Mälardalen University Master Thesis

Abstract Interpretation and Abstract Domains

with special attention to the congruence domain

Stefan Bygde

May 2006

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

Abstract

This thesis is a survey on a framework for program analysis known as Abstract interpretation. Abstract interpretation uses abstract semantics to obtain important properties of programs. Different abstract semantics can be used in abstract interpretation, to obtain different properties. For instance there are semantics that spots upper and lower limits for the values of the variables of the program. These different semantics are called *abstract domains* and this thesis is meant to summarize different numerical abstract domains as well as give mathematical properties to them. The thesis also offers a small survey of free software libraries that implements abstract domains. Special attention will be given to the *congruence domain*. The congruence domain was implemented in a program analysis tool developed at Mälardalen University. In the congruence domain the property "variable x is always congruent to n modulo m" is obtained. Abstract operators for this domain are presented and new bit-operators are introduced.

Contents

Acknowledgments

This Master's Thesis has been written at Mälardalen University, Department of Computer Science and Electronics between January 2005 and April 2006.

I would especially like to thank my supervisor Björn Lisper and Andreas Ermedahl at the department for help, useful suggestions, comments and for developing the idea for the thesis. Also, thanks go to Jan Gustafsson and Christer Sandberg from the WCET project. Thanks also to Jan Carlsson for recommending me the WCET project.

1 Introduction

Static program analysis is an automatic process that obtains properties of a program by analysing its source code or object code. It has been widely used in optimizing compilers, for instance. But as software becomes more and more important in our society and as the reliability of software becomes safety-critical, static program analysis has become more important. Program analysis can and has been used to find errors in software or to verify correctness of programs. Obtaining information about the time a program will take to execute is also possible with program analysis. However, it is not possible to get exact information about programs with static analysis. For example, it would be (very) nice if program analysis could guarantee that a program is correct or if it could tell the how exact time that a certain program will take to execute but such problems are undecidable (that is, impossible for a mechanical process to solve). Instead of solving undecidable problems, program analysis can introduce some sort of *approximation* to transform it to a decidable problem to the cost of loosing information.

Abstract interpretation is a framework for program analysis introduced by Radhia and Patrick Cousot [CC77]. Generally, abstract interpretation can be regarded as a theory of abstracting for complete lattices. In practice, the lattice to be abstracted corresponds to the state space of a program. The basic idea is that a (possibly infinite) set of states is approximated by one *abstract state*. This abstraction is made based on some property of interest. The variables and functions of a program is also abstracted using the same property. This leads to a computable set of states which correctly approximates the program.

Abstract interpretation has been used in different contexts as theory for abstraction. For instance, it has been used in model checking. Model checking is an approach to formal verification, that requires exhaustive search of the state space of a program. Since the state space in general is far too vast to be explored exhaustively, abstraction is typically required. This abstraction can be done using the framework for abstract interpretation.

Abstract interpretation simulates execution with an abstract semantics over a program. Different abstract semantics gives information about different properties of the program. The different forms of abstraction over the semantics are called *abstract domains*.

1.1 Static analysis and WCET

A hard real-time system is a system where the importance of the program meeting its deadlines are as important as the correctness of the program itself (in fact, the program is considered to be incorrect if the deadlines are not met). Typically, safety-critical systems are hard real-time systems. In this context it is important to have information about the execution time of a program. Execution time, of course, depends on hardware, software platform, input, etc. The execution time varies from time to time and measuring the execution time on one system may not agree with measurements on another. An important notion when it comes to measuring execution time is the WCET (Worst Case Execution Time) which is, as the name suggests, the maximal time it can take for a certain piece of code to execute on a specific system. The WCET is important to know because one can be sure that the piece of code always takes less than or equal the amount of time of the WCET, thus, the WCET is a time-measure. However, to determine the WCET for a piece of code on a system automatically is impossible due to the difficulty caused by schedulers, cache-memories, execution path's etc. It is,

however, possible to determine a *safe upper bound* for the WCET of a piece of code. A safe upper bound of the WCET is any time value that is equal to or greater than the real WCET. For example, an ”infinite amount of time” would be a safe upper bound for *any* WCET, but it would not give any useful information about the actual WCET. Therefore, the ambition is to calculate such a *tight* upper bound as possible, that is to say an upper bound that is as close as possible to the actual WCET.

To be able to calculate a safe upper bound for a WCET of a program, we must consider both the source code of the program and the given hardware. A WCET tool usually implement this in two different phases and then combines the results to give the final result. These phases are called *high-level analysis*(for the source/object code), *low-level analysis* (for the target platform) and the *calculation phase*(where the results from high- and low-level analysis are combined). High-level analysis analyses the source or object code to find loop-bounds or execution counts for basic blocks, whereas low-level analysis calculates execution times on atomic parts of the program based on the hardware model. The hardware model must take pipe-lines, cache memories, etc. into account. Finally, the results from the different phases are combined in the calculation phase. This calculation can be done with different techniques. Calculation techniques include, for example IPET(Implicit Path Enumeration Technique).

1.2 SWEET

A static time analysis tool is currently under development at Mälardalen University [MdH06]. This tool is called SWEET (SWEdish Execution time Tool). The purpose of SWEET is to find a safe upper bound to WCET of a program automatically, without active user-interaction or annotations.

In SWEET, the high-level analysis is performed in four steps: program reduction, value analysis, syntactic analysis and abstract execution. The program reduction prepares the program for the other phases by eliminating variables that are not important for the control flow, from the program. The value analysis uses abstract interpretation to get information about possible values for the variables. The syntactical analysis tries to find known patterns of loops to give an estimation of the loop bounds. The abstract execution simulates program execution over abstract states. The simulation is based on abstract interpretation.

The analysis is performed over an intermediate language called NIC (New Intermediate Code), developed at Uppsala University [GLSB03]. That it is an intermediate language means that it is on a lower level than the source code but on a higher level than the object code. There are benefits in performing the abstract interpretation on an intermediate language; it is close to the object code, so the code won't change too much when compiled but it also contains more information than the object code which will give a richer analysis.

NIC stores values of integers as bit-strings of length 8,16 or 32. The type information is not as precise as it is in C in NIC. In C, an integer can be declared as signed or unsigned. This type information is not explicitly expressed in NIC. The abstract interpretation that we shall be interested in deals only with bitstrings interpreted as integers , therefore, Table 1 shows the integer- and bit operators that NIC supports.

The number inside parentheses is the number of arguments that the operator takes. All operators except the relational ones returns integers and all arguments are integers, the relational ones returns booleans. Some operators has signed and unsigned versions because the operators are implemented differently on a lower level. A signed operator asserts thats all its arguments are interpreted as signed, and an unsigned interprets all

Arithmetic operators	
`neg` (1)	negation
`add` (2)	addition
`sub` (2)	subtraction
`u_mul` (2)	unsigned multiplication
`s_mul` (2)	signed multiplication
`u_div` (2)	unsigned division
`s_div` (2)	signed division
`u_mod` (2)	unsigned remainder
`s_mod` (2)	signed remainder
Bit operators	
`l_shift` (2)	left shift (second argument is number of steps)
`r_shift` (2)	right shift (second argument is number of steps)
`r_shift_a` (2)	arithmetical right shift (second argument is number of steps)
`not` (2)	bitwise not
`and` (2)	bitwise and
`or` (2)	bitwise or
`xor` (2)	bitwise xor
`trunc` (2)	truncate bits (second argument is number of bits)
`z_ext`(2)	zero-extend bits (second argument is number of bits)
`s_ext`(2)	signed-extend bits (second argument is number of bits)
Relational operators	(returns true/false)
`cmp_eq`(2)	compare integers for equality
`cmp_ne`(2)	compare integers for non-equality
`u_cmp_lt`(2)	unsigned less than comparison
`s_cmp_lt`(2)	signed less than comparison
`u_cmp_ge`(2)	unsigned greater than or equal comparison
`s_cmp_ge`(2)	signed greater than or equal comparison
`u_cmp_gt`(2)	unsigned greater than comparison
`s_cmp_gt`(2)	signed greater than comparison
`u_cmp_le`(2)	unsigned less than equal comparison
`s_cmp_le`(2)	signed less than or equal comparison

Table 1: Integer operators supported by NIC

its arguments as unsigned. Therefore, it is possible to determine the sign of a variable when it is applied to a signed or unsigned operator.

1.3 Motivation and results

As mentioned, important parts of the flow analysis in SWEET uses abstract interpretation. Currently, the abstract interpretation uses the *interval domain* which is an abstract domain that can determine safe lower and upper limits program variables. However, there are many abstract domains and they can be used to obtain different properties of programs. The choice of abstract domain is a trade-off between efficiency and precision. The main purpose of this thesis is to examine the possibility to enhance the abstract interpretation used in SWEET. This has resulted in a formal introduction to abstract interpretation, a summary of invented numerical abstract domains, a summary of free software libraries of abstract domains, a small survey on mathematical properties of abstract domains and finally a special study of the congruence domain. The congruence domain seems like a reasonable domain to add to the SWEET tool. Since SWEET analyses low-level code, the congruence domain has to be enhanced with bit-level operators to work fully. No such operators has to our knowledge been published, so we will introduce bit-operators for the congruence domain in this thesis. A prototype implementation of the congruence domain has also been developed, a brief description of this implementation is also included in this work.

1.4 Related work

The original idea of abstract interpretation was presented by Patrick and Radhia Cousot [CC77]. They have extended their work in several papers, such as [CC79].

An introduction to program analysis in general (including abstract interpretation) is given in [NNH05]. Introduction to lattice theory, which is an important part of abstract interpretation, can be found in [DP90].

As several abstract domains will be presented in this thesis, a list of the inventors and the domains, as well as pointers to the original work is provided.

- Antoine Miné - The octagon domain [Min06]
- Patrick Cousot/Nicolas Halbwachs - The convex polyhedra domain [CH78]
- Philippe Granger - The congruence domains [Gra89].
- Patrick Cousot - The interval domain [CC77]

The original idea of manipulating domains by regarding them as elements of a lattice was also first presented by Cousot & Cousot [CC79]. Later expanded by complementation [CFG+97].

Publications on the SWEET tool can be found on [MdH06]. Important publications includes [Gus00] and [Erm03].

1.5 Outline

The first part of the thesis is an introduction to abstract interpretation. It will build the idea of abstract interpretation step-by-step in sections 2 through 5.

Section 6 will then work as a survey of existing abstract domains with discussion on their properties and complexity.

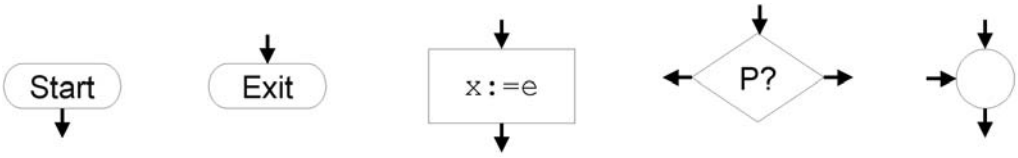

Figure 1: Flowchart nodes

A brief introduction to the nature of abstract domains as mathematical objects and their properties is presented in section 7, followed by a list of implemented and free-to-use software libraries of abstract domains in section 8.

The congruence domain is then studied in detail in Secion 9, where abstract operators of this domain is summerized, and new bit-operators are introduced.

A prototype implementation of the abstract congruence domain has been implemented and it is described in section 11.1.

Conclusions, discussions and future work are in Sections 12 through 13

2 Collecting semantics

The following sections will introduce abstract interpretation in a fashion quite close to the original description by the Cousots. It will begin by introducing a concrete collecting semantics which is the foundation on which abstract interpretation is built, followed by a section about abstraction and finally defining the abstract interpretation.

The semantics of a program defines the mathematical *meaning* of a program. The semantics is defined for each part of a programming language's constructs and maps each program to a mathematical representation of its meaning. For imperative languages, the semantics often computes the *states* of a program. The collecting semantics is a semantics that computes all possible memory states that can occur during the execution of the program. However, in most cases the calculation for finding out all the exact memory states that occurs during execution is practically impossible. But since it in theory computes the exact states, it is a good foundation to build program analyses on. Abstract interpretation uses the collecting semantics as base for theory.

2.1 Definitions

To be somewhat language independent in the presentation we will consider a program as a flowchart with five types of nodes (see figure 1): start, exit, assignment, conditional and merge nodes. A program is assumed to start at the start node and, if the program terminates, it will do so at the exit node. An assignment node assigns a program variable to an expression. The conditional node has two emerging edges called the *true* and the *false* edge. The true edge is followed if the conditional P is true for the current memory configuration, otherwise the false edge is followed. The merge nodes has at least two incoming edges. This representation demands that the language is imperative but not so much more.

Let Prg be a program, then we define IDENT_{Prg} to be the set of identifiers (variables) in the program Prg. However, the more convenient notation IDENT will be used when there is no risk of ambiguity. Let $\mathcal{V}$ be the set of values that each identifier can be assigned to. Before an identifier has been assigned to a value it is assumed to be assigned to the special value $\perp \in \mathcal{V}$. If a value is assigned to $\perp$ it means that it is

uninitialized. The values that each identifier, at a certain moment, is assigned to can be represented as a map $\sigma : \text{IDENT} \rightarrow \mathcal{V}$ so that each identifier is associated with a value. Such a map will be called an *environment*. Thus, an environment can be regarded as a memory that holds the values for each identifier. The set of all possible environments will be denoted ENV. It is assumed that every instruction is executed in an environment and that the instruction possibly can update this environment.

A *program point* is an edge of the flowchart, that is a pair $\langle i_1, i_2 \rangle$ where i_1 and i_2 are nodes of the flowchart. The set of possible program points will usually be denoted Q. Each program point can be associated with one or several environments. It is natural to associate a program point $\langle i_1, i_2 \rangle$ with the environment that was updated by the node i_1 and in which i_2 will execute. An environment associated with a certain program point $q \in Q$ is called a *state*. A state is denoted S_q (where q is a program point) or possibly $\langle \sigma, q \rangle$ where $\sigma \in \text{ENV}, q \in Q$. The set of all states is denoted $\text{STATES} = \text{ENV} \times Q$ and the set of all states associated with a certain program point q is denoted STATES_q. With this terminology, we can define a collecting semantics.

2.2 The transition system

The goal of the semantics is to collect all environments that can possible occur at each program point during execution. The semantics can be described with a transition system which maps a set of states to a new set of states depending on the type of program point.

$$\tau \colon \mathcal{P}(\text{STATES}) \rightarrow \mathcal{P}(\text{STATES})$$

The actual definition of this function depends on the program. The states are then collected by solving a fixed point equation. This computation simulates execution of the program with all possible input. So after the computation is done, a *set* of states is associated with each program point. A set of states at a program point is called a *context* (Thus, a context is an element from the set $\mathcal{P}(\text{ENV})$). Such a context represents all values that all variables assume under execution of the program with all possible input. So after applying the transition function and a fix point is reached, a context is associated with each program point.

The subsets of the set ENV (in other words the set of contexts) constitutes a complete lattice:

$$(S, \sqcap, \sqcup, \sqsubseteq, \top, \bot) \Leftrightarrow (\mathcal{P}(\text{ENV}), \cap, \cup, \subseteq, \text{ENV}, \varnothing)$$

This lattice is called *the concrete semantic domain*. Now, with the concrete semantic domain and the transition system we have the tools for computing all different states that can possible occur for any given input during execution. Unfortunately this is too good to be true; the computation gives exact information of the program for all possible input and is uncomputable at compile time. If this calculation could be made, the halting problem could have been solved, which is impossible. Luckily, there are ways to deal with this problem. The key is to simulate this semantics over an *abstract domain* to make it computable. In next section we define the concept of abstraction.

3 Abstract Interpretation

The concrete semantic domain is clearly to advanced to be of practical use. If we replace the concrete semantic domain with something simpler, it is possible to get an approximate result in finite time. This result will not be as precise as the ”real”

Figure 2: The lattice of signs

uncomputable result from the concrete semantic domain. An abstract domain is in some way an abstraction of the concrete semantic domain. In the abstract domain we ”forget” some of the properties of the concrete semantic domain in order to get the domain computable and computer-representable. The abstract domain has to be *correct* in the sense that all states derived in the concrete semantic domain should also be derived in the abstract domain, but the abstract domain will also include some states that actually not will occur in the concrete one – we get an over approximated result. To mimic the concrete semantic domain, the abstract domain should also be a complete lattice. There should then be a correspondence between the concrete semantic domain and the abstract domain to make sure that the abstract domain actually is a sound approximation of the concrete semantic domain.

3.1 The idea of abstraction

Suppose that $\mathcal{V} = \mathbb{Z} \cup \{\bot\}$ is the set of values that identifiers can be assigned to. Since we cannot compute the exact context for a program point, we drop some information in order to get a computable model. For instance, drop all information about every variable except whether it is positive, negative or zero at a program point. This can be done by replacing $\mathcal{V}$ with the set $\hat{\mathcal{V}}_{sign} = \{\bot, -, 0, +, \top\}$, where $\bot$ means that the variable has no value, $\top$ means that we don't know the sign of the variable and $-, 0$ and $+$ has the obvious meanings. It is easy to see that each value in $\mathcal{V}$ corresponds to a value in $\hat{\mathcal{V}}_{sign}$. We can arrange the values of $\hat{\mathcal{V}}_{sign}$ in a complete lattice as shown in figure 2. We will refer to $\hat{\mathcal{V}}_{sign}$ as the *sign domain*. We say that $\hat{\mathcal{V}}_{sign}$ is an *abstraction* of $\mathcal{V}$ because it contains less information about the variable than the set $\mathcal{V}$, even though the information is ”correct”. The idea of abstraction will be formalized in next section.

3.2 Galois connections

Given two lattices C and A, we can define a pair of maps
$(\alpha, \gamma) \in (C \to A) \times (A \to C)$. Further, for (α, γ) the following properties should hold:

$$\alpha \circ \gamma \sqsubseteq id_A \text{ and } \gamma \circ \alpha \sqsupseteq id_C$$

where id_X is the identity function on X. A pair of maps with these properties is known as a *Galois connection*. If it also holds that

$$\alpha \circ \gamma = id_A$$

Then the maps are called a *Galois insertion*. When used in abstract interpretation we call α the *abstraction* map and γ the *concretization* map. If (α, γ) are given for an

abstraction A of C, then A is a sound abstraction of C. That is, we can be sure that A correctly abstracts the information of C. In the context of abstract interpretation we shall say that A *abstracts* C if there is a galois connection $(\alpha, \gamma) \in (C \to A) \times (A \to C)$.

3.2.1 An example

Consider the sign abstraction from Section 3.1. Let $A = \hat{\mathcal{V}}_{sign}$ and $C = \mathcal{P}(\mathbb{Z})$. Then we can define a galois connection between A and C as follows:

- $\alpha_{sign}(A) = +$, if all elements in A are greater than zero
- $\alpha_{sign}(A) = -$, if all elements in A are less than zero
- $\alpha_{sign}(\{0\}) = 0$
- $\alpha_{sign}(\varnothing) = \bot$
- $\alpha_{sign}(A) = \top$, otherwise
- $\gamma_{sign}(+) = \mathbb{Z}_+$
- $\gamma_{sign}(-) = \mathbb{Z}_-$
- $\gamma_{sign}(0) = \{0\}$
- $\gamma_{sign}(\bot) = \varnothing$
- $\gamma_{sign}(\top) = \mathbb{Z}$

where $\mathbb{Z}_+ = \{x \in \mathbb{Z} | x > 0\}, \mathbb{Z}_- = \{x \in \mathbb{Z} | x < 0\}$. It is easy to verify that this is in fact also a Galois insertion.

3.3 Relational and non-relational domains

The concrete semantic domain $\mathcal{C}$ is defined as the complete lattice of environments, or formally $\mathcal{C} = \mathcal{P}(\text{IDENT} \to \mathcal{V})$. Thus, a domain which abstracts the concrete semantic domain is an abstraction of concrete environments. The abstractions of the concrete semantic domain fall into two important categories; those that preserves relations between variables and those that don't. These two categories are known as *relational domains* and *non-relational domains*.

3.3.1 Non-relational domains

In a non-relational domain all identifiers are abstracted independent of each other. In this case a concrete context is abstracted through *one* map. We will formalize how an non-relational domain abstracts a set of concrete environments (i.e. a context). Let $\mathcal{C} = \mathcal{P}(\text{IDENT} \to \mathcal{V})$ be the concrete semantic domain. Also, let there be a galois connection $(\alpha_{\mathcal{V}}, \gamma_{\mathcal{V}}) \in (\mathcal{V} \to \hat{\mathcal{V}}, \hat{\mathcal{V}} \to \mathcal{V})$. Then we can construct a galois connection between the concrete semantic domain $\mathcal{C}$ and a non-relational abstract domain $\mathcal{A} = \text{IDENT} \to \hat{\mathcal{V}}$ through the following galois connection:

$$\gamma_{NR}(a) = \{\lambda x.v | v \in (\gamma_{\mathcal{V}} \circ a)(x)\}$$

$$\alpha_{NR}(c) = \lambda x.\alpha_{\mathcal{V}}(\bigcup\{\tau(x)|\tau \in c\})$$

where $a \in \mathcal{A}, c \in \mathcal{C}$. A non-relational domain relies on an abstraction between the set of values $\mathcal{V}$ and an abstraction $\hat{\mathcal{V}}$. Each identifier is then approximated independent of each other. To summarize, a non-relational domain is a map which maps each variable on an *abstract value*. This approach yields an effective but rather imprecise analysis.

To give an example of an non-relational domain, consider the sign abstraction from Section 3.1. Set $(\alpha_{\mathcal{V}}, \gamma_{\mathcal{V}}) = (\alpha_{Sign}, \gamma_{Sign})$ in the definitions above and we get an abstract domain a non-relational abstract domain of signs.

3.3.2 Relational domains

Consider the sign domain mentioned above. Suppose that, in a certain program point, the variable x always is positive iff y is negative, and that x is negative iff y is positive in the concrete context c, further suppose that neither x nor y can be zero. If we compute the result of $\alpha_{NR}(c)$ of the sign domain, we would get a mapping which maps x to $\{-,+\}$ and y to $\{-,+\}$. Evidently, this is a great loss of information, but since α_{NR} treats each variable independently, this is actually the smallest correct mapping we can do with a non-relational domain. To increase precision we might take dependencies between variables into consideration, to the cost of a less efficient analysis of course. To do this, we cannot let a single map approximate an abstract environment. Rather, in this case, we would like to approximate the context with *two* maps – one map where $[x \mapsto +][y \mapsto -]$ and one where $[x \mapsto -][y \mapsto +]$. These two maps abstracts the concrete context and respects dependencies between the two variables which makes the set of them an abstraction of the concrete semantic domain.

We are not going to define a general α and γ for non-relational domains as we did for relational ones because relational domains can be constructed in different ways. For instance, we do not need abstract values (like $\hat{\mathcal{V}}$) to construct relational domains.

To summarize, non-relational domains are more efficient than non-relational domains but they give a less precise analysis because they ignore the relationship between variables. So generally, if more precision is desired, choose a relational domain and if a faster analysis is required, a non-relational one may be a better choice.

3.4 More definitions

As mentioned, to make an interpretation of a program computable, we want to have an abstract interpretation dealing with abstract domains. We shall therefore summarize the abstract versions of the concepts of values, environments and states that was defined in Section 2.1.

- An *abstract value* is an element from a set $\hat{\mathcal{V}}$ which through a galois connection abstracts the set of values $\mathcal{V}$ of a program. For instance is $+$ an abstract value of the sign domain. Note that in some relational domains there is no need for abstract values.

- An *abstract context* is an abstraction (through a Galois connection) of a concrete context. The set of abstract environments is denoted $\widehat{\text{CTX}}$. Note that in a non-relational domain, an abstract context is *one* map, whereas in a relational domain it is a *set* of maps.

- An *abstract state* is an abstract context associated with a program point (similar to the concrete definition). An abstract state is usually denoted $\hat{S}_q$, where q is a program point.

4 Abstract operators

In an assignment in a program, a variable is often assigned to an arithmetic expression. In an abstract domain, the exact value of a variable is not known and hence not the exact value of an expression (if it contains variables). To be able to give an abstract estimation of an expression, all the operators in the expression must be abstracted. How to do this is formalized in this section. Let $f : \mathcal{V}^n \to \mathcal{V}$ be any n-ary operator in a program language. Then we can create another operator $f_{\mathcal{P}} : \mathcal{P}(\mathcal{V}^n) \to \mathcal{P}(\mathcal{V})$ which is defined as

$$f_{\mathcal{P}}(M) = \{f(z^n) | z^n \in M\} \tag{1}$$

Now, let $\hat{\mathcal{V}}$ be an abstraction of $\mathcal{V}$. An *abstract operator* is an operator $\hat{f} : \hat{\mathcal{V}}^n \to \hat{\mathcal{V}}$ such that

$$f_{\mathcal{P}}(M) \sqsubseteq \gamma \circ \hat{f} \circ \alpha(M) \tag{2}$$

for all $M \in \mathcal{P}(\mathcal{V})$.

In the same way, it is possible to abstract relations on the form $\sim: \mathcal{V}^n \to \{True, False\}$ to an abstract version $\hat{\sim} : \hat{\mathcal{V}} \to \{\bot, True, False, \top\}$.

4.1 An example

Consider the sign domain from Section 3.1. The sign abstraction abstracts $\mathbb{Z}$. Suppose that we want to make an abstract version of the operator $* : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. To do so we begin by examining the properties of $*_{\mathcal{P}}$. By (1) we have

$$A *_{\mathcal{P}} B = \{a * b | (a, b) \in A \times B\}$$

So, by (2) it is required that

$$\{a * b | (a, b) \in A \times B\} \subseteq \gamma(\alpha(A) \hat{*}_{Sign} \alpha(B))$$

The product of any two integers is positive and so are the product of any two negative integers. Thus, any *set* of positive integers multiplied (with $*_{\mathcal{P}}$) with another positive set yields another set of positive integers. It is therefore safe to make the conclusion that

$$+ \hat{*}_{Sign} + = +$$

since $\gamma(+) = \mathbb{Z}_+$ and $\mathbb{Z}_+$ is certainly a superset to any set of positive integers. By simple reasoning one can also conclude that

$$\begin{aligned}
- \hat{*} - &= + \\
- \hat{*} + &= - \\
0 \hat{*} z &= 0, \ \forall z \in \hat{\mathcal{V}}_{Sign} \backslash \{\bot\} \\
\top \hat{*} z &= \top, \ \forall z \in \hat{\mathcal{V}}_{Sign} \backslash \{\bot, 0\} \\
\bot \hat{*} z &= \bot, \ \forall z \in \hat{\mathcal{V}}_{Sign}
\end{aligned}$$

The rest of the cases follows from the commutativity of $\hat{*}$.

To demonstrate an abstract relation, consider the relation $\leq$ on integers. In the sign domain it is easy to verify that:

$$
\begin{aligned}
- \hat{\leq} - &= \top \\
- \hat{\leq} 0 &= True \\
- \hat{\leq} + &= True \\
0 \hat{\leq} - &= False \\
0 \hat{\leq} 0 &= True \\
0 \hat{\leq} + &= True \\
+ \hat{\leq} - &= False \\
+ \hat{\leq} 0 &= False \\
+ \hat{\leq} + &= \top \\
\top \hat{\leq} z = z \hat{\leq} \top &= \top, \ \forall z \in \hat{\mathcal{V}}_{Sign} \backslash \{\bot\} \\
\bot \hat{\leq} z = z \hat{\leq} \bot &= \bot, \ \forall z \in \hat{\mathcal{V}}_{Sign}
\end{aligned}
$$

where $\top$ denotes that the condition may or may not be true.

It is important to realize that arithmetic operators like $+$ and $*$ are defined for integers, hence they can be abstracted only by domains abstracting $\mathbb{Z}$. In abstract interpretation we use abstraction of *contexts*. So in order to do abstractions of $+$ and $*$, we have to have a proper definition of these operators for contexts. In non-relational domains, however, variables are abstracted independently of other variables and each variable is an abstraction of $\mathcal{V}$, so if $\mathcal{V} = \mathbb{Z} \cup \{\bot\}$ then the regular abstractions of $+$ and $*$ can be used.

5 The abstract transition function

With a collecting semantics defined and a notion of abstraction and abstract domains, we can now define how the framework for abstract interpretation works. To mimic the semantics over an abstract domain we shall define the *abstract* transition function:

$$\hat{\tau} \colon \mathcal{P}(\widehat{\text{STATES}}) \rightarrow \mathcal{P}(\widehat{\text{STATES}})$$

The abstract transition function is defined in terms of the considered program. Therefore, next section will begin by classifying all program points.

5.1 Classifying program points

The program points of a program (i.e. the edges of the flowchart) can be categorized in five different groups.

- A program point $\langle n_1, n_2 \rangle$ is called a *start point* iff n_1 is the start node of a program.
- $\langle n_1, n_2 \rangle$ is called an *assignment point* iff n_1 is an assignment node.
- $\langle n_1, n_2 \rangle$ is called a *conditional true point* iff the point corresponds is the true edge from a conditional node n_1.

- $\langle n_1, n_2 \rangle$ is called a *conditional false point* iff the point corresponds is the false edge from a conditional node n_1.
- Finally, $\langle n_1, n_2 \rangle$ is called a *merge point* iff n_1 is a merge node.

5.2 Definition of the abstract function

Two important operators will be defined before the abstract transition function can be presented

$$\mathrm{pre}\colon Q \to \mathcal{P}(Q)$$

and

$$\mathrm{suc}\colon Q \to \mathcal{P}(Q)$$

where Q is the set of program points. The map pre simply maps the set of possible predecessors of a program point (that is, all program points that could immediately precede it). Similarly, suc maps a program point to all its possible immediate successors. Formally:

$$\mathrm{pre}\langle n_1, n_2 \rangle = \{\langle n, n_1 \rangle | n \in \textsc{Nodes}\}$$
$$\mathrm{suc}\langle n_1, n_2 \rangle = \{\langle n_2, n \rangle | n \in \textsc{Nodes}\}$$

where NODES is the set of nodes in the considered flowchart. Note that $|preq| > 1$ iff q is a merge point and the $|sucq| > 1$ iff q is an edge to an conditional node. Now we can define the abstract transition map.

The abstract transition map $\hat{\tau}$ maps from a set of abstract states to a new updated set of abstract states. In the following definition, let $\hat{S}$ be a set of abstract states and let $\hat{S}' = \hat{\tau}(S)$. Also, let $\hat{S}_q$ and $\hat{S}'_q$ denote the abstract context associated with the program point q in $\hat{S}$ and $\hat{S}'$ respectively. The abstract transition function will be defined in terms of $\hat{S}_q$ and $\hat{S}'_q$, so that we get a system of equations.

Initial program point Since we are considering *all* input of a program, we assume that every variable can assume *any* value. This is (for every abstract domain) modeled by the top element of the abstract domain, hence:

$$\hat{S}'_q = \top$$

if q is an initial program point.

Assignment To avoid complexity, only this type of program point will only be explained for non-relational domains, where each program point is associated with *one* abstract environment only. In a non-relational domain $\hat{S}_q$ is a function from variables to abstract values.

Assuming that every arithmetic operator $*$ of $\mathcal{V}$ has an abstract version $\hat{*}$ (see Section 4), we can also for every expression E create an abstract version of the expression $\hat{E}(\hat{S}_q)$ by replacing all operators $*$ in E with $\hat{*}$ and all constants n with $\alpha(n)$. Further, all identifiers x in E has to be replaced with $\hat{S}_q(x)$, which means that the identifiers are taken from the abstract value $\hat{S}_q$.

$$x = E$$

yields the equation

$$\hat{S}'_q = \hat{S}_{\mathrm{pre}\,q}[x \mapsto \hat{E}(\hat{S}_{\mathrm{pre}\,q})]$$

if q is an assignment point. The function $f[x \mapsto a]$ is the function f but with $f(x) = a$. Note that $\hat{S}_{\mathrm{pre}\,q}$ always is just *one* abstract state because an assignment point always has one predecessor.

conditional-true A condition is a boolean expression A condition (i.e. a boolean expression) $cond$ is assumed to have an abstract version $\widehat{cond}(\hat{S}_q)$ where each arithmetical expression E is replaced by $\hat{E}(\hat{S}_q)$, each relational operator $\sim$ replaced by $\hat{\sim}$ and each boolean operator $*$ replaced by $\hat{*}$. At a conditional-true point, the conditional is assumed to be true. The values of the identifiers has not been changed since the last program point, though. So reasonably, at a conditional-true point, an identifier can only have the values it had at the previous program point which satisfies the predicate conditional. Therefore, we have

$$\hat{S}'_q = \bigsqcup\{s | s \sqsubseteq \hat{S}_{\mathrm{pre}\,q} \wedge \widehat{cond}(s) \sqsubseteq\ True\}$$

where q is a conditional-true point.

conditional-false Following the discussion above we let (if q is a conditional false)

$$\hat{S}'_q = \bigsqcup\{s | s \sqsubseteq \hat{S}_{\mathrm{pre}\,q} \wedge \widehat{cond}(s) \sqsubseteq\ False\}$$

merge Finally, at a merge point, each of the "incoming" states may be true, hence:

$$\hat{S}'_q = \bigsqcup \hat{S}_{\mathrm{pre}(q)}$$

if q is a merge point.

The abstract interpretation is then performed by solving the equation:

$$\hat{S}^i = \hat{\tau}(\hat{S}^i)$$

where $\hat{S}^0 = \bot$, $\hat{S}^i = \hat{\tau}(\hat{S}^{i-1})$. In order for this function to have a solution is that $\hat{\tau}$ is a monotone function. That is, $S \sqsubseteq S' \Rightarrow \hat{\tau}(S) \sqsubseteq \hat{\tau}(S')$ for all S and S'. This equation can be solved by computing the sequence $(S^{\,i})^{i \in \mathbb{N}} = \hat{S}^0 \sqsubseteq \hat{S}^1 \sqsubseteq \hat{S}^2 \sqsubseteq ...$[1] until $\hat{S}^{i+1} = \hat{S}^i$ for some i.

5.3 An example

As an illustration of abstract interpretation, consider the flowchart in Figure 3. If the abstract domain of signs (see Section 3.1) is used then the abstract transition function yields the following system of equations.

[1] or equivalently $\hat{\bot} \sqsubseteq \hat{\tau}(\hat{\bot}) \sqsubseteq \hat{\tau}(\hat{\tau}(\hat{\bot})) \sqsubseteq ...$

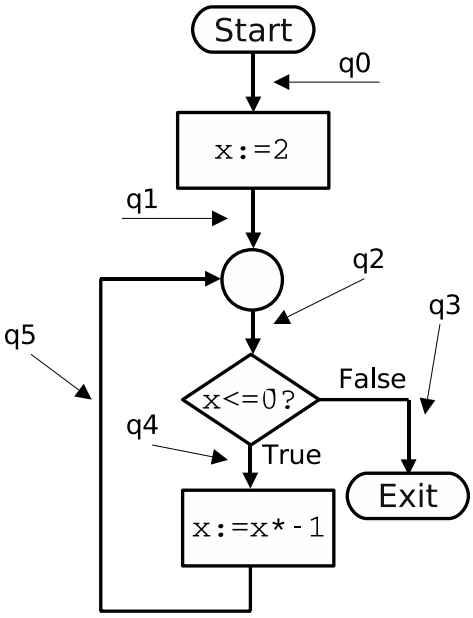

Figure 3: Flowchart

$$\hat{S}_{q_0} = [x \mapsto \top]$$
$$\hat{S}_{q_1} = \hat{S}_{q_0}[x \mapsto \alpha(2)] = \hat{S}_{q_0}[x \mapsto +]$$
$$\hat{S}_{q_2} = \hat{S}_{q_1} \sqcup \hat{S}_{q_1}$$
$$\hat{S}_{q_3} = \bigsqcup\{s | s \sqsubseteq \hat{S}_{q_2} \wedge \hat{S}_{q_2}(x) \hat{\leq} 0 \sqsubseteq False\}$$
$$\hat{S}_{q_4} = \bigsqcup\{s | s \sqsubseteq \hat{S}_{q_2} \wedge \hat{S}_{q_2}(x) \hat{<} 0 \sqsubseteq True\}$$
$$\hat{S}_{q_5} = \hat{S}_{q_4}[x \mapsto \hat{S}_{q_4}(x) \hat{*} \alpha(-1)] = \hat{S}_{q_4}[x \mapsto \hat{S}_{q_4}(x) \hat{*} -]$$

To solve this system of equations, we begin by setting $\hat{S}_{q_n} = \bot$ for all program points q_n. Then the left hand side is replaced by the right hand side of the system. The process is iterated until the left hand side equals the right hand side. The iterations is shown in figure 4. A fixed point is reached after seven iterations (iterations seven and eight yields the same result).

5.4 Widening/Narrowing

There is a risk that the fix point calculation will never terminate even though abstract semantics is used. This is because, the lattice of an abstract domain might contain infinite chains. If the chain $(\hat{S}^i)^{i \in \mathbb{N}}$ never stabilizes to a constant, then the calculation will never terminate. Sometimes the calculation terminates but takes too much time. There are solutions to these problems however. In order to ensure (or in some cases just speed up) termination, a *widening* operator is needed. This operator ensures termination but

Program point	Iter 0	1	2	3	4	5	6
q_0	$\bot$	$[x \mapsto \top]$	$[x \mapsto \top]$	$[x \mapsto \top]$	$[x \mapsto \top]$	$[x \mapsto \top]$	$[x \mapsto \top]$
q_1	$\bot$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto +]$
q_2	$\bot$	$\bot$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto \top]$	$[x \mapsto \top]$
q_3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$[x \mapsto -]$
q_4	$\bot$	$\bot$	$\bot$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto +]$	$[x \mapsto \top]$
q_5	$\bot$	$\bot$	$\bot$	$\bot$	$[x \mapsto \top]$	$[x \mapsto \top]$	$[x \mapsto \top]$

Figure 4: Iterations of applying the abstract transition function

may over approximate the result. To give a somewhat better approximation there is also a somewhat dual concept of *narrowing*.

Definition 5.1. *Let L be a lattice and $\nabla \colon L \times L \to L$ be a function. Then ∇ is a* widening operator *if*

$$x \sqsubseteq x \nabla y \text{ and } y \sqsubseteq x \nabla y \text{ for all } x, y \in L$$

and that for every increasing chain $(y^i)^{i \in \mathbb{N}}$ the increasing chain $(y^i_\nabla)^{i \in \mathbb{N}}$ defined by

- $y^0_\nabla = y_0$
- $y^{i+1}_\nabla = y^n_\nabla \nabla y_n$

after a while stabilizes to a constant. That is, $\exists j \forall k (k \geq j \Rightarrow x_k = x_j)$.

If the original chain is replaced with the widening chain in the fix point equation then termination is guaranteed. Let $(\hat{S}^i)^{i \in \mathbb{N}}$ be a chain of abstract states (as described in Section 5.2). If the abstract domain contains infinite chains, then, to ensure termination, the abstract domain must have a widening operator for the abstract contexts in the domain. Then we can construct the chain $\hat{S}^m_\nabla$ and a solution to the equation $\hat{S}^m_\nabla = \hat{\tau}(\hat{S}^m_\nabla)$ is always guaranteed.

Because of the loss of information when using widening, it is possible to get a slightly better result with a narrowing operator [CC77]. When the over approximated fixed point has been reached, the result can gradually be better with narrowing. The narrowing is defined in the same way as widening but as a decreasing chain that is always an over-approximation of the fixed point.

6 Existing domains

In this section some existing (and used) abstract domains will be presented. Only numerical domains will be considered here though, in fact we shall assume that $\mathcal{V} = \mathbb{Z} \cup \{\bot\}$ in all domains.

6.1 Non-relational domains

6.1.1 Property domains

The simplest domains are those that only collect simple properties of variables. The sign domain presented in Section 3.1 is a property domain describing the integer property of being positive, negative or zero. Another property domain is the *parity domain*. It is a simple lattice where

$$\bot \sqsubseteq \text{odd} \sqsubseteq \top$$

and

$$\bot \sqsubseteq \text{even} \sqsubseteq \top$$

It is easy to lift ordinary operators like $+$ and $*$ to this abstract domain. We give an example of the abstract addition operator $\hat{+}$:

$$\begin{array}{lcl} even \,\hat{+}\, even & = & even \\ even \,\hat{+}\, odd & = & odd \\ odd \,\hat{+}\, even & = & odd \\ odd \,\hat{+}\, odd & = & even \\ \top \hat{+}\, x = x \,\hat{+}\, \top & = & \top \text{ if } x \neq \bot \\ \bot \hat{+}\, x = x \,\hat{+}\, \bot & = & \bot \end{array}$$

6.1.2 The interval domain

The idea of abstracting values to properties will not give very precise information about the values - a far more precise domain is the interval domain [CC77]. In this domain, a set of integers are approximated by the least single interval enclosing them. A set Z of integers is approximated with $[a, b]$ where $a = \min Z_{i \in \mathbb{N}}, b = \max Z_{i \in \mathbb{N}}$. However, sometimes it is not possible to know about the upper/lower limit of a set of integers so we will also allow intervals $[a, b]$ where a may be $-\infty$ and/or b may be ∞.

This domain is indeed a lattice. We introduce the ordering $\sqsubseteq$ such that $[a, b] \sqsubseteq [c, d]$ iff the whole interval $[a, b]$ is contained inside of $[c, d]$ or equally, $a \geq c$ and $b \leq d$. The top element $\top$ is then the interval $(-\infty, \infty)$ since each interval is contained inside it. The bottom element $\bot$ is an empty interval which contains no elements. This lattice has infinite height and contains infinite chains and therefore it needs a widening operator, which can be found in [CC77].

This lattice is fast and easy to deal with, but it has some severe drawbacks. For instance, consider the set $\{-100, 5000\}$ (cardinality$= 2$). This set of two values will be approximated as a much greater set $[-100, 5000]$ (cardinality$= 5101$) which is an immense over-approximation. A summary of important operators of this domain is given in Appendix A.

6.1.3 The congruence domain

It is possible to generalize the parity domain discussed above to the congruence domain [Gra89]. In the congruence domain a set of values are mapped to the least common set where each variable has the form $a\mathbb{Z} + b$. By least common set we mean that the value of a shall be maximized. Thus, an abstract element represents an infinite, discrete, equal-spaced set of integers. The parity domain is a special case of this domain where a is fixed to 2. A summary of some important operators for this domain is given in Appendix A. The congruence domain and the interval domain models quite orthogonal concepts. The interval domain gives upper and lower bounds for a variable's possible values, whereas the congruence domain measures density of its values. As we shall see in Section 7 it is possible to combine properties of different abstract domains to obtain new, more precise domains. There is much more on the congruence domain in Section 9.

6.2 Relational domains

6.2.1 The polyhedral domain

The domain of convex polyhedra was suggested by Cousot & Halbwachs [CH78] for which a full description is given in [Hal79]. This is one of the most commonly used numerical relational abstract domain. A polyhedron in n-space (where n is the number of variables in the program) describes limits for the variables. A polyhedron can be described on the form:

$$A\boldsymbol{x} \leq \boldsymbol{b}$$

where A is an $n \times n$ matrix and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. A polyhedron can be described as above; as a system of linear equations, that is, a finite intersection of half-spaces. The polyhedron represents all integers enclosed within the polyhedron, and each dimensional axis represent a program variable.

The linear equation is not the only way to represent a convex polyhedron however. In fact we will need an alternative representation to use convex polyhedra as an abstract domain. Consider a set of points $S \in \mathbb{Z}^n$. The *convex hull* of these points is the least polyhedron enclosing all points. A subset of S entirely determines the convex hull, and these points are called *vertices*. If we have no bounds for the set S (i.e. S is infinite), the vertices will not provide enough information to represent a unbounded polyhedron. A vector for which any positive combination of it still is contained inside the polyhedron is called a *ray*. A set of rays R and the a of vertices V, is enough information to determine a polyhedron. The reason for having two representation is for implementation purposes. The intersection of two polyhedra (represented by linear constraints) is easily constructed by adding the constraints from one polyhedron to the other, but if we are computing the convex hull of two polyhedra (represented by vertices and rays), the new set of vertices is simply the union of the two vertex sets and the same for the rays. Therefore it seems that a conversion between the two representations is needed. The polyhedral domain gives really good limits for variables as long as the relations are linear, sadly it is very complex and as the number of variables grow it is hard to deal with.

6.2.2 The linear congruence domain

It is simple to add relationships between variables to the non-relational congruence domain above [Gra91]. An integral lattice (of dimension equal to the number of variables) has the form:

$$A\boldsymbol{x} \equiv a\mathbb{Z} + b$$

An element of this domain is the set of variables that can be written as a linear combination of the row vectors of A, where all coefficients are positive. This is a slower but more precise version of the non-relational congruence domain.

6.2.3 Weakly relational domains

A compromise between the efficiency of non relational domains and the precision of relational ones are so-called *weakly relational domains* [Min02] which preserve some relationships between variables on the form $x - y \in C$, where C is a non-relational domain. A special case of such domains are the difference-bound matrices where one can detect relations of the form $x - y \leq b$ or $\pm x \leq b$. The octagon domain [Min06],

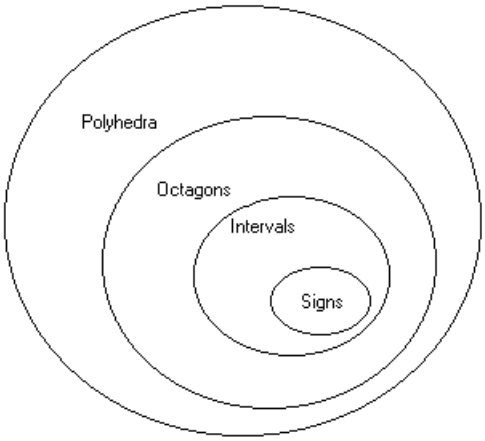

Figure 5: The relation among domains

which is a restriction of the convex polyhedral domain, is an extension of difference-bound matrices. The relation with respect to precision among the comparable domains signs, intervals, octahedra and polyhedra is shown in figure 5.

6.2.4 Combinations & operators

It is possible to combine quite orthogonal concepts from above to get interesting domains, for example the $\mathcal{Z}$-polyhedra. $\mathcal{Z}$-polyhedra are exactly the intersection between the linear congruence domain and the polyhedral domain (which is called the reduced product, see Section 7.2). Thus, the domain is integral lattices bounded by polyhedra.

The Trapezoidal domain [Mas92] has mixed the idea of intervals and linear congruences. This domain describes sets on the form $ux + vy \in [a, b] \bmod q$. Some of the ideas can be formalized, which we will see in the following section.

7 Manipulating Abstract Domains

It is possible to generalize the concept of abstract domains further in order to find interesting theoretic properties of them. In this section abstract domains will be seen as elements of the set of *all* abstract domains. As will be revealed, that set is a lattice which allows us to perform lattice operators on abstract domains. The actual theory and of the ideas presented here are out of scope of this thesis, so the ideas will merely be introduced with a pointer to the original text.

7.1 Closure operators

Definition 7.1. *Let D be a lattice and $\varphi\colon D \to D$ an operator on D. Then φ is a (upper)* closure operator *on D if:*

- $d \sqsubseteq \varphi(d)$ *(Extensiveness)*
- $d \sqsubseteq d' \Rightarrow \varphi(d) \sqsubseteq \varphi(d')$ *(Monotonicity)*

- $\varphi(d) = \varphi(\varphi(d))$ ***(Idempotence)***

A closure operator maps elements of certain subsets to the most abstract element of that subset (d is more abstract than d' iff $d' \sqsubseteq d$). The image of D through a closure operator φ constitutes by itself a complete lattice. So the image of a closure operator can be seen as an abstraction of D. Now, let $uco(D)$ be the set of closure operators on D, then $uco(D)$ is a lattice $(uco(D), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x)$. We give the definition of $\sqsubseteq$:

$$\varphi \sqsubseteq \varphi' \Leftrightarrow \varphi(d) \sqsubseteq \varphi'(d) \text{ for all } d \in D$$

The reason for introducing closure operators is that Galois insertions (α, γ) will not alone suffice to show that the set of abstract domains is a lattice. A Galois insertion is dependent on the internal representation of elements in an abstract domain, thus making it hard to see differences and similarities. Therefore, we need a more general definition. It can be proved [DP90] that the map $\gamma \circ \alpha$ is a closure operator on C. Thus, any abstract domain defined by a Galois insertion is isomorphic to a closure operator on the concrete semantic domain. The result is that $uco(D)$, where D is the concrete semantic domain, and the set of abstract domains on D is isomorphic, and the set of abstract domains in itself is indeed a complete lattice!

Since the set of abstract domains over the concrete semantic domain is a lattice, it is natural to ask how the lattice operators are defined on this lattice. First, we observe that the bottom element is the identity closure operator which corresponds to no abstraction at all. The top element is the closure operator mapping each element to the top element, yielding no information at all (i.e the abstract domain with only one element).

7.2 Direct and Reduced product

Consider two abstract domains A and B and suppose that we want a domain that captures the properties from both domains. A straightforward approach to do this is to consider elements $a \in A \times B$. Further we define the concretization map for this domain as

$$\gamma_{A\times B}\langle a, b\rangle = \gamma_A(a) \cap \gamma_B(b)$$

The domain $A \times B$ is called the *direct product* of A and B. This domain forms a Galois connection but in general not a Galois insertion.

Remember from Section 3.2 that a Galois insertion is a Galois connection where $\alpha \circ \gamma = id_A$. Every Galois connection can be reduced to a Galois insertion by introducing a equivalence relation on elements of A, such that $a \sim a'$ iff $\alpha(a) = \alpha(a')$ and then work with the abstract elements modulo $\sim$. This procedure is known as *reducing* a domain.

The *reduced* product is the domain that remains after reducing the direct product domain. The reduced product corresponds to the lattice operator $\sqcap$ of the $uco(C)$ lattice. To implement an abstract domain which is a direct product of two domains is straight forward; all operators are performed component wise (this holds for the abstraction map as well). Thus, the direct product of two domains can be implemented by running the analysis with the two domains simultaneously and returning the reduced intersection of the result. More on analysis with direct and reduced product can be found in [CMB$^+$93].

7.3 Complement

It is natural to ask whether it is a inverse concept of the reduced product. That is, given a domain, we want to know which domains it is a reduced product of. This can be interesting because it gives a possibility to decompose a domain which was not originally defined by composition. however, quotients does not always exist. Before we make a formal definition of a complement, we need to define a few notions.

A *pseudo complement* for an element a in a lattice is the most general element b such that $a \sqcap b = \bot$. That is the pseudo complement a^* for a is:

$$a^* = \sqcup\{x | a \sqcap x\}$$

A domain is called *pseudo complemented* iff every element in the domain has a pseudo complement. We can now give a formal definition of complements of abstract domains (according to [CFG$^+$97]).

Definition 7.2. *Let B be an abstract domain abstracting another abstract domain A (that is, $A \sqsubseteq B$). If A is pseudo complemented, then the* complement *of B relative to A is the pseudo complement B^*. We denote this complement $A \sim B$.*

If $\times$ represents the reduced product, then $(A \sim B) \times A = B$.

7.4 Powerset domains

The sign domain maps sets of integer to the set $\hat{\mathcal{V}}_{Sign}$. From this abstraction we can define a new, more precise abstraction $\mathcal{P}(\hat{\mathcal{V}}_{Sign})$. By defining $\gamma : \mathcal{V} \rightarrow \mathcal{P}(\hat{\mathcal{V}})$:

$$\gamma_{\mathcal{P}(Sign)}(A) = \bigcup\{\gamma_{Sign}(a) | a \in A\}$$

This has a well defined meaning. For instance, the element $\{+, 0\}$ corresponds to non-negative values and $\{-, +\}$ a non-zero element. Simply put, a power set domain is the lifting of an abstract set $\hat{\mathcal{V}}$ to its power set $\mathcal{P}(\hat{\mathcal{V}})$. A power set domain is more precise but slower than the original. Since power set domains often has infinite chains, suggestions on how to construct widenings on them are given in [BHRZ03].

7.5 An example

A domain that is using the concept of the reduced product is the $\mathcal{Z}$-polyhedra [SPR00]. The $\mathcal{Z}$-polyhedra domain is the reduced product of the convex polyhedra and the linear congruence domain. This analysis can give very precise results since it is relational and uses two very different analyses at the same time. However, since an implementation must run both the already costly polyhedral domain and the linear congruence domain, this domain is *very* costly.

8 Software libraries

In this section a few (free) software libraries that implements some of the classic abstract domains will be presented. All libraries are working under GCC (GNU Compiler Collection).

8.1 Convex polyhedra libraries

The *Parma Polyhedra Library* (PPL) [Par06] is a C++ library that implements the convex polyhedra domain. The library is well documented and the source code is free. The regular polyhedra operators (including two widening operators) are available. It also features an implementation of the power set domain of convex polyhedra as well as special cases where the polyhedrons are not closed. Documentation for users as well as developers is provided. *New Polka* [New06]is another software library implementing convex polyhedra (mainly based on Polylib below). Polylib [Pol06] is a C library with free source code. This library also implements the convex polyhedra domain, or rather, the convex polyhedra power set domain, since it is implemented as finite unions of polyhedra. But the linear congruence domain is implemented as well. Finally, operators for the $\mathcal{Z}$-polyhedra are included. A newer version is available as "PolyLib". This library is mainly focused on the power set domain of convex polyhedra.

8.2 The Octagon Abstract Domain Library

This is a fully documented C library for UNIX systems. It contains operators for the octagon abstract domain (including widenings).

9 The Congruence domain revisited

In the following sections, the congruence domain will be studied in detail. Granger's original work will be briefly summerized and then some enhancements for low-level analysis in this domain will be introduced.

9.1 Mathematical preliminaries

The details about the abstract operators of the congruence domain requires some mathematical preliminaries. The congruence domain and its operators are founded on number theory, so some knowledge in elementary number theory is expected from the reader. It will be assumed that the reader is familiar with terms such as *greatest common divisor* and *least common multiple* as well as basic modular arithmetic and the division algorithm. If not it can be found in any elementary textbook in algebra or number theory, for example[Kos02]. Since bit-operators will be developed in the following sections, some preliminaries of bit-strings will also be given here.

9.1.1 Number Theory

We will use the common definition of modulo in this thesis.

Definition 9.1. $a \equiv b \mod m$ *iff* $m|(a-b)$

A direct consequence of this is $a \equiv b \mod m$ iff $\exists q(a = mq + b)$.

9.1.2 Bits

Bit strings are ordinary numbers represented in base 2. However, there are several ways to deal with negative integers. The most common way (and the way we are going to use) to represent negative numbers as bit strings is *twos complement*. To acquire the negative representation of a given integer through two's complement is to "flip" all bits

of the given integer and then add one. That is if the number 2 is represented by the byte 00000010, then -2 is 11111110. With this representation it is the most significant bit that decides whether an integer is negative or not. This bit will be referred to as the *signed bit*.

9.2 Important definitions

The congruence domain consists of abstract values denoted, $a\mathbb{Z} + b$, Where $b \in \mathbb{Z}$ and $a \in \mathbb{N}$. We will call a the *modulo* and b the *remainder*.

The lattice operators $\sqcup$ and $\sqcap$ are defined as follows (due to [Gra89]).

$$(a\mathbb{Z} + b) \sqcup (a'\mathbb{Z} + b') = gcd\{a, a', |b - b'|\}\mathbb{Z} + min\{b, b'\}$$
$$(a\mathbb{Z} + b) \sqcap (a'\mathbb{Z} + b') = lcm\{a, a'\}\mathbb{Z} + b'' \text{ if } b \equiv b' \mod gcd\{a, a'\}$$
$$(a\mathbb{Z} + b) \sqcap (a'\mathbb{Z} + b') = \bot \text{ otherwise.}$$

where $b'' \equiv b \mod a$ and $b'' \equiv b' \mod a'$. Other cases follows from the lattice axioms.

The abstraction and concretization maps for this domain are defined as follows:

$\alpha(\{n\}) = 0\mathbb{Z} + n$
$\alpha(M) = gcd\{|b - b'||b, b' \in M\}\mathbb{Z} + min\{b|b \in M\} \quad \gamma(a\mathbb{Z} + b) = \{an + b|\forall n \in \mathbb{Z}\}$
$\gamma(\top) = 1\mathbb{Z} + 0 = \mathbb{Z}$
$\gamma(\bot) = \varnothing$

In words the set $\gamma(a\mathbb{Z} + b)$ contains all integers that are congruent to b modulo a. Often it will be convenient to talk about the values in the set $\gamma(a\mathbb{Z} + b)$. These values will be referred to as elements of an *abstract value* and we will often use the abusive notation $d \in a\mathbb{Z} + b$ rather than the more correct $d \in \gamma(a\mathbb{Z} + b)$. In the rest of the text, whenever *abstract value* is mentioned, we shall mean an abstract value of the *congruence domain*.

9.2.1 Equivalence classes

The above given α and γ are not an Galois insertion because γ is not injective. This is shown by the following example:

$$\gamma(2\mathbb{Z} + 1) = \{2n + 1|\forall n \in \mathbb{Z}\} = \{2n + 3|\forall n \in \mathbb{Z}\} = \gamma(2\mathbb{Z} + 3)$$

The domain is easily reduced by introducing the equivalence class $a\mathbb{Z} + b\tilde{a}'\mathbb{Z} + b'$ iff $\gamma(a\mathbb{Z} + b) = \gamma(a'\mathbb{Z} + b')$. However, we want to have a representing value that represents the equivalence class. A good choice of representing value is to choose the *smallest positive* b in the equivalence class. This implies that if $a \neq 0$ then $b < a$. Any abstract value with this property is said to be on *standard form*. This also implies that b is the least positive integer in $\gamma(a\mathbb{Z} + b)$.

9.2.2 Singleton and non-singleton values

In the special case where $a = 0$ in an abstract value $a\mathbb{Z} + b$, we verify that

$$\gamma(0\mathbb{Z} + b) = \{0n + b | \forall n \in \mathbb{Z}\} = \{b\}$$

This means that the abstract value actually represent a single value. In the case where a is greater than 0, the set $\gamma(a\mathbb{Z} + b)$ is an infinite set.

9.3 Modification of the domain

Bit-strings with the signed bit set can always be interpreted in two ways – as signed and as unsigned which results in two different integer values. Depending on how we interpret a bit-string, we will get different abstract values when applying the abstraction function to it. To give an example, if we are working with 8-bit strings, then the string 10011100 represents 156 as signed, but as unsigned it represents the number -100. It can be shown that the difference between the signed and the unsigned value is always 2^B (where B is the number of bits we work with). In this section we will develop a solution to this problem by abstracting both the signed and the unsigned values to get the abstract value. Let B be a bit-string of length n, and let B_{signed} be its signed interpretation and $B_{unsigned}$ its unsigned interpretation. Then

$$\begin{aligned}
\alpha(\{B_{signed}, B_{unsigned}\}) &= \alpha(B_{signed}) \sqcup \alpha(B_{unsigned}) \\
&= (0\mathbb{Z} + B_{signed}) \sqcup (0\mathbb{Z} + B_{unsigned}) \\
&= gcd(0, B_{signed} - B_{unsigned})\mathbb{Z} + min(B_{signed}, B_{unsigned}) \\
&= gcd(0, 2^n)\mathbb{Z} + B_{signed} = 2^n\mathbb{Z} + B_{signed} = 2^n\mathbb{Z} + B_{unsigned}
\end{aligned}$$

The last equality demonstrates the fact that the signed and the unsigned value of a bit string always are congruent modulo 2^n. When abstracting a value which has its signed bit set we say that the abstract value is *uncertain*. If the signed bit is not set, then B_{signed} and $B_{unsigned}$ are the same and α behaves it did in the definition in Section 9 (because n will be zero). We will denote an uncertain abstract value with an asterisk $(2^n\mathbb{Z} + B*)$. Uncertain abstract values will make the domain loose precision quite considerably. A single value would be abstracted to a singleton value, but uncertain values are abstracted to an infinite non-singleton value. However, since we have assumed that each concrete value is represented with n bits, the only values in $\gamma(2^n\mathbb{Z} + B*)$ that can be represented with n bits is B_{signed} and $B_{unsigned}$. These two values in $\gamma(2^n\mathbb{Z} + B*)$ is the only ones we'd be interested in and we wish to eliminate one of them. In a later phase of the analysis, it might be possible to determine which interpretation of the bitstring that is the appropriate one (for instance by observing which operators the bit-string is applied to). In this case, an uncertain abstract value can be turned into a non-uncertain one. This will be formalized using two operators, signed and $\mathrm{unsigned}$ that takes an abstract value (uncertain or not) and returns an abstract value which is *not* uncertain.

$$\begin{aligned}
\mathrm{signed}(2^n\mathbb{Z} + B*) &= 0\mathbb{Z} + 2^n - B \\
\mathrm{signed}(a\mathbb{Z} + b) &= a\mathbb{Z} + b \\
\mathrm{signed}(\bot) &= \bot \\
\mathrm{unsigned}(2^n\mathbb{Z} + B*) &= 0\mathbb{Z} + b \\
\mathrm{unsigned}(a\mathbb{Z} + b) &= a\mathbb{Z} + b \\
\mathrm{unsigned}(\bot) &= \bot
\end{aligned}$$

where $2^n\mathbb{Z} + B*$ is any uncertain value and $a\mathbb{Z} + b$ is any non-uncertain value. As can be seen, non-uncertain values are not effected by signed or $\mathrm{unsigned}$. Note that it is safe to apply signed or $\mathrm{unsigned}$ to an uncertain value *only* if the interpretation is known (that is, that the value is applied to some operator with known sign).

9.4 An example

We give an example using one of NIC:s operators (see Section 1.2). Let $\widehat{\texttt{s\_mul}}$ be the abstract version of `s_mul`. The operator `s_mul` corresponds to *signed* multiplication, hence it is safe to use the signed operator on both sides like below.

$$\mathrm{signed}(a\mathbb{Z} + b)\ \widehat{\texttt{s\_mul}}\ \mathrm{signed}(a'\mathbb{Z} + b')$$

Note that

$$\mathrm{signed}(\alpha(\{B_{signed}, B_{unsigned}\})) = \alpha(B_{signed})$$

and that

$$\mathrm{unsigned}(\alpha(\{B_{signed}, B_{unsigned}\})) = \alpha(B_{unsigned})$$

10 Abstract operators of the congruence domain

In the following sections will abstract versions of an array of common integer operators be presented. The arithmetic abstract operators was already introduced by Philippe Granger in [Gra89]. These operators will briefly be summarized in the following section.

10.1 Arithmetic Operators

Abstract versions of common arithmetic operators are defined and proved to be correct for the congruence domain in [Gra89]. All abstract operators in this domain are *strict*, that is, whenever the bottom value($\perp$) occurs in some argument, then the result is the bottom value as well. All theorems in this section are due to [Gra89].

The abstract operators for addition, subtraction and multiplication follow:

Theorem 10.1. *Let $D = a\mathbb{Z} + b$ and $a'\mathbb{Z} + b'$ be two non-bottom abstract values of the congruence domain. Then*

$$D\hat{\pm}D' = gcd(a, a')\mathbb{Z} \pm min(b, b')$$

is a correct approximation of addition and subtraction.

Theorem 10.2. *Let $a\mathbb{Z} + b$ and $a'\mathbb{Z} + b'$ be two non-bottom abstract values. Then*

$$(a\mathbb{Z} + b)(a'\mathbb{Z} + b') = gcd\{aa', ab', a'b\}\mathbb{Z} + bb'$$

is a correct approximation of multiplication.

Integer division is defined as $\lfloor \frac{n}{m} \rfloor$. That is the largest integer that is less than or equal to the real quotient. The result of abstract integer division between two non-bottom, non-singular abstract values in the congruence domain is unfortunately always the top element. Formally:

Theorem 10.3. *Let D and D' be two non-bottom, non-singular abstract values. Then the best approximation for*

$$\left\lfloor \frac{D}{D'} \right\rfloor \text{ is } \mathbb{Z}$$

Theorem 10.4. *Let $D = a\mathbb{Z}+b$ and $D' = a'\mathbb{Z}+b'$ be two non-bottom abstract values. Then*

$$D \widehat{mod} D' = gcd\{a, a', b'\}\mathbb{Z} + b$$

is the best correct abstraction for modulo.

10.2 Conditionals and relations

The set $\gamma(a\mathbb{Z} + b)$ of an non-singular abstract value contains an infinite amount of values. It contains values which are greater than any arbitrary number and also values that are smaller than any arbitrary number. That is for all integers n there exists values $d_0, d_1 \in \gamma(a\mathbb{Z} + b)$ such that $n < d_0, n > d_1$. This implies that relational operators is not working well on the congruence domain. In fact, the abstract version of $<, >, \leq$ and $\geq$ is the mapping $\lambda(x, y).\top$ if x and y are non-bottom, non-singleton values.

10.3 Bit operators

To be able to create a lower-level analysis, the lower-level bit-operators also have to be abstracted. No bit-operators were developed in Granger's work, therefore these operators will be defined and proved to be correct in this section. The abstract bit operators that will be developed is: NOT,AND,OR,XOR, left shift, right shift, truncating and extension.

In general the abstract congruence domain doesn't fit too well when working with bits. When the abstracting the arithmetic operators, one can rely on that every bit-string has an *interpretation* as one or two integers. Bitwise operators, however, need to know about the actual bits, not just the representation as an integer. In general, for any fixed bit, say bit k there are values $d_0, d_1 \in \gamma(a\mathbb{Z} + b)$ such that bit k is one in d_0 and zero in d_1. That is, unless $a = 2^m$ for some $m > 0$. Before introducing the abstract bit-operators, some important lemmas and definitions will be presented.

In the following lemmas we shall interpret $\gamma(a\mathbb{Z} + b)$ as a set of bit-strings rather than a set of integers.

Lemma 10.5. *The set $2^N\mathbb{Z}$ for some $N \in \mathbb{N}$ is the set of bit-strings in which (at least) the N least significant bits are all zeros.*

Proof. Let $N = 0$. Then $2^0\mathbb{Z} = \mathbb{Z} =$ all bit-strings. So the case $N = 0$ is trivial, therefore assume that $N = K$ and $2^{K-1}\mathbb{Z}$ is the set of all bit-strings in which the $K-1$ least significant bits are all zeros. Then $2^K\mathbb{Z} = 2 * 2^{K-1}\mathbb{Z}$. Now take $b \in 2^{K-1}\mathbb{Z}$ arbitrary. Multiplication by two is equivalent to left shifting a bit-string. But since b ends with $K - 1$ zeros, it will after a left shift (multiplication by two) "insert" another zero from the left and the K least significants bits will be zeros, and the theorem is proved. □

From this lemma it follows that if B is an N-digit bit string, then B interpreted as an integer belongs to $2^K\mathbb{Z}$ for some $K \leq N$ if and only if B ends in K zeros. Another important consequence, if an abstract value is on standard form (that is $b < a$ if $a \neq 0$)

and if a is a power of two, all integers in the abstract value will end in the bit string b. That is, all elements in $\gamma(2^m\mathbb{Z} + b)$ has its m least significant bits equal to b.

Lemma 10.6. *Let a be any integer. Let a be written on the form $a = 2^k c$ for some $k \in \mathbb{N}$, such that $2 \nmid c$ (which is obviously possible for all integers), then 2^k is the least significant non-zero digit of a, when a is interpreted as a bit-string.*

Proof. Since c is odd, and it is a well known fact that an integer, interpreted as a bit string, has its least significant bit set. Multiplication with 2^k corresponds shifting this least significant bit k steps, making 2^k the least significant non-zero digit. □

Lemma 10.7. *Let a be an integer represented by N bits. Then $gcd(2^N, a) = 2^k$ where 2^k is the least significant non-zero bit of a.*

Proof. We rewrite a as $2^k c$, where $k \in \mathbb{N}$, $2 \nmid c$, as Lemma 10.6 suggests, and we have:

$$gcd(2^N, a) = gcd(2^N, 2^k c)$$

Clearly $k \leq N$, so we may write

$$gcd(2^N, 2^k c) = 2^k gcd(2^{N-k}, c)$$

Since c is odd and 2^{N-k} is one or even, we have $gcd(2^{N-k}, c) = 1$ and hence draw the conclusion:

$$gcd(2^N, a) = gcd(2^N, 2^k c) = 2^k$$

Lemma 10.7 tells us that 2^k is the least significant non-zero bit (interpreted as a bit-string) of a. □

The results above motivates to introduce a *weakening* operator, which is defined as:

$$\xi(a\mathbb{Z} + b) = gcd(2^N, a)\mathbb{Z} + b \text{ if } a \neq 0$$

where N equals the number of bits the values are assumed to have.

Lemma 10.8. *Let $a\mathbb{Z} + b$ be an abstract value, then:*

$$a\mathbb{Z} + b \sqsubseteq \xi(a\mathbb{Z} + b)$$

Proof. By definition $a\mathbb{Z} + b \sqsubseteq a'\mathbb{Z} + b'$ iff $a'|a$ and $b \equiv b' \mod a'$. Again, by definition we have that $gcd(2^N, a)|a$. And since b is unaffected by the ξ operator, we naturally have that $b \equiv b' \mod gcd(2^N, a)$. □

The idea behind the weakening operator is that the modulo of a value applied to the operator is guaranteed to be a power of two (by Lemma 10.7). However, the value gained after applying the weakening operator is more abstract and using it will loose precision. The motivation to use it is to simplify information about bits in abstract values.

10.3.1 The NOT operator

The operator bitwise NOT has an easy interpretation in the decimal system as well, which we are going to use. if n is a B-bit number, then $NOT(n) = 2^B - n - 1$. From this, we can verify that

$$NOT(an + b) = 2^B - (an + b) - 1 = an + 2^B - b - 1$$

for all $n \in \mathbb{Z}$. This implies that that the abstract NOT-operator will be

$$\widehat{NOT}(a\mathbb{Z} + b) = a\mathbb{Z} + (2^B - b - 1)$$

where it is assumed that we are working with B-bit numbers.

10.3.2 The AND, OR and XOR operators

The abstract versions of the bitwise AND, OR and XOR operators need information about the bits in the abstract values. Therefore, the weakening operator introduced in Section 10.3 will be used.

Lemma 10.9. *Let $a\mathbb{Z} + b$ and $a'\mathbb{Z} + b'$ be two non-bottom abstract values. And let*

$$\xi(a\mathbb{Z} + b) = 2^n\mathbb{Z} + b \text{ and } \xi(a'\mathbb{Z} + b') = 2^m\mathbb{Z} + b'$$

Then the following are correct approximations of AND, OR and XOR.

$$(a\mathbb{Z} + b)\widehat{\text{AND}}(a'\mathbb{Z} + b') = 2^{min(n,m)}\mathbb{Z} + (b \,\text{AND}\, b')$$
$$(a\mathbb{Z} + b)\widehat{\text{OR}}(a'\mathbb{Z} + b') = 2^{min(n,m)}\mathbb{Z} + (b \,\text{OR}\, b')$$
$$(a\mathbb{Z} + b)\widehat{\text{XOR}}(a'\mathbb{Z} + b') = 2^{min(n,m)}\mathbb{Z} + (b \,\text{XOR}\, b')$$

Proof. All integers in $2^n\mathbb{Z} + b$ ends with the bit string b which contains n bits (it follows from Lemma 10.5). All integers in $2^m\mathbb{Z} + b'$ ends with the bit string b' which is m bits long. This means that we can compare the bits from the $min(n, m)$ last bits and perform the desired bit operator on these known bits. □

10.3.3 Left shifting

We will define a left shift as multiplication by 2 (this holds for unsigned as well as signed values). The left shifting operator is denoted $<<$. Hence, the operation $a << n$ corresponds to multiply a with 2^n. In the analysis however, we have abstract values on the left hand as well as on the right hand. In the following let $A = a\mathbb{Z} + b$ and $A' = a'\mathbb{Z} + b'$, and let $D = \gamma(A) = \{an + b | \forall n \in \mathbb{Z}\}$ and $D' = \gamma(A') = \{a'm + b' | \forall m \in \mathbb{Z}\}$. The concrete left shift on the sets D and D' is for all $n \in \mathbb{Z}$:

$$D << D' = 2^{D'}D = \{2^{D'}(an + b) | n \in \mathbb{Z}\} = \{2^{D'}an + 2^{D'}b | n \in \mathbb{Z}\} \quad (3)$$

The left shift operator is assumed to take a non-negative right hand side, so in the abstract version we shall only regard the non-negative integers of D' (if negative integers of D' were allowed, then $2^{D'}$ would not always be an integer).

Equation (3) actually describes a set of abstract values, namely

$$2^{D'}a\mathbb{Z} + 2^{D'}b$$

This implies that a correct definition of the abstract left shift operator is:

$$A\widehat{<<}A' = \bigsqcup\{2^{d'}a\mathbb{Z} + 2^{d'}b | d' \in D'\}$$

The least upper bound of these abstract values constitutes a correct approximation of them.

$$\bigsqcup\{2^{d'}a\mathbb{Z} + 2^{d'}b | d' \in D'\} = gcd\{2^{d'}a, |2^{d'_0}b - 2^{d'_1}b| \mid d', d'_0, d'_1 \in D'\}\mathbb{Z} + \min\{2^{d'}b\}$$
$$gcd\{2^{d'}a, b|2^{d'_0} - 2^{d'_1}| \mid d', d'_0, d'_1 \in D'\}\mathbb{Z} + \min\{2^{d'}b\}$$

Since b' is the least positive number in D', it is clear that $gcd\{2^{D'}a\} = 2^{b'}a$, and also that $\min\{2^{D'}b\} = 2^{b'}b$. Hence, we have a correct approximation as:

$$A\widehat{<<}A' = gcd(2^{b'}a, b|2^{d'_0} - 2^{d'_1}|)\mathbb{Z} + 2^{b'}b$$

for all $d'_0, d'_1 \in D'$ and all $n \in \mathbb{Z}$. Now we need to find out which values that may occur in the set $\{b|2^{d'_0} - 2^{d'_1}| \colon d'_0, d'_1 \in D'\}$. Without restriction we can assume that $d'_1 \leq d'_0$. Then we can write $b|2^{d'_0} - 2^{d'_1}|$ as

$$2^{d'_1}b|2^{d'_0 - d'_1} - 1| = 2^{d'_1}b|2^{ka'} - 1| \tag{4}$$

for some $k \in \mathbb{N}$ and for all $d'_1 \in D'$. Since the set (4) is used in a gcd computation, the only relevant value of $2^{d'_1}$ will be the least one. Trivially the least $d'_1 \in D'$ is b', therefore we have can define the abstract operator as

$$A\widehat{<<}A' = gcd(2^{b'}a, b2^{b'}|2^{ka} - 1|)\mathbb{Z} + 2^{b'}b$$

for some $k \in \mathbb{N}$.

Theorem 10.10. *(Fermat)*
If $n = m \mod p-1$ where p is a prime, then $z^n = z^m \mod p$ for all integers $z \in \mathbb{Z}$.

Now consider only the factor $|2^{ka'}-1|$ from (4). We know that $ka' = 0 \mod a'$ for all $k \in \mathbb{Z}$. As a result of the preceding theorem we have: $2^{ka'} = 2^0 = 1 \mod a'+1$ if $a'+1$ is prime. So if $a'+1$ is prime, it follows from (4) that $2^{ka'} - 1 = 0 \mod a'+1$. That is to say that $a' + 1$ divides $2^{ka'} - 1$ if $a' + 1$ is prime (whatever k might be). Apart from this, it is very hard to determine which values that might occur in the set $2^{ka'} - 1$.

Theorem 10.11. *Let $A = a\mathbb{Z} + b$ and $A' = a'\mathbb{Z} + b'$ be two non-bottom abstract values. Then a correct abstraction of the left shift operator is:*

$$A\widehat{<<}A' = (2^{b'}gcd(a, b(a' + 1)))\mathbb{Z} + 2^{b'}b$$

if $a' + 1$ is prime and

$$A\widehat{<<}A' = (2^{b'}gcd(a, b))\mathbb{Z} + 2^{b'}b$$

if $a' + 1$ is not a prime.

Proof. We know that $A\widehat{<<}A' \sqsubseteq gcd(2^{b'}a, b2^{b'}|2^{ka'} - 1|)\mathbb{Z} + 2^{b'}b$
$= gcd(2^{b'}a, b2^{b'}|2^{ka'} - 1|)\mathbb{Z} + 2^{b'}b$, for some $k \in \mathbb{N}$. We will be satisfied with a common divisor instead of the greatest common divisor in this case. We will use the

fact that $gcd(x,y)|gcd(x,yz)$ for any x,y,z to get a common divisor. This means that $gcd(x,y)$ is a common divisor for x and yz but not necessarily the greatest one. Therefore we will, unless $a'+1$ is prime, drop the factor $|2^{ka'}-1|$ from the gcd calculation and obtain a common divisor (we will do this due to the complicated nature of the factor). A consequence of dropping a factor from the calculation is that we will loose some precision; we don't have the best approximation for the left shift. We can conclude:

$$A\widehat{<<}A' \sqsubseteq gcd(2^{b'}a, b2^{b'}(a'+1))\mathbb{Z} + 2^{b'}b$$

if $a'+1$ is prime and

$$A\widehat{<<}A' \sqsubseteq gcd(2^{b'}a, b2^{b'})\mathbb{Z} + 2^{b'}b$$

if not. The factor $2^{b'}$ is common in both gcd computations and can be factorized. □

10.3.4 Right shifting

To right shift one step is equivalent to make an integer division by 2 (this holds for signed as well as unsigned numbers). To simplify the proofs, the weakening operator will be applied to all abstract elements. That is, we will approximate $a\mathbb{Z}+b\widehat{>>}a'\mathbb{Z}+b'$ with $\xi(a\mathbb{Z}+b)\widehat{>>}\xi(a'\mathbb{Z}+b')$. This implies that we can assume that all abstract values have the form $2^n\mathbb{Z}+b$.

Lemma 10.12. *Let $2^n\mathbb{Z}+b$ be a non-bottom abstract value and let k be a singular abstract value. Then*

$$2^n\mathbb{Z}+b\widehat{>>}k = 2^{n-k}\mathbb{Z}+(b>>k) \text{ if } k<n$$

is a correct approximation of the right shifting operator. If $k \geq n$ then the best approximation (if the abstract values are weakened) is $\mathbb{Z}$.

Proof. The set $2^n\mathbb{Z}+b$ is the set of bit strings ending with the exact bit-string b according to lemma 10.5. If we right-shift such a string k steps, then of course it will end with $b>>k$. Since the ending of a string (bits on the right side of the n:th bit) is all we know about the string, right-shifting will force to reduce 2^n to 2^{n-k}. If k is equal to n we will have $\mathbb{Z}$; and any right-shift of $\mathbb{Z}$ will result in $\mathbb{Z}$. □

Theorem 10.13. *Let $2^n\mathbb{Z}+b$ and $2^{n'}\mathbb{Z}+b'$ be two non-bottom, non-singular abstract values. Then the best approximation (if the abstract values are weakened) of*

$$2^n\mathbb{Z}+b\widehat{>>}2^{n'}\mathbb{Z}+b'$$

is $\mathbb{Z}$.

Proof. Since $2^{n'}\mathbb{Z}+b'$ is non-singular, there is a value $d \in 2^{n'}\mathbb{Z}+b'$ such that $d \geq n$. According to lemma 10.12 the best approximation of $2^n\mathbb{Z}+b>>d$ if $d \geq n$ is $\mathbb{Z}$. Since $2^n\mathbb{Z}+b>>d$ must be included to the resulting set, the theorem follows. □

10.3.5 Truncating

Definition 10.14. *Let B be a N-digit number. Then $\mathrm{TRUNC}_n(B)$, for a non-negative integer n such that $n \leq N$ is defined as the n least significant digits of B.*

Lemma 10.15. *Let B be a N-digit number. Then $(B \equiv \mathrm{TRUNC}_n(B)) \bmod 2^n$.*

Proof.

$$(B \equiv \mathrm{TRUNC}_n(B)) \bmod 2^n \Leftrightarrow 2^n | (B - \mathrm{TRUNC}_n(B)) \Leftrightarrow B - \mathrm{TRUNC}_n(B) \in 2^n\mathbb{Z}$$

Which is by Lemma 10.5 equivalent to say that $B - \mathrm{TRUNC}_n(B)$ ends in n zeros. Since the n least significant digits of B by definition are the same as of $\mathrm{TRUNC}_n(B)$, the least significant digits of the difference between the least significant digits of of B and $\mathrm{TRUNC}_n(B)$ is a string of n zeros. □

Theorem 10.16. *Let $D = a\mathbb{Z} + b$ be a non-bottom abstract value. Then*

$$\widehat{\mathrm{TRUNC}}_n(a\mathbb{Z} + b) = gcd(2^n, a)\mathbb{Z} + b$$

is a correct approximation of the truncating operator.

Proof. From Lemma 10.15 we know that $\mathrm{TRUNC}_n(ak + b) \equiv ak + b \mod 2^n$ for every $k \in \mathbb{Z}$. This is implies that:

$$\mathrm{TRUNC}_n(ak + b) \in 2^n\mathbb{Z} + ak + b$$

This holds for all $k \in \mathbb{Z}$, that is

$$\widehat{\mathrm{TRUNC}}_n(a\mathbb{Z} + b) \in 2^n\mathbb{Z} + a\mathbb{Z} + b = gcd(2^n, a)\mathbb{Z} + b$$

□

10.3.6 Extensions

The extension of all bit-strings in $\gamma(a\mathbb{Z} + b)$ from n to m bits where $n < m$ does not effect the values of $\gamma(a\mathbb{Z} + b)$. Therefore, in mathematical terms, the abstract version of ”zero extend” and ”sign extend” is the identity mapping on abstract values.

11 SWEET and the congruence domain

As mentioned, important parts of SWEET are based on abstract interpretation. SWEET currently uses the interval domain (see Section 6.1.2) to find upper and lower bounds of variables. The interval domain is not a very precise domain, and higher precision of the domain is desired. Since the congruence domain complements the interval domain very well, it seemed a good reason to implement this domain as well. Another reason for selecting the congruence domain was that the pointer analysis of SWEET often yields over approximations as pointers in real programs often ”steps” in non-unit steps, and analysis with a congruence domain can easily detect such behavior.

11.1 Implementation

A prototype of the congruence domain for the SWEET tool has been implemented. SWEET is written in standard C++ and the current version works with the GCC compiler. SWEET is implemented so that new abstract domains easily can be added to the code. Abstract domains are arranged in subclasses of a general class of abstract values. Important direct subclasses of this general class are the classes for abstract integers, abstract pointers and abstract floats. The class for abstract interval integers (i.e. an implementation of the interval domain) is directly inherited from the class of abstract

integers and so is the congruence domain. The abstract domains are then arranged in an abstract superclass of the domain and then subclasses where $8, 16$ and 32-bits values has its own class. Since NIC does not provide as good type information as C does, a special class for unspecified values is also needed. If no type information is provided, a value is treated as "unspecified" until type information can be determined. A special class for bottom values is inherited directly from the class of abstract values. All bottom values are instances of this class, no matter which domain is used.

An implementation of an abstract domain is mainly an implementation of abstract versions of NICs operators. All abstract operators exist in the general class of abstract values, but most of them are virtual [2].

When this is written, the implementation is not fully integrated with SWEET and no results about the implementation can be presented here.

12 Conclusions and discussion

We have presented abstract interpretation in a formal way by presenting the collecting semantics and how to make an abstract version of it by using abstract domains. A short summary of the most common numerical abstract domains has been given, this summary includes (among others), the interval domain, the congruence domain and the polyhedral domain. The abstract congruence domain (invented by Philippe Granger) has been studied in more detail. This thesis has contributed by enhancing the congruence domain to lower-level analyses. The enhancement is done by introducing abstract bit-operators for the abstract congruence domain. To be able to find abstract bit operators we have introduced a weakening operator that obtains information of a set of bit-strings by studying the least significant bit that is common for the whole set. A suggestion to a solution to the signed/unsigned-problem has also been proposed. This solution is based on the idea of temporary abstracting different interpretations of a bit-string to one abstract (but still correct) value and later, when the appropriate interpretation of the bit-string has been determined, the temporary abstraction can be resolved into a more accurate one. A prototype written in C++ of the congruence domain for the SWEET tool has been implemented. The prototype implements Granger's original abstract operators as well as the bit-operators introduced in this thesis, while not fully integrated with SWEET, no results about the implementation is given in this thesis.

Abstract interpretation is a relatively new science, and it is being developed constantly. As more domains are being developed and more will be known about abstract domains in general it will probably be even more useful in the future. The choice of abstract domain for a specific need will probably be easier as tools for custom made abstract domains probably will be richer.

The congruence domain is a harsh domain to deal with, since it quite often happens that a set of values is abstracted to the top element. This is because almost all operators includes a gcd computation which will return the top element if two relatively prime numbers are in the same set. Thus, this domain works very poorly when used alone. However, combining it with other domains with for instance the reduced product can be very effective. The domain is fast and does not need any widening or narrowing operators so it will probably not slow the analysis down so much.

[2] i.e. they contain no implementation and have to be overridden.

13 Future Work

- As abstract interpretation presumably will increase in popularity, more abstract domains will probably be invented. The choice of an abstract domain is a delicate balance between precision and complexity. The key to find a suitable abstract domain will be to keep an eye on the research in the area.
- The operators of the congruence domain are correct, but several of them are not the *best* approximation of the concrete operator. It is certainly possible to increase the precision of some of the operators.
- As the congruence domain in itself is even less precise than the interval domain, the idea behind the implementation was to combine it with the interval domain using the direct product. The direct product is a simple thing to implement. The combined domain could then win some precision on reducing it.
- When the prototype implementation has been fully integrated with SWEET it would be interesting with an evaluation of the results.
- Other abstract domains would certainly interesting to use in SWEET, for instance some relational ones. A relational domain would not be that easy to implement, however, because NIC uses pointers, arrays and structures, and it may be troublesome to determine what can be regarded as a variable.

References

[BHRZ03] R. Bagnara, P. Hill, E. Ricci, and E. Za. Precise widening operators for convex polyhedra, 2003.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4^{th} ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CFG+97] Agostino Cortesi, Gilberto Filé, Roberto Giacobazzi, Catuscia Palamidessi, and Francesco Ranzato. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.

[CH78] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[CMB+93] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *PEPM '93: Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 194–205, New York, NY, USA, 1993. ACM Press.

[DP90] B. Davey and H. Priestley. *introduction to lattices and order*. Cambridge University Press, 1990.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden, June 2003.

[GLSB03] Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In Bob Werner, editor, *Proc. 8^{th} IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Guadalajara, Mexico, 2003. IEEE.

[Gra89] Philippe Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, pages 165–199, 1989.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Gus00] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Dept. of Information Technology, Uppsala University, May 2000.

[Hal79] Nicholas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. Thèse de 3eme cycle, Univ. de Grenoble, March 1979.

[Kos02] Thomas Koshy. *Elementary Number Theory with Applications*. HARCOURT/ACADEMIC PRESS, 2002.

[Mas92] Francois Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 226–235, New York, NY, USA, 1992. ACM Press.

[MdH06] Mälardalen University WCET project homepage, 2006. `www.mrtc.mdh.se/projects/wcet`.

[Min02] Antoine Miné. A few graph-based relational numerical abstract domains. In Manuel V. Hermenegildo and German Puebla, editors, *Proc. 9th Int. Symposium on Static Analysis*, volume 2477 of *Lecture Notes in Comput. Sci.*, pages 117–132, Madrid, Spain, September 2002. Springer-Verlag.

[Min06] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.

[New06] New Polka website, 2006. `http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html`.

[NNH05] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2^{nd} edition*. Springer, 2005. ISBN 3-540-65410-0.

[Par06] PPL: The Parma Polyhedra Library, 2006.
http://www.cs.unipr.it/ppl/.

[Pol06] Polylib - A library of polyhedral functions , 2006.
http://www.ee.byu.edu/beta/faculty/wilde/polyhedra.html.

[SPR00] Kumar Nookala Sunder Phani and Tanguy Risset. a library for z-polyhedral operations, 2000. technical report 1330, institut de recherche en informatique et systemes aleatories.

Appendix A Domain summaries

In this appendix a brief summary of important operators from two important numerical abstract domains is given. For more information about notation, see the sections corresponding to the domains. In all domains we will assume that $\hat{+}$ and $\hat{*}$ are strict, that is $a\hat{+}\bot = \bot\hat{+}a = \bot$ and $a\hat{*}\bot = \bot\hat{*}a = \bot$ for all $a \in \mathcal{A}$. Moreover, $\gamma(\bot) = \varnothing$ for all domains.

Appendix A.1 The interval domain

An element of the interval domain is denoted $[a, b]$ where $a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}$. All operators are taken from [CC77].

Operator	Interval domain
$\sqcup$	$[a,b] \sqcup [a',b'] = [\min(a,a'), \max(b,b')]$
$\sqcap$	$[a,b] \sqcap [a',b'] = [\max(a,a'), \min(b,b')]$
$\sqsubseteq$	$[a,b] \sqsubseteq [a',b'] \Leftrightarrow [a,b] \subseteq [a',b'] \Leftrightarrow a \geq a' \wedge b \leq b'$
∇	$[a,b] \nabla [a',b'] = [\mathrm{cond}(a \leq a', a, -\infty), \mathrm{cond}(b \geq b', b, \infty)]$
Δ	$[a,b] \Delta [a',b'] = [\mathrm{cond}(a = -\infty, a', a), \mathrm{cond}(b = \infty, b', b)]$
$\hat{+}$	$[a,b] \hat{+} [c,d] = [a+c, b+d]$
Elements	
$\top$	$(-\infty, \infty)$
$\bot$	$\varnothing$
Galois connection	
α	$\alpha(k) = [k,k]$
γ	$\gamma([a,b]) = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$

Appendix A.2 The congruence domain

An element of the congruence domain is denoted $a\mathbb{Z} + b$. The operators are taken from [Min02].

Operator	Congruence
$\sqcup$	$(a\mathbb{Z}+b) \sqcup (a'\mathbb{Z}+b') = \gcd\{a, a', b-b'\}\mathbb{Z} + \min(b,b')$
$\sqcap$	$(a\mathbb{Z}+b) \sqcap (a'\mathbb{Z}+b') = \mathrm{cond}(b \equiv b' \bmod \gcd(a,a'), \mathrm{lcm}(a,a')\mathbb{Z} + b'', \bot)$
$\sqsubseteq$	$(a\mathbb{Z}+b) \sqsubseteq (a'\mathbb{Z}+b') \Leftrightarrow a' \mid a$ and $b \equiv b' \bmod a'$
∇	N/A
Δ	N/A
$\hat{+}$	$(a\mathbb{Z}+b)\hat{+}(a'\mathbb{Z}+b') = \gcd(a,a')\mathbb{Z} + (b+b')$
Elements	
$\top$	$\mathbb{Z}$ (that is, $a = 1, b = 0$)
$\bot$	$\varnothing$
Galois connection	
α	$\alpha(k) = 0\mathbb{Z} + k$
γ	$\gamma(a\mathbb{Z}+b) = \{ak+b \mid k \in \mathbb{Z}\}$ if $a \neq 0$ $\gamma(a\mathbb{Z}+b) = \{b\}$ if $a = 0$