Software Verification — Homework

Jérôme Leroux, Vincent Penelle, and Grégoire Sutre

Univ. de Bordeaux & CNRS, LaBRI, UMR 5800, Talence, France

Abstract Several numerical abstract domains have been studied and implemented in the software verification course. The goal of this homework is to implement another numerical abstract domain, namely the domain of *arithmetical congruences* proposed by Granger [1,2].

1 Project Overview

The properties that can be discovered by abstract interpretation heavily depend on the abstract domain(s) used for the analysis. Three numerical (non-relational) abstract domains have been presented during the course: constants, signs and intervals. The objective of this homework is to implement a fourth numerical abstract domain: arithmetical congruences.

1.1 The abstract domain of arithmetical congruences

This subsection introduces the abstract domain of arithmetical congruences. Please refer to [1,2] for complete details and for the characterization of lattice operations and numerical operations on this abstract domain. The publications [1] and [2] can be found as PDF files in `~gsutre/Library` on the CREMI network.

Recall that an arithmetical *congruence class* is a subset of $\mathbb{Z}$ of the form $r + m\mathbb{Z}$ where $r, m \in \mathbb{Z}$. Such a set is completely determined by the pair (r, m). Put differently, a pair $(r, m) \in \mathbb{Z} \times \mathbb{Z}$ provides a machine representation of the congruence class $r + m\mathbb{Z}$. But the integers r and m are not unique in general. For instance, $2 + 3\mathbb{Z} = -4 + 3\mathbb{Z} = 5 - 3\mathbb{Z}$. This means that the pairs $(2, 3)$, $(-4, 3)$ and $(5, -3)$ represent the same congruence class. However, the integers r and m are unique if we require them to satisfy the following condition:

$$m = 0 \quad \text{or} \quad 0 \leq r < m \tag{1}$$

Formally, every congruence class admits a unique decomposition of the form $r + m\mathbb{Z}$ where $r, m \in \mathbb{Z}$ satisfy (1). Accordingly, the set A of abstract congruence classes is defined as

$$A \;=\; \{\bot\} \;\cup\; \{(r, m) \in \mathbb{Z} \times \mathbb{Z} \mid m = 0 \vee 0 \leq r < m\}$$

The *concretization* function $\gamma : A \to 2^{\mathbb{Z}}$ and the *partial order* $\preceq$ on A are defined as expected by

$$\begin{aligned} \gamma(\bot) &= \emptyset \\ \gamma(r, m) &= r + m\mathbb{Z} \\ a \preceq b &\Leftrightarrow \gamma(a) \subseteq \gamma(b) \end{aligned}$$

The partially-ordered set $(A, \preceq)$ turns out to be a complete lattice. Its infimum is $\bot$ and its supremum is $(0, 1)$.

The last ingredient to obtain a Galois connection is the *abstraction* function $\alpha : 2^{\mathbb{Z}} \to A$. As expected, for every $r, m \in \mathbb{Z}$ satisfying (1), we let

$$\begin{aligned} \alpha(\emptyset) &= \bot \\ \alpha(r + m\mathbb{Z}) &= (r, m) \end{aligned}$$

It is shown in [1,2] that the intersection of a family of congruence classes is either empty or a congruence class. So we extend α to $2^{\mathbb{Z}}$ by letting $\alpha(S)$ be the abstraction of the least congruence class containing S. Formally,

$$\alpha(S) \;=\; \alpha\left(\bigcap \{C \mid C \in \mathcal{C} \wedge S \subseteq C\}\right)$$

where $\mathcal{C}$ denotes the set of all congruence classes. For instance,

$$\begin{aligned} \alpha(\{2, 7, 12\}) &= \alpha(2 + 5\mathbb{Z}) = (2, 5) \\ \alpha(\{-3, 5, 17\}) &= \alpha(5 + 4\mathbb{Z}) = (1, 4) \end{aligned}$$

You are invited to verify that, by construction, $(2^{\mathbb{Z}}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preceq)$ is a Galois connection (it is even a Galois insertion).

1.2 Congruence linear diophantine equations with one variable

Some operations on abstract congruence classes (such as the greatest lower bound) require solving *congruence linear diophantine equations with one variable*, i.e., equations of the form

$$cx \in (r + m\mathbb{Z}) \tag{2}$$

where c, r, m are three given integers with $m \geq 0$. It can be shown (see, e.g., Proposition 4.3 in [1]) that the set S of all $x \in \mathbb{Z}$ such that (2) holds is non-empty if, and only if, $r \in g\mathbb{Z}$ where g is the greatest common divisor[1] of c and m. Moreover, if S is non-empty, then given any element $s \in S$, it holds that

$$S \;=\; s + \left(\frac{m}{g}\right)\mathbb{Z}$$

In practice, when S is non-empty, an element s in S can be found using *Bézout coefficients* for (c, m). Indeed, assuming that $g \neq 0$ and that $r \in g\mathbb{Z}$, and given $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$ such that $cx + my = g$, the integer $s = \frac{r}{g}x$ satisfies $cs \in (r + m\mathbb{Z})$.

[1] If both c and m are zero then g is defined to be zero.

2 Working on the Project

2.1 Homework's archive

The archive encloses a skeleton of your project with:

- This PDF document (in the root directory).
- The "`.mli`" files that define the API of the whole project. Run '`make doc`' to generate the corresponding HTML documentation and visit `./doc/index.html` with a web browser to look at it.
- The necessary "`.ml`" files to provide you functions to help you to focus on the heart of the program (look at it and use it to save time!). In particular, `Arithmetic.mli` provides useful functions related to divisibility such as `gcd` and `bezout`.
- Binary programs (`sai.solution.byte` and `sai.solution.native`) to give you a taste of the final static analyzer that you should obtain.
- Once your have implemented the missing module (see below), you can run unit tests with '`make test-domcongruence`'. These unit tests rely on the functions `print`, `abs` and `lub`. These functions should be implemented fully and correctly before running unit tests.
- The whole analyzer can be compiled with '`make`'. You can test your analyzer on the examples given in the `examples` directory, and compare your results with the ones given by `sai.solution.byte` or `sai.solution.native`. Two examples are of particular interest: `congruence_1.aut` and `congruence_2.aut`. You are of course welcome to add your own examples.

2.2 Tool invocation

Run `./sai.byte -help` to know how to use the program and which arguments to give it.

2.3 What to code ?

You have to implement the `DomCongruence` module by filling the empty file `lib/DomCongruence.ml`. This module shall implement the numerical abstract domain of arithmetical congruences. It must conform to the signature specified in the file `lib/DomCongruence.mli` (i.e., the signature `NumericalDomain.S`).

Similarly to the numerical domains implemented in the course, your implementation should use **arbitrary-precision integers** provided by the OCaml module `Z` of the Zarith library.

Unit tests (run with '`make test-domcongruence`') assume that the `print` function behaves in a specific way. Your implementation of the `print` function must conform to the specification given in the following table (where $m \geq 2$).

a	$\bot$	$(r, 0)$	$(0, 1)$	$(0, m)$	(r, m)
`print`(a)	$\bot$	`r`	$\mathbb{Z}$	`m`$\mathbb{Z}$	`r` $+$ `m`$\mathbb{Z}$

As mentioned earlier, lattice operations and numerical operations are detailed in [1,2]. In [1], computable characterizations are given in Corollary 3.3 for lattice operations, Proposition 4.1 for abstract addition and multiplication, and in the paragraph overlapping pages 178–179 for abstract division.

In addition to providing arbitrary-precision integers, the `Z` module provides some functions that can be leveraged to implement the `DomCongruence` module, namely `divisible`, `erem`, `gcd` and `gcdext`.

You will probably find it useful to implement a solver for congruence linear diophantine equations with one variable (see Subsection 1.2). Such a solver can be used to compute the greatest lower bound and to implement the `Op.equality` function.

2.4 A word of advice

Be **very careful** with the `(/)` and `(mod)` functions of OCaml. They behave as expected (i.e., they return the quotient and remainder of the Euclidean division) when their arguments are nonnegative:

```
# 5 / 2;;
- : int = 2
# 5 mod 2;;
- : int = 1
```

But the results do not conform to the standard Euclidean division in general as the remainder may be negative:

```
# -5 / 2;;
- : int = -2
# -5 mod 2;;
- : int = -1
```

The functions `(/)` and `(mod)` of the `Z` module behave in the same way. This is not specific to OCaml by the way. Most programming languages (e.g., C) behave in the same way.

To avoid potential mistakes in your code, we advise you to:

1. Avoid the `(mod)` and `rem` functions of the `Z` module. Use the function `erem` instead.
2. Each time you write `x / y`, make sure that $\texttt{y} > 0 \ \wedge \ (\texttt{x} \geq 0 \ \vee \ \texttt{x} \in \texttt{y}\mathbb{Z})$.

2.5 How and when to send your homework ?

The deadline for the project is on Friday, January 20, 2023 at 23h59 (CET). Put an archive with your full project on the ≪Projet≫ activity on the moodle page of the course. Your archive should contain the whole project including:

- The `DomCongruence.ml` module fully implemented.
- A few extra tests for the implemented module.
- A report (written in LaTeX but sent as a PDF file) about your implementation and the choices that you had to do while programming.

References

1. Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.
2. Philippe Granger. *Analyses Sémantiques de Congruence*. PhD thesis, Ecole Polytechnique, July 1991.