Novel Automatic Solver for Program Analysis

Requirement Analysis

Novel Automatic Solver for Program Analysis

Confidentiality		Client
Defense	Industry	ANR
No	No	
Contractual		**Funding**
Yes		ANR
WorkPackage		
D1.1		

Title

Requirement Analysis

Author(s)
Claire Dross – Adacore
Johannes Kanig – Adacore
François Bobot – CEA

Revision	Number of pages
D1.1-v1	21

Abstract

This report gives a summary of the most important point where improvement of current solver is seeked by industrial partners

Keywords

requirement, non-linear arithmetic, floating-point, bitvector, arrays, memory-models, models, counter-example

Partners	Leader of the Workpackage
Adacore CEA	Adacore

Contents

Chapter 1

Introduction

1.1 SPARK

SPARK is both a programming language, defined as a very safe and unambiguous subset of the language Ada, and a tool, which enables the user to apply formal methods to SPARK programs [7]. One important component of these formal methods is the proof of properties of the program such as absence of runtime errors and proof of function contracts.

As a general purpose programming language, SPARK contains all common data types such as arrays, records, strings etc, but also the common numeric types such as integers and floating point numbers, and also allows bitwise operations on integer values.

The SPARK tools work by translating a SPARK program into a Why3 program[2]. Why3 then generates verification conditions (VCs - which are simply logical formulas), and is capable of providing these VCs in various textual formats suitable for many different provers.

The most useful provers in this context are currently provers of the SMT family [1]. These solvers can reason very well about the boolean structure of a formula, and are also very good in reasoning about certain operations, such as linear arithmetic on integers, bitwise operations, and array operations,

However, certain other operations are very badly supported by current SMT solvers: When nonlinear arithmetic operations, such as multiplication of two variables or division, or floating-point arithmetic is involved, we can expect that an SMT-solver will not provide good results.

As mentioned before, SMT solvers are very good at reasoning about bitwise operations and linear arithmetic. The combination framework of Nelson-Oppen makes sure that this also works when both bitwise operations and linear arithmetic appear in the same formula.

But programs which use bitwise operations rarely do only bitwise opera-

tions, and usually it is important to know the impact of an arithmetic operation on the bitpattern of the result, and conversely bitwise operations are often used to provide more efficient implementations for arithmetic operations. SMT solvers are usually very bad at bitwise and arithmetic reasoning *at the same time*, and also conversion between the bitwise and integer representation of an integer is very costly.

To summarize, the SPARK tools rely almost exclusively on SMT solvers to establish properties of SPARK programs, and AdaCore has identified the following shortcomings in SMT solvers:

- proof of properties using floating point arithmetic
- proof of properties using non-linear arithmetic (integer or floating point)
- proof of properties using bitwise arithmetic, without performance penalty for linear arithmetic

The next chapter will provide concrete examples of each problem, and other areas where there are shortcomings of current prover technology.

1.2 Frama-C

Frama-C is a platform for proving C programs. The Frama-C plugin WP, like SPARK followed by Why3, generates VCs and can send them to Alt-Ergo, Coq or other provers through Why3.

The problems identified in the previous section are also valid for The Frama-C plugin WP. However the memory model of C is more complicated than the one in SPARK, since in SPARK there is no aliasing (the possibility to access or modify some value by two different means), which requires the WP plugin to model the computer memory in the VCs. The memory is modeled by functional arrays, we access the value at address p in the memory state m by $\text{get}(p, m)$ and the memory state resulting from the modification of a memory state m at address p with value v is $\text{set}(m, p, v)$. This theory is classic in SMT-solvers however there are very few links between this theory and other theories (like arithmetic) which add expensive reasonings. However it could be avoidable like in Abstract Interpretation where array summary are used[3].

In addition to aliasing, C allows to access the same memory location in different ways. One program can write an integer and read only the first part as a char or read it as a floating point number. For this low-level C code we need an array theory that provide functions to write or read chuck of memory at the same time.

Finally, we need to be able to compose arrays. Indeed the specification of a C function indicates which part of the memory can be modified by a call to

this function. It allows functions that call it to compute which information are preserved by the function call. Currently the modeling of this part involve quantification which could be avoided with a richer array theory.

Frama-C WP try to overcome the limitation of current provers by applying a rewriting and simplifying phase before sending the VCs to the prover. So specific WP functions and operator can be simplified directly, something that provers can't do because they don't have these theories built in.

Chapter 2

Theories

2.1 Floating Point Arithmetic

Floating point numbers are now ubiquitous even in embedded computing, for any application where numeric computations with good precision are required. Ada provides 32bit and 64bit floating point numbers. Older versions of SPARK used real numbers to model floating point numbers, making the assumption that infinite precision is available (so were unsound). The current version of SPARK does not make this assumption anymore (so is sound for IEEE 754 floating point arithmetic), but reaches this goal with a heavy axiomatization, that corresponds to the one presented by Monniaux [5]:

- models floating point values as real
- inserts uninterpreted rounding operations on every arithmetic operation
- use axiom (3) from the paper to bound the error, based on the magnitude of operands

Although this axiomatization allows proving tight bounds on small codes, automatic provers are not very efficient using this axiomatization.

Before using provers, SPARK performs a simple analysis of bounds of expressions based on the types of variables, which allows to statically check many overflow and range checks without generating a VC. In many cases the SMT solvers would not be capable of proving the corresponding VC. But there are still many VCs left which are hard to prove automatically.

The challenge is to provide sound and efficient reasoning for floating-point operations in SPARK.

Currently, bounds on floating-point operations are obtained by axioms on the exact representation of smaller integers as floats, so conversions between

such integers and floats are value preserving. Float literals like 3.14 are represented as the sum of an integral part and a fractional part (as 3 + 0.14) so that the prover knows that it is between 3 and 4. This transformation is done by a special Why3 pass. This is an ad-hoc workaround for the current lack of support of precise floating point operations, but we'll also need some support of conversions between integers and floats, so that the prover can show that 3.0 = Float(3).

Below is a list of specifically prepared SPARK programs which concisely summarize the problems AdaCore has encountered in customer code using floating point numbers. These tests (and other tests which will follow during the project) will serve as a a benchmark for the other members to evaluate the progress of the project.

```
1  with Ada.Unchecked_Conversion;

3  package body Floating_Point with
     SPARK_Mode
5  is
      procedure Range_Add (X : Float_32; Res : out Float_32) is
7     begin
         pragma Assume (X in 10.0 .. 1000.0);
9        Res := X + 2.0;
         pragma Assert (Res >= 12.0);
11    end Range_Add;

13    procedure Range_Mult (X : Float_32; Res : out Float_32) is
      begin
15       pragma Assume (X in 5.0 .. 10.0);
         Res := X * 2.0 - 5.0;
17       pragma Assert (Res >= X);
      end Range_Mult;
19
      procedure Range_Add_Mult (X, Y, Z : Float_32;
21                              Res     : out Float_32) is
      begin
23       pragma Assume (X >= 0.0 and then X <= 180.0);
         pragma Assume (Y >= -180.0 and then Y <= 0.0);
25       pragma Assume (Z >= 0.0 and then Z <= 1.0);
         pragma Assume (X + Y >= 0.0);
27       Res := X + Y * Z;
         pragma Assert (Res >= 0.0 and then Res <= 360.0);
29    end Range_Add_Mult;

31    procedure Int_To_Float_Complex (X   : Unsigned_16;
                                      Y   : Float_32;
33                                    Res : out Float_32) is
         S_Max   : constant := 10.0;
35       S_MSB   : constant := S_Max * 2.0;
         S_Scale : constant := 2.0 ** 16 / S_MSB;
37    begin
         pragma Assume (Y in 0.25 .. 1.0);
39       Res := Float_32 (X) / S_Scale;
         if Res >= S_Max then
41          Res := Res - S_MSB;
```

```
      end if;
43       Res := Res / Y;
   end Int_To_Float_Complex;
45
   procedure Int_To_Float_Simple (X   : Unsigned_16;
47                                 Res : out Float_32) is
      L : constant := 7.3526e6;
49 begin
      pragma Assume (X /= 0);
51    pragma Assert (Float_32 (X) >= 0.9);
      Res := L / Float_32 (X);
53 end Int_To_Float_Simple;

55 function Float_To_Long_Float (X : Float) return Long_Float is
      Tmp : Long_Float;
57 begin
      pragma Assume (X >= Float'First and X <= Float'Last);
59    Tmp := Long_Float (X);
      pragma Assert
61          (Tmp >= Long_Float (Float'First) and
             Tmp <= Long_Float (Float'Last));
63    return Tmp;
   end Float_To_Long_Float;
65
   procedure Incr_By_Const (State : in out Float_32;
67                            X     : T)
   is
69 begin
      State := State + C;
71 end Incr_By_Const;

73 function Approximate_Inverse_Square_Root (X : Float)
      return Float
75 is
      function To_Float is new
77       Ada.Unchecked_Conversion (Source => Integer,
                                   Target => Float);
79    function To_Int is new
         Ada.Unchecked_Conversion (Source => Float,
81                                 Target => Integer);
      I          : Integer;
83    Y          : Float;
      X2         : constant Float := X * 0.5;
85    Threehalfs : constant Float := 1.5;
   begin
87    I := To_Int (X);
      I := 16#5F3759DF# - (I / 2);
89    Y := To_Float (I);
      Y := Y * (Threehalfs - (X2 * (Y * Y)));
91    --  Y := Y * (Threehalfs - (X2 * (Y * Y)));
      --  Second iteration can be enabled for more precision.
93    return Y;
   end Approximate_Inverse_Square_Root;
95
end Floating_Point;
```

floating_point.adb

2.2 Bitvector Arithmetic

Bitwise operations are commonly used in embedded code, for:

- low-level software communicating with components which are external to the program (e.g. the hardware), so that the exact bit pattern in memory which is used for the data, usually irrelevant and not observable in Ada, becomes relevant for the application.
- cryptographic code, where the efficiency of bitwise operations over classical arithmetic operations (e.g. right shift instead of division by a power of 2) is critical.

In Ada, there are 9 bitwise operations, which are only defined for modular integer types:

- bitwise logical operations: `and`, `or`, `xor` (see Ada RM 4.5.1), `not` (see Ada RM 4.5.6)
- operations on the bit representation of a number: left or right shift (right shift comes in 2 versions, plain and arithmetic), left or right rotation (see Ada RM B.2)

Currently, SPARK translates modular types using the Why3 theory of bitvectors. This theory is then translated directly into native bitvectors for provers supporting them (for example CVC4) and into mathematical integers with axioms for other provers (for example Alt-Ergo). Signed integer types, on the other hand, are always translated into mathematical integers.

The challenge is to provide efficient reasoning about bitwise operations on modular types while retaining the link with linear integer arithmetics so that equivalence between two computations, one using modular types, the other using signed types can be verified.

```
package body Bitwise with
  SPARK_Mode
is
   procedure Mask (X : Unsigned_32; Res : out Unsigned_8) is
   begin
      Res := Unsigned_8(((X and 16#FFFFFF00#) or 1)
                        and 16#000000FF#);
      pragma Assert (Res = 1);
   end Mask;

   procedure Mask_8bits (X : in Unsigned_32; Res : out Unsigned_32)
   is
      LMask_8 : constant Unsigned_8 := 2**8 - 1;
   begin
      Res := Unsigned_32(LMask_8);
```

```
16       pragma Assert ((X and Res) = (X and Unsigned_32(LMask_8)));
   end Mask_8bits;
18
   procedure Shift_Is_Div (X : Unsigned_32; Res : out Unsigned_32)
20 is
   begin
22    Res := Shift_Right (X, 1);
      pragma Assert (Res = X / 2);
24    Res := Shift_Right (Res, 2);
      pragma Assert (Res = X / 8);
26 end Shift_Is_Div;

28 procedure Swap (X, Y : in out Unsigned_32) is
      XX : constant Unsigned_32 := X;
30    YY : constant Unsigned_32 := Y;
   begin
32    X := X xor Y;
      Y := X xor Y;
34    X := X xor Y;
      pragma Assert (X = YY);
36    pragma Assert (Y = XX);
   end Swap;
38
   procedure Write16 (Val : in Unsigned_16;
40                    FstHalf, SndHalf : out Unsigned_8) is
   begin
42    FstHalf := Unsigned_8 (Val and 16#00FF#);
      SndHalf := Unsigned_8 (Shift_Right(Val, 8) and 16#00FF#);
44    pragma assert (Val =
                     (Unsigned_16(FstHalf) or
46                        Shift_Left(Unsigned_16(SndHalf), 8 )));
   end Write16;
48
   procedure Read32 (Val1, Val2, Val3, Val4 : in Unsigned_8;
50                   Res : out Unsigned_32) is
   begin
52    Res := (Unsigned_32(Val1) or
              Shift_Left(Unsigned_32(Val2), 8) or
54              Shift_Left(Unsigned_32(Val3), 16) or
              Shift_Left(Unsigned_32(Val4), 24));
56    pragma assert (Res =
                     (Unsigned_32(
58                      Unsigned_16(Val1)
                        or Shift_Left(Unsigned_16(Val2), 8)
60                    ) or Shift_Left(Unsigned_32(
                        Unsigned_16(Val3)
62                      or Shift_Left(Unsigned_16(Val4), 8)
                      ), 16)));
64 end Read32;

66 end Bitwise;
```

bitwise.adb

2.3 Nonlinear Arithmetic

We call nonlinear any arithmetic operation which is not an addition or subtraction, nor a multiplication by a constant. The non-linear operations available in Ada are: multiplication, division, exponentiation and the modulus operations `mod` and `rem`, that are defined for both signed and modular integer types. In fact, the bitwise operations could also be considered nonlinear. Although nonlinear operations may occur with integers or floating point numbers, we only consider here integer operations.

The challenge is that nonlinear operations are not well-supported by SMT provers, so most verification conditions which involve more than a simple nonlinear operation are usually not proved.

```
package body Nonlinear with
  SPARK_Mode
is

   procedure Scale (X, Y, Z : Natural_32; Res : out Natural_32) is
   begin
      pragma Assume (X <= Y);
      pragma Assume (Y < Z);
      pragma Assume (Y < 2 ** 15);
      Res := (X * Y) / Z;
      pragma Assert (Res <= X);
   end Scale;

   procedure Unsigned_Scale (X, Y, Z : Unsigned_32;
                             Res : out Unsigned_32) is
   begin
      pragma Assume (X <= Y);
      pragma Assume (Y < Z);
      Res := (X * Y) / Z;
      pragma Assert (Res <= X);
   end Unsigned_Scale;

   procedure Divide (X, Y : Positive_32; Res : out Positive_32) is
   begin
      pragma Assume (X >= Y);
      Res := X / Y;
      pragma Assert ((Res * Y) / Res = Y);
   end Divide;

   procedure Power (X : Natural) is
   begin
      pragma Assume (X in 2 .. 29);
      pragma Assert (2 ** X + 2 ** (X-1) < 2 ** (X+1));
   end Power;

   procedure Mult (X, Y : Integer; Res : out Integer) is
   begin
      pragma Assume (X in -10 .. -1 and Y in 1 .. 10);
      Res := X * Y + 1;
      pragma Assert (Res in -99 .. -Y);
   end Mult;
```

```
42
   procedure Round (X, Y, Z : Positive_32; Res : out Natural_32) is
44 begin
      pragma Assume (Y <= Z);
46    Res := (X * Y) / Z;
      pragma Assert (Res <= X);
48 end Round;

50 end Nonlinear;
```

nonlinear.adb

2.4 Arrays

In C programs global arrays must be often initialized to specific values. The following seemingly simple example are in fact hard problem. The two requirements that state that `g` contains only 0 and that `h1` and `h2` are equal are not proved by Alt-Ergo, CVC4 and Z3 with a 10s timeout. It is proved by Alt-ergo with a timeout of 20s, Z3 return unknown in 20s, and CVC4 does not terminate in 100s. In Qed, we help the provers by computing a summary of the values of the array $\forall i. 50 \leq i \leq 500 \implies t[\text{shift}(h_1, i)] = 0$.

```
const int g[500]= {
2   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
4   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
6   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
8   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
10  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
12  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
14  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
16  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
18  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
20  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
22  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
24  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
26  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};
28
const int h1[500]= {
30  0 ,  1,  2,  3,  4,  5,  6,  7,  8,  9,
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
32  20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
```

```
  30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};

const int h2[500]= {
  0 ,  1,  2,  3,  4,  5,  6,  7,  8,  9,
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
  30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};
```

```
90
/*@
92   requires \forall integer k; 0 <= k < 500 ==> g[k] == 0;
   requires \forall integer k; 0 <= k < 500 ==> h1[k] == h2[k];
94  @*/
void main(){
96   int *i = &g; /* force g, h1, h2 to be in the typed model */
   i = &h1;
98   i = &h2;
   return;
100 }
```

initialization.c

The basic memory model for C [4] (one where proofs are harder to do but that express all the semantic of C) needs to model each bit, or more precisely to allow reading contiguous set of bit in different way. Some programs copy a float or integer char by char or convert a double into 64 bits integer. In fact that continuous set of bits can be seen as a memory. So we only need a theory of piece of memory, that we will call vectors. These vectors have the additional advantage to be useful for expressing the side effect of function calls since they model part of the memory. The big picture of this theory is the following:

```
type vector
2
val start: vector -> int
4 val stop : vector -> int

6 val extract: vector -> int -> int -> vector
(** [extract v start stop] return the sub-vector that goes from
8 [start] to [stop]. It is unspecified if [start] and [stop] are not
in the bound of [v] *)
10
val blit: vector -> vector -> vector
12 (** [blit v1 v2] replace the sub-vector of [v1] located at
[start v2] and [stop v2] by v2 *)
14
val translate: vector -> int -> vector
16 (** [translate v x] move [v] index by [x] *)

18 val to_double: vector -> real
(** convert the vector to double, it is unspecified if it doesn't
20 have the right length *)

22 val to_signedint  : vector -> int
(** convert the vector to int, taking account the sign bit *)
24
val to_unsignedint  : vector -> int
26 (** convert the vector to int *)

28 val from_double: real -> vector
(** convert the double to vector, it is unspecified if it the real
30 is not a representable double. *)

32 val from_int  : vector -> int
```

```
(** convert the int to a vector *)
```

This theory could be even generalized by allowing non contiguous chunk of memory.

Chapter 3

Tools

3.1 Models

The previous chapter showed many example where we want the solver to prove the formula. But of course for wrong programs it should be not the case. When the prover is not able to prove a VC, could happen for different reason:

1. The VC is indeed true, so it means that the prover is not powerful enough to prove it
2. The VC is really false but the program does verify the property, it means that the translation of the program into the formula over approximate too much the behavior of the program.
3. The program doesn't verify the property.

The last case can be even further split, there is a bug in the program or the specification doesn't express the real intent of the user. But only the user can make the difference.

On the other end the prover result when it is not able to prove a VC is not unique:

5. The solver found a model of the VC and it can prove that it is a true one.
6. The solver stops by itself because it reached some internal heuristic limits (we says that it answer unknown)
7. The solver stops because it reached the timeout imposed externally

The last two cases are quit similar in a user perspective, except that in the later case there is hope that with more time the solver will give another answer.

By correctness of the prover the case 5 can't appear with 1. By execution of the model given by the prover on the program with the semantic of the programming language one can distinguished the last two cases.

On the other end the model given in the case 6 and 7 are less useful. Its only property is that it has not yet been ruled out. By execution one can find if it is in the case 3 but if it is not it is hard for the user to find how to help the prover for the remaining case because the model is too precise. Indeed it is hard for the user to infer from one model the set of problematic models in order to add informations (assertions, lemmas) for removing them. So we need to get more informations on where is prover is searching. For example it can help to know :

- the set of values that have already been eliminated (level 0)
- the decisions that have been taken in the current branch
- the set of values that have been eliminated at each decision level.

How to present these informations to the user can be challenging but interesting.

3.2 API

Solvers are used in more and more ways. Firstly they have been used just in a binary way sat/unsat, then models gave more informations in the sat cases, now people are using solvers for abstract interpretation propagators [6]. For that last case the information is not efficiently extracted from the actual solver. So we need solvers that provide API for creating terms and for obtaining an over-approximation of the domains of some terms under the hypothesis of the domain of some other terms.

Bibliography

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard version 2.0. *Technical report, University of Iowa*, december 2010.

[2] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[3] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. *ACM SIGPLAN Notices*, 40(1):338–350, 2005.

[4] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[5] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008.

[6] D. Monniaux. Automatic modular abstractions for template numerical constraints. *Logical Methods in Computer Science*, June 2010.

[7] www.spark-2014.org.