) \mid b \in S \,\}) \end{array}$$

where p is defined in (1). The optimal abstract operators can be derived through the Galois connection $\mathcal{P}(\mathbb{Z}) \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} \mathbb{Z} \times \mathbb{Z}$. We present the most interesting ones in Fig. 9. They use the bit-wise operators on $\mathbb{Z}$ defined in (2). For bit shifts, we only handle the case where the right argument represents a positive singleton (i.e., it has the form $n_b^\sharp$ for some constant $n \geq 0$). Wrapping around an unsigned interval $[0, 2^n - 1]$ is handled by masking high bits. Wrapping around a signed interval $[-2^n, 2^n - 1]$ additionaly performs a sign extension. Our domain has infinite increasing chains (e.g., $X_n^\sharp = (-1, 2^n - 1)$), and so, requires a widening: $\triangledown_b^\sharp$ will set all the bits to 0 (resp. 1) if the mask for bits at 0 (resp. at 1) is not stable.

Efficiency. The three domains $\mathcal{D}_i^\sharp$, $\mathcal{D}_m^\sharp$, $\mathcal{D}_b^\sharp$ are non-relational, and so, very efficient. Using functional maps, even joins and widenings can be implemented with a sub-linear cost in practice [7, §III.H.1]. Each abstract operation costs only a few integer operations. Moreover, the values encountered during an analysis generally fit in a machine word. Our analyzer uses an arbitrary precision library able to exploit this fact to improve the performance [22].

$$
\begin{array}{rcl}
n_b^\sharp & \stackrel{\text{def}}{=} & (\sim n, n) \qquad (n \in \mathbb{Z}) \\
\sim_b^\sharp(z, o) & \stackrel{\text{def}}{=} & (o, z) \\
(z_1, o_1) \,\&_b^\sharp\, (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \mid z_2,\ o_1 \,\&\, o_2) \\
(z_1, o_1) \mid_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \,\&\, z_2,\ o_1 \mid o_2) \\
(z_1, o_1) \,\hat{}_b^\sharp\, (z_2, o_2) & \stackrel{\text{def}}{=} & ((z_1 \,\&\, z_2) \mid (o_1 \,\&\, o_2),\ (z_1 \,\&\, o_2) \mid (o_1 \,\&\, z_2)) \\
(z_1, o_1) \ll_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & ((z_1 \ll n) \mid ((1 \ll n) - 1),\ o_1 \ll n),\ \text{when } \exists n \geq 0 : (z_2, o_2) = n_b^\sharp \\
(z_1, o_1) \gg_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \gg n,\ o_1 \gg n),\ \text{when } \exists n \geq 0 : (z_2, o_2) = n_b^\sharp \\
(z_1, o_1) \cup_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \mid z_2,\ o_1 \mid o_2) \\
(z_1, o_1) \triangledown_b^\sharp (z_2, o_2) & \stackrel{\text{def}}{=} & (z_1 \triangledown z_2,\ o_1 \triangledown o_2) \text{ with } x \triangledown y \stackrel{\text{def}}{=} \text{ if } x = x \mid y \text{ then } x \text{ else } -1 \\
\mathit{wrap}_b^\sharp((z, o), [0, 2^n - 1]) & \stackrel{\text{def}}{=} & (z \mid (-2^n),\ o \,\&\, (2^n - 1)) \\
\mathit{wrap}_b^\sharp((z, o), [-2^n, 2^n - 1]) & \stackrel{\text{def}}{=} & ((z \,\&\, (2^n - 1)) \mid (-2^n z_n),\ (o \,\&\, (2^n - 1)) \mid (-2^n o_n))
\end{array}
$$

Figure 9: Abstract operators in the bitfield domain $\mathcal{D}_b^\sharp$. See also (2).

$$
\begin{array}{l}
\mathit{synt} : (\mathcal{L} \times \mathcal{P}(\mathcal{L})) \to \mathit{expr} \\
\mathit{synt}((V, o, t), C) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
c & \text{if } c = (V, o, t) \in C \\
\texttt{(}t\texttt{)}\, c & \text{if } c = (V, o, t') \in C \wedge t, t' \in \mathit{int\text{-}type} \wedge \mathit{bitsize}(t) = \mathit{bitsize}(t') \\
\mathit{hi\text{-}word\text{-}of\text{-}dbl}(c) & \text{if } c = (V, o, t') \in C \wedge t \in \mathit{int\text{-}type} \wedge t' = \texttt{double} \wedge \mathit{bitsize}(t) = 32 \\
\mathit{dbl\text{-}of\text{-}word}(c_1, c_2) & \text{if } c_1 = (V, o, t') \in C \wedge c_2 = (V, o + 4, t') \in C \wedge t = \texttt{double} \wedge \\
 & \quad t' \in \mathit{int\text{-}type} \wedge \mathit{bitsize}(t') = 32 \\
\mathit{range}(t) & \text{otherwise}
\end{cases}
\end{array}
$$

Figure 10: Cell synthesis to handle physical casts. A big endian architecture is assumed.

3 Memory Abstract Domain

A prerequisite to analyze the programs in Fig. 2 is to detect and handle physical casts, that is, the re-interpretation of a portion of memory as representing an object of different type through the use of union types (Fig. 2.a) or pointer casts (Fig. 2.b).

In this section, we are no longer restricted to integers and consider arbitrary C expressions and types. Nevertheless our first step is to reduce expressions to the following simplified grammar:

$$
\begin{array}{lcl}
\mathit{expr} & ::= & n \mid \&V \mid \diamond\ \mathit{expr} \mid \mathit{expr} \circ \mathit{expr} \mid \texttt{(}\mathit{scalar\text{-}type}\texttt{)}\ \mathit{expr} \mid \texttt{*}_{\mathit{scalar\text{-}type}}\ \mathit{expr} \\
\mathit{stat} & ::= & (\texttt{*}_{\mathit{scalar\text{-}type}}\ \mathit{expr}) = \mathit{expr} \mid \mathit{assert}(\mathit{expr}) \\
\mathit{scalar\text{-}type} & ::= & \mathit{int\text{-}type} \mid \texttt{float} \mid \texttt{double} \mid \texttt{char *}
\end{array}
$$

Memory accesses (including field and array accesses, variable reads and modifications) are through typed dereferences $\texttt{*}_t\ e$, and pointer arithmetics is reduced to arithmetics on byte-based memory addresses. The translation to such expressions can be performed statically by a front-end. For instance, in Fig. 2.a, the statement `y.i[1] = i ^ x.i[1]` is translated into $\texttt{*}_{\texttt{int}}\ (\texttt{\&y} + 4) = (\texttt{*}_{\texttt{int}}\ \texttt{\&i})\ \texttt{\^{}}\ (\texttt{*}_{\texttt{int}}\ (\texttt{\&x} + 4))$. We do not detail this translation further.

A low-level semantics would model memory states as maps in $\{\,(V, i) \in \mathcal{V} \times \mathbb{N} \mid i < \mathit{bitsize}(\tau(V))\,\} \to \{\,0, 1\,\}$ from bit positions to bit values. We consider instead a slightly higher-level concrete semantics: the memory is a collection of cells in $\mathcal{L} \stackrel{\text{def}}{=} \{\,(V, o, t) \in$

$\mathcal{V} \times \mathbb{N} \times$ *scalar-type* $\mid o + \textit{sizeof}(t) \leq \textit{sizeof}(V)\}$, where V is the variable the cell lives in, o is its offset in bytes from the start of V, and t is the cell type (integer, float, or pointer). A concrete environment ρ is a partial function from cells in $\mathcal{L}$ to values in $\mathbb{V}$, where $\mathbb{V}$ contains all the integer, float, and pointer values. Pointer values are represented as pairs composed of a variable name $V \in \mathcal{V}$ and a byte offset $o \in \mathbb{N}$ from the start of the variable, and written as $\&V + o$. Cells are added on-demand when writing into memory: an assignment $*_t\ e = e'$ creates a cell (V, o, t) for each pointer $\&V + o$ pointed to by e. Reading a cell (V, o, t) from a memory with cell set $C \subseteq \mathcal{L}$ returns $\rho(V, o, t)$ if $(V, o, t) \in C$. If $(V, o, t) \notin C$, we try to synthesize its value. An example synthesis function, *synt*, is presented in Fig. 10: it returns an expression over-approximating the value of a cell (V, o, t) using only cells in C. Firstly, if t is an integer type and there is an integer cell $(V, o, t') \in C$ with the same size but $t \neq t'$, its value is converted to t — i.e., a physical cast `*((int*)&V)` is treated as a regular cast `(int)V`. Secondly, selected bit-level manipulations of floats are detected and translated into expressions using two new operators: *hi-word-of-dbl* and *dbl-of-word*, that denote, respectively, extracting the most significant 32-bit integer word of a 64-bit double and building a 64-bit double from two words (this semantics is formalized in Sec. 4.1). Thirdly, if no synthesis can occur, the whole range of t is returned.

This concrete semantics is then abstracted in a straightforward way: a specific memory domain maintains a set $C \subseteq \mathcal{L}$ of current cells and delegates the abstraction of $\mathcal{P}(C \to \mathbb{V})$ to an underlying domain. Pointer values are abstracted by maintaining an explicit set of pointed-to variables $C \to \mathcal{P}(\mathcal{V})$ and delegating the abstraction of offsets to a numeric domain. Given an expression and an abstract environment, the memory domain resolves pointer dereferences, synthesizes cell values, and translates dynamically expressions into simple numeric expressions, similar to Fig. 4 extended with the operators introduced during synthesis. Such expressions can then be handled directly by numeric domains, as shown in Sec. 4.2.

This memory domain was introduced in [19] and we will not discuss it further. Note however that the synthesis was limited to integer cells in [19], while we extend it here to express some bit-level float manipulations.[3]

Example. In Fig. 2.a, writing into `x.i[0]` and `x.i[1]` creates the cells $c_1 = (\texttt{x}, 0, \texttt{int})$ and $c_2 = (\texttt{x}, 4, \texttt{int})$. Reading back `x.d` amounts to evaluating the expression *dbl-of-word*(c_1, c_2). In Fig. 2.b, the expression `(*p & 0x7fe00000) >> 20` is translated into `(`*hi-word-of-dbl*(c) `& 0x7fe00000) >> 20`, where the cell $c = (\texttt{d}, 0, \texttt{double})$ represents the variable `d`.

4 Floating-Point Abstract Domains

4.1 Concrete Bit-Level Floating-Point Semantics

We now consider the analysis of programs manipulating floating-point numbers. Due to the limited precision of computers, float arithmetics exhibits rounding errors. For many purposes, it is sufficient to model floats as reals, with rounding abstracted as a non-deterministic choice in an error interval. This permits the use of the classic abstract domains defined on rationals or reals (intervals, but also relational domains [17]) to model floating-point computations. However, this is not sufficient when the bit representation of numbers is exploited, as in Fig. 2.

We introduce a bit-level semantics based on the ubiquitous IEEE 754-1985 standard [13]. We focus here on double-precision numbers, which occupy 64 bits: a 1-bit sign s, a 11-bit

[3] Our analyzer (Sec. 5) extends this further to synthesize and decompose integers, bitfields, and 32-bit floats (not presented here for the sake of conciseness).

$$
\begin{array}{l}
\mathit{dbl} : \{0,1\}^{64} \to \mathbb{V} \\
\mathit{dbl}(s, e_{10}, \ldots e_0, m_0 \ldots, m_{51}) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
(-1)^s \times (1 + \sum_{i=0}^{51} 2^{-i-1} m_i) \times 2^{(\sum_{i=0}^{10} 2^i e_i - 1023)} & \text{if } \sum_{i=0}^{10} 2^i e_i \notin \{\, 0, 2047 \,\} \\
(-1)^s \times (\sum_{i=0}^{51} 2^{-i-1} m_i) \times 2^{-1022} & \text{if } \forall i : e_i = 0 \\
(-1)^s \times \infty & \text{if } \forall i : e_i = 1 \wedge \forall j : m_j = 0 \\
\mathit{NaN} & \text{if } \forall i : e_i = 1 \wedge \exists j : m_j = 1
\end{cases} \\
[\![\, \mathit{dbl\text{-}of\text{-}word}(e_1, e_2) \,]\!]\rho \stackrel{\text{def}}{=} \{\, \mathit{dbl}(b^1_{31}, \ldots, b^1_0, b^2_{31}, \ldots, b^2_0) \mid \\
\qquad\qquad \forall j \in \{1,2\} : \sum_{i=0}^{31} 2^i b^j_i \in [\![\, \mathit{wrap}(e_j, [0, 2^{32} - 1]) \,]\!]\rho \,\} \\
[\![\, \mathit{hi\text{-}word\text{-}of\text{-}dbl}(e) \,]\!]\rho \stackrel{\text{def}}{=} \{\, \sum_{i=0}^{31} 2^i b_{i+32} \mid \exists b_0, \ldots, b_{31} : \mathit{dbl}(b_{63}, \ldots, b_0) \in [\![\, e \,]\!]\rho \,\}
\end{array}
$$

Figure 11: Bit-level concrete semantics of floating-point numbers, extending Fig. 7.

exponent e_0 to e_{10}, and a 52-bit mantissa m_0 to m_{51}. The mapping from bit values to float values is described by the function *dbl* in Fig. 11. In addition to normalized numbers (of the form $\pm 1.m_0 m_1 \cdots \times 2^x$), the standard allows denormalized (i.e., small) numbers, signed infinities $\pm\infty$, and NaNs (*Not a Numbers*, standing for error codes). This representation gives a concrete semantics to the operators *dbl-of-word* and *hi-word-of-dbl* introduced by cell synthesis. As in the case of integers, legacy abstract domains can be adapted to our new semantics by defining an abstraction for our new operators. Straightforward ones would state that $\mathit{dbl\text{-}of\text{-}word}^\sharp(e_1, e_2) = \mathit{range}(\texttt{double})$ and $\mathit{hi\text{-}word\text{-}of\text{-}dbl}^\sharp(e) = \mathit{range}(\texttt{unsigned})$, but we propose more precise ones below.

4.2 Predicate Domain on Binary Floating-Point Numbers $\mathcal{D}_p^\sharp$

The programs in Fig. 2 are idiomatic. It is difficult to envision a general domain that can reason precisely about arbitrary binary floating-point manipulations. Instead, we propose a lightweight and extensible technique based on pattern matching of selected expression fragments. However, matching each expression independently is not sufficient: it provides only a local view that cannot model computations spread across several statements precisely enough. We need to infer and propagate semantic properties to gain in precision.

To analyze Fig. 2.a, we use a domain $\mathcal{D}_p^\sharp$ of predicates of the form $V = e$, where $V \in \mathcal{V}$ and e is an expression chosen from a fixed list $\mathbb{P}$ with a parameter $W \in \mathcal{V}$. At most one predicate is associated to each V, so, an abstract element is a map from $\mathcal{V}$ to $\mathbb{P} \cup \{\, \top \,\}$, where $\top$ denotes the absence of a predicate:

$$
\begin{array}{l}
\mathcal{D}_p^\sharp \stackrel{\text{def}}{=} \mathcal{V} \to (\mathbb{P} \cup \{\, \top \,\}) \quad \text{where:} \\
\quad \mathbb{P} \quad ::= \quad W \,\hat{}\, \texttt{0x80000000} \qquad\qquad (W \in \mathcal{V}) \\
\qquad\quad\;\; \mid \quad \mathit{dbl\text{-}of\text{-}word}(\texttt{0x43300000}, W) \quad (W \in \mathcal{V}) \\
\gamma_p(X_p^\sharp) \stackrel{\text{def}}{=} \{\, \rho \in \mathcal{E} \mid \forall V \in \mathcal{V} : X_p^\sharp(V) = \top \vee \rho(V) \in [\![\, X_p^\sharp(V) \,]\!]\rho \,\}
\end{array}
\tag{3}
$$

The concretization is the set of environments that satisfy all the predicates, and there is no Galois connection. Figure 12 presents a few example transfer functions. Assignments and tests operate on a pair of abstractions: a predicate $X_p^\sharp$ and an interval $X_i^\sharp$. Sub-expressions from e are combined by *combine* with predicates from $X_p^\sharp$ to form idioms. The assignment then removes (in $Y_p^\sharp$) the predicates about the modified variable ($\mathit{var}(p)$ is the set of variables appearing in p) and tries to infer a new predicate. The matching algorithm is not purely syntactic: it uses a semantic interval information from $X_i^\sharp$ (e.g., to evaluate sub-expressions). Dually, successfully

$$
\begin{array}{l}
[\![V = e]\!]_p^\sharp(X_p^\sharp, X_i^\sharp) \stackrel{\text{def}}{=} \\
\quad \text{let } Y_i^\sharp = [\![V = \mathit{combine}(e, X_p^\sharp, X_i^\sharp)]\!]_i^\sharp \; X_i^\sharp \text{ in} \\
\quad \text{let } Y_p^\sharp = \lambda W : \text{if } W = V \text{ or } V \in \mathit{var}(X_p^\sharp(W)) \text{ then } \top \text{ else } X_p^\sharp(W) \text{ in} \\
\quad \text{if } e = W \texttt{ \^{} } e_1 \wedge [\![e_1]\!]_i^\sharp X_i^\sharp \in \{[2^{31}, 2^{31}], [-2^{31}, -2^{31}]\} \\
\qquad \text{then } (Y_p^\sharp[V \mapsto W \texttt{ \^{} } \texttt{0x80000000}], Y_i^\sharp) \\
\quad \text{else if } e = \mathit{dbl\text{-}of\text{-}word}(e_1, W) \wedge [\![e_1]\!]_i^\sharp X_i^\sharp = [1127219200, 1127219200] \\
\qquad \text{then } (Y_p^\sharp[V \mapsto \mathit{dbl\text{-}of\text{-}word}(\texttt{0x43300000}, W)], Y_i^\sharp) \\
\quad \text{otherwise } (Y_p^\sharp, Y_i^\sharp)
\end{array}
$$

$$[\![\mathit{assert}(e)]\!]_p^\sharp(X_p^\sharp, X_i^\sharp) \stackrel{\text{def}}{=} (X_p^\sharp, [\![\mathit{assert}(\mathit{combine}(e, X_p^\sharp, X_i^\sharp))]\!]_i^\sharp X_i^\sharp)$$

$$X_p^\sharp \cup^\sharp Y_p^\sharp \stackrel{\text{def}}{=} \lambda V : \text{if } X_p^\sharp(V) = Y_p^\sharp(V) \text{ then } X_p^\sharp(V) \text{ else } \top$$

$\mathit{combine}(e, X_p^\sharp, X_i^\sharp)$ replaces sub-expressions $V_1 \texttt{ - } V_2$ in e with $\texttt{(double)}I$ when:
$\exists V_1', V_2' : \forall j \in \{1, 2\} : X_p^\sharp(V_j) = \mathit{dbl\text{-}of\text{-}word}(\texttt{0x43300000}, V_j') \wedge$
$X_p^\sharp(V_1') = I \texttt{ \^{} } \texttt{0x80000000} \wedge X_i^\sharp(V_2') = [-2^{31}, -2^{31}]$

Figure 12: Abstract operator examples in the predicate domain $\mathcal{D}_p^\sharp$.

```
double frexp(double d, int* e) {
  int x = 0;
  double r = 1, dd = fabs(d);
  if (dd >= 1) { while ★ (dd > r) { x++; ♦ r *= 2; } }
  else { while (dd < r) { x--; r /= 2; } }
  *e = x;
  return d/r;
}
```

Figure 13: Floating-point decomposition into mantissa and exponent.

matched idioms refine the interval information. The join on $\mathcal{D}_p^\sharp$ removes the predicates that are not identical. As $\mathcal{D}_p^\sharp$ is flat, it has no need for a widening. Similarly to non-relational domains, the cost of each operation is sub-linear when implemented with functional maps [7, §III.H.1].

To stay concise, we only present the transfer functions sufficient to analyze Fig. 2.a. It is easy to enrich $\mathbb{P}$ with new predicates and extend the pattern matching and the propagation rules to accommodate other idioms. For instance, Fig. 2.b can be analyzed by adding the predicate $V = \mathit{hi\text{-}word\text{-}of\text{-}dbl}(W)$ and using information gathered by the bitfield domain $\mathcal{D}_b^\sharp$ during pattern matching.

4.3 Exponent Domain $\mathcal{D}_e^\sharp$

As last example, we consider the program in Fig. 13 that decomposes a float into its exponent and mantissa. Although it is possible to extract exponents using bit manipulations, as in Fig. 2.b, this function uses another method which illustrates the analysis of loops: when $|\texttt{d}| \geq 1$ it computes $\texttt{r} = 2^{\texttt{x}}$ iteratively, incrementing `x` until $\texttt{r} \geq |\texttt{d}|$. This example is, as those in Figs. 1–2, inspired from actual industrial codes and out of scope of existing abstract domains.

To provide precise bounds for the returned value, a first step is to bound `r`. We focus on

	size	with domains		w/o domains		pre-processed	
	(KLoc)	time	alarms	time	alarms	time	alarms
	154	10h44	22	10h04	22	11h38	22
	186	7h44	10	7h22	10	7h16	10
	103	54mn	**2**	44mn	451	46mn	6
(a)	493	7h34	**3**	15h27	1,833	8h40	195
	661	14h46	**2**	16h23	3,419	13h32	253
	616	22h03	**5**	26h46	5,350	20h45	300
	859	65h08	**110**	41h03	5,968	59h55	316
	2,428	48h28	**1**	44h06	3,822	44h57	674
(b)	113	25mn	30	20mn	30	17mn	30
(c)	79	3h22	7	3h09	7	3h27	7
(d)	102	46mn	64	59mn	64	59mn	64
	1,727	30h18	**2,133**	26h15	2,388	28h46	2,190

Figure 14: Analysis performance on industrial benchmarks.

the case where $|\mathtt{d}| \geq 1$. To prove that $\mathtt{r}$ and $\mathtt{d}/\mathtt{r}$ are bounded, it is necessary to infer at ★ the loop invariant $\mathtt{r} \leq 2\mathtt{d}$ and use the loop exit condition $\mathtt{d} \leq \mathtt{r}$. Several solutions exist to infer this invariant (such as using the polyhedra domain). We advocate the use of a variation $\mathcal{D}_e^\sharp$ of the, more efficient, zone domain [16]. While the original domain infers invariants of the form $V - W \in [\ell, h]$, we infer invariants of the form $V/W \in [\ell, h]$:

$$\begin{array}{rcl} \mathcal{D}_e^\sharp & \stackrel{\text{def}}{=} & (\mathcal{V} \times \mathcal{V}) \to \{\, [\ell, h] \mid \ell, h \in \mathbb{R} \cup \{\, \pm\infty \,\} \,\} \\ \gamma_e(X_e^\sharp) & \stackrel{\text{def}}{=} & \{\, \rho \in \mathcal{E} \mid \forall V, W \in \mathcal{V} : \rho(W) = 0 \vee \rho(V)/\rho(W) \in X_e^\sharp(V, W) \,\} \\ \alpha_e(X) & \stackrel{\text{def}}{=} & \lambda(V, W) : [\min \{\, \rho(V)/\rho(W) \mid \rho \in X \,\}, \max \{\, \rho(V)/\rho(W) \mid \rho \in X \,\}] \end{array}$$

The domain is constructed by applying a straightforward log transformation to [16]. A near linear cost can be achieved by using packing techniques [7, §III.H.5].

Now that $\mathtt{r}$ is bounded, in order to prove that $\mathtt{x}$ is also bounded, it is sufficient to infer a relationship between $\mathtt{x}$ and $\mathtt{r}$, i.e., $\mathtt{r} = 2^{\mathtt{x}}$ at ★, and $\mathtt{r} = 2^{\mathtt{x}-1}$ at ♦. This is possible, for instance, by using the predicate domain $\mathcal{D}_p^\sharp$ from Sec. 4.2 enriched with a new predicate parameterized by a variable W and an integer i:

$$\mathbb{P}' ::= \mathbb{P} \mid 2^{W+i} \qquad (W \in \mathcal{V},\, i \in \mathbb{Z})$$

To stay concise, we do not describe the adapted transfer functions.

5 Experimental Results

All the domains we presented have been incorporated into the Astrée static analyzer for synchronous embedded C programs [3, 7] and its extension AstréeA for multi-threaded programs [21]. Both check for arithmetic and memory errors (integer and float overflows and invalid operations, array overflows, invalid pointer dereferences). Thanks to a modular design based on a partially reduced product of communicating domains, new domains can be easily added.

Figure 14 presents the running-time and number of alarms when analyzing large (up to 2.5 MLoc) C applications from the aeronautic industry. Figure 14.a corresponds to a family

of control-command software; each one consists in a single very large reactive loop generated from a graphical specification language (similar to Lustre) and features much floating-point computations (integrations, digital filters, etc.). Figure 14.b is a software to perform hardware checks; it features many loops and is mainly integer-based. Figures 14.c–d are embedded communication software that manipulate strings and buffers; they feature much physical casts to pack, transmit, and unpack typed and untyped messages, as well as some amount of boolean and numeric control. Additionally, the applications in Fig. 14.d are multi-threaded. More details on the analyzed applications are available in [7, III.M] and [21].

In all the tests, the default domains are enabled (intervals, octagons, binary decision trees, etc., we refer the reader to [7] for an exhaustive list). The first and second columns show, respectively, the result with and without our new domains. In many cases (shown in boldface), our domains strictly improve the precision. Moreover, the improved analysis is between twice slower and twice faster. This variation can be explained as follows: adding new domains incurs a cost per abstract operation, but improving the precision may decrease or increase the number of loop iterations to reach a fixpoint. In the third column, our new domains are disabled but the code is pre-processed with ad-hoc syntactic scripts to (partially) remove their need (e.g., replacing the `cast` function in Fig. 1.b with a C cast), which is possibly unsound and incomplete (hence the remaining alarms). Comparing the first and last columns shows that being sound does not cause a significant loss of efficiency.

6 Conclusion

We presented several abstract domains to analyze C codes exploiting the binary representation of integers and floats. They are based on a concrete semantics that allows reasoning about these low-level implementation aspects in a sound way, while being compatible with legacy abstract domains. Each introduced domain focuses on a specific coding practice, so, they are not general. However, they are simple, fast to design and implement, efficient, and easy to extend. They have been included in the industrial-strength analyzer Astrée [3] to supplement existing domains and enable the precise analysis of codes exploiting the binary representation of numbers without resorting to unsound source pre-processing.

Future work includes generalizing our domains and developing additional domains specialized to code patterns we will encounter when analyzing new programs, with the hope of building a library of domains covering most analysis needs.

References

[1] GMP: The GNU multiple precision arithmetic library. `http://gmplib.org/`.

[2] How do I get rid of red OVFL outside PolySpace for Model Link TL? `http://www.mathworks.fr/support/solutions/en/data/1-5D3ONT/`.

[3] AbsInt, Angewandte Informatik. Astrée run-time error analyzer. `http://www.absint.com/astree`.

[4] ANSI Technical Committee and ISO/IEC JTC 1 Working Group. Rationale for international standard, Programming languages, C. Technical Report 897, rev. 2, ANSI, ISO/IEC, Oct. 1999.

[5] ANSI Technical Committee and ISO/IEC JTC 1 Working Group. C standard. Technical Report 9899:1999(E), ANSI, ISO & IEC, 2007.

[6] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI'01)*, pages 203–213. ACM, 2001.

[7] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385, pages 1–38. AIAA, Apr. 2010.

[8] J. Brauer and A. King. Transfer function synthesis without quantifier elimination. In *Proc. of the 20th European Symp. on Prog. (ESOP'11)*, volume 6602 of *LNCS*, pages 97–115. Springer, Mar. 2011.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISOP'76*, pages 106–130. Dunod, Paris, France, 1976.

[10] dSpace. TargetLink code generator. `http://www.dspaceinc.com`.

[11] P. Granger. Static analysis of arithmetic congruences. *Int. Journal of Computer Mathematics*, 30:165–199, 1989.

[12] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proc. of the Int. Joint Conf. on Theory and Practice of Soft. Development (TAPSOFT'91)*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

[13] IEEE Computer Society. Standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 745-1985, 1985.

[14] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the absence of runtime errors. In *Embedded Real Time Soft. and Syst. (ERTS^2 2010)*, pages 1–9, May 2010.

[15] F. Masdupuy. Semantic analysis of interval congruences. In *Proc. of the Int. Conf on Formal Methods in Prog. and Their Applications (FMPTA'93)*, volume 735 of *LNCS*, pages 142–155. Springer, 1993.

[16] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. of the 2d Symp. on Programs as Data Objects (PADO II)*, volume 2053 of *LNCS*, pages 155–172. Springer, May 2001.

[17] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. of the European Symp. on Prog. (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[18] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, Dec. 2004.

[19] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM Conf. on Lang., Compilers, and Tools for Embedded Syst. (LCTES'06)*, pages 54–63. ACM, June 2006.

[20] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[21] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Proc. of the 20th European Symp. on Prog. (ESOP'11)*, volume 6602 of *LNCS*, pages 398–418. Springer, Mar. 2011.

[22] A. Miné, X. Leroy, and P. Cuoq. ZArith: Arbitrary precision integers library for OCaml. `http://forge.ocamlcore.org/projects/zarith`.

[23] D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In *Proc. of the 7th ACM & IEEE Int. Conf. on Embedded Soft. (EMSOFT'07)*, pages 30–36. ACM, Sep. 2007.

[24] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *Proc. of the 14th European Symp. on Prog. (ESOP'05)*, volume 3444 of *LNCS*, pages 46–60. Springer, Apr. 2005.

[25] J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Proc. of the ACM Conf. on Lang., Compilers, and Tools for Embedded Syst. (LCTES'06)*, pages 34–43. ACM, June 2006.

[26] A. Simon and A. King. Taming the wrapping of integer arithmetic. In *Proc. of the 14th Int. Symp. on Static Analysis (SAS'07)*, volume 4634 of *LNCS*, pages 121–136. Springer, Aug. 2007.