URL: http://www.elsevier.nl/locate/entcs/volume59.html 15 pages

Interpreting Abstract Interpretations in Membership Equational Logic

Bernd Fischer and Grigore Roşu

Research Institute for Advanced Computer Science
Automated Software Engineering Group
NASA Ames Research Center
Moffett Field, California, 94035, USA
{fisch,grosu}@ptolemy.arc.nasa.gov

Abstract

We present a logical framework in which abstract interpretations can be naturally specified and then verified. Our approach is based on membership equational logic which extends equational logics by membership axioms, asserting that a term has a certain sort. We represent an abstract interpretation as a membership equational logic specification, usually as an overloaded order-sorted signature with membership axioms. It turns out that, for any term, its least sort over this specification corresponds to its most concrete abstract value. Maude implements membership equational logic and provides mechanisms to calculate the least sort of a term efficiently. We first show how Maude can be used to get prototyping of abstract interpretations "for free." Building on the meta-logic facilities of Maude, we further develop a tool that automatically checks an abstract interpretation against a set of user-defined properties. This can be used to select an appropriate abstract interpretation, to characterize the specific loss of information during abstraction, and to compare different abstractions with each other.

1 Introduction

Abstract interpretation [5] has become a unifying framework for a variety of program analysis techniques. It is also increasingly finding application in other areas of computer science, e.g., model checking [2,13] and theorem proving [12]. Its appeal is precisely the involved abstraction which simplifies calculation and search, thus allowing us to efficiently find approximate but correct solutions to problems which may otherwise be too hard.

In practice, however, finding the right abstraction for a problem at hand is often all but trivial. Different abstractions offer different trade-offs between efficiency and approximation which can often be validated and evaluated only with a prototype implementation of the abstract interpretation. A

©2001 Published by Elsevier Science B. V.

more definitive evaluation of the abstraction, e.g., in the form of a correctness or completeness proof, even requires support for formal verification.

In this paper, we are addressing these validation and verification issues. We show how abstract interpretation can be specified and interpreted in membership equational logic (MEL) [11,1], an expressive logic which generalizes the various forms of equational logics used in algebraic specification techniques. With this re-interpretation, we can use the Maude system [4,3] as environment to execute the specification of an abstraction, or in other words, we get a prototyping environment "for free." We further developed a tool which uses Maude's meta-logic capabilities to verify abstract interpretations against properties that are also specified as MEL sentences. More precisely, we provide a Maude module that allows us to write commands like

```
Maude> reduce verify PROPERTIES on ABSTRACTION .
```

where `ABSTRACTION` is a specification of an abstract interpretation and `PROPERTIES` is another MEL specification. `verify_on_` returns a list of warnings showing all sentences in `PROPERTIES` that are violated by the abstract interpretation, together with (abstract) counterexamples. We can see at this stage three applications of such a tool, depending on what the specification `PROPERTIES` actually represents:

- If `PROPERTIES` is a set of desired properties an abstract interpretation should have in order to be employed in a certain context, then we can use our tool to retrieve from a library abstract interpretations that satisfy them;
- If `PROPERTIES` is an axiomatic presentation of the concrete domain, then we can use the tool to see how much and what kind of information the abstraction looses and then decide if it is acceptable for a specific task;
- If `PROPERTIES` is another abstract interpretation of the same concrete domain, then an empty list of warnings means that `ABSTRACTION` refines this second abstraction.

The basic idea of our approach is relatively straightforward. If we think of an abstract interpretation as a lattice, then a *sort* is associated to each node in the lattice and a *subsort* is associated to each edge. The tables defining the abstract operations are replaced by a series of overloaded operators, while properties that cannot be captured by tables are declared via MEL sentences.

```
sorts Bottom Neg Pos Real .
subsorts Bottom < Neg Pos < Real .
op  _+_  : Neg Neg -> Neg .
op  _+_  : Pos Pos -> Pos .
op  _*_  : Neg Pos -> Neg .
op  _*_  : Neg Neg -> Pos .
op  _/_  : Real Real -> Real .
op  _/_  : Real Zero -> Bottom .
mb X / X : Pos-one .
```

is a fragment of the usual sign abstraction [1]. The abstract value of an ex-

[1] The precise formalism used here will be explained later.

pression in the concrete domain then corresponds to its least sort over the specified signature; an efficient automatic least sort calculation is provided by Maude.

Our approach to abstract interpretation in MEL is more logic-oriented than the usual lattice-oriented approach. We advocate a *loose semantics* for the specifications corresponding to abstract interpretations. The intended abstract interpretation is then *just one* model of the specification, namely that one that adds exactly one new element to each carrier, besides those resulting from the inclusion of carriers required by the subsort declarations. Another model is the one that structures the concrete domain into subsets of elements of the same "type". This model is thus similar to the powerset construction usually used as concrete domain in the lattice-oriented approach.

The ultimate goal of abstract interpretations is to simplify reasoning about the concrete domain. We thus prefer to add to the specification as many sentences that are valid properties of the concrete domain as needed rather than to add exactly those that make the intended abstract interpretation an initial model. Our approach requires no structure on the concrete domain and the abstract domain need not be a complete lattice or a lower semilattice. Instead, the associated specification is expected to be regular [1] in order for Maude to correctly compute the least sort of a term. Interestingly, the fact that the partial order on sorts is a lattice tends to be sufficient for regularity in practice.

Our work here represents only first steps towards a full re-interpretation of abstract interpretations in MEL. Our approach is currently restricted to finite number of sorts, or abstract types, and we cannot yet handle fixpoint operators. These restrictions will be subject to further investigation. The focus of this paper is on the tool aspects rather than on the theoretical aspects.

The remainder of the paper is organized as follows. In the next section we briefly introduce the fundamental notions and notations of membership equational logic and the Maude-system, as far as they are required for our purposes. We then show in Section 3 how abstract interpretations can be seen as membership algebras. Section 4 shows how a MEL specification can be associated to an abstract interpretation and how abstract types can be automatically calculated at the level of that specification. Section 5 finally discusses how abstract interpretations can be verified in the framework proposed in this paper.

2 Membership Equational Logic and Maude

Membership equational logic [11,1] extends many-sorted equational logic [8] with membership assertions $t : s$ stating that a term t belongs to a sort s. It subsumes a wide variety of specification formalisms, including order-sorted [7,9] and partial equational logics. Despite its generality, it still enjoys the good properties of equational logics: it is simple, efficiently implementable, and

admits sound and complete deduction as well as free models. In this section we informally present membership equational logic and some of the notational conventions used, referring to [11,1,4,3] for a comprehensive exposition. We assume basic familiarity with many-sorted equational logic.

In membership equational logic, sorts are grouped in *kinds* and operations are defined on kinds. A signature Ω consists of a set S of *sorts*, a set K of kinds, a map $\pi\colon S \to K$, and a $K^\star \times K$-indexed set $\Sigma = \{\Sigma_{w,k} \mid (w,k) \in K^\star \times K\}$ of *operations*. An Ω-algebra is a Σ-algebra A together with a subset $A_s \subseteq A_k$ for each $k \in K$ and each $s \in \pi^{-1}(k)$. For any K-indexed set of variables X, $T_\Sigma(X)$ is the usual (K,Σ)-algebra of terms. The sentences of MEL generalize the conditional equations $(\forall X)\ t = t'$ **if** C of equational logics by allowing membership assertions. They are universally quantified Horn clauses of the form $(\forall X)\ t : s$ **if** C. In both cases, the condition C is a finite set $\{u_1 = v_1, ..., u_n = v_n, t_1 : s_1, ..., t_m : s_m\}$ and $t, t', t_1, ...t_m, u_1, v_1, ..., u_n, v_n$ are terms in $T_\Sigma(X)$. If $n = m = 0$ then the sentence is called *unconditional* or *atomic*.

We discuss only satisfaction of atomic sentences in this paragraph; the general case follows easily. For an Ω-algebra A and an assignment $a\colon X \to A$ we use $a^\star\colon T_\Sigma(X) \to A$ to denote the unique extension of a to a morphism of (K,Σ)-algebras. We then say that A *satisfies* $(\forall X)\ t = t'$ (or $(\forall X)\ t : s$) if and only if for each assignment a we also have that $a^\star(t) = a^\star(t')$ (or $a^\star(t) \in A_s$). A MEL *specification* or *theory* is a pair (Ω, Γ), where Γ is a set of Ω-sentences, and it defines a class of Ω-algebras (those that satisfy it) denoted $\mathbf{Alg}_{(\Omega,\Gamma)}$.

The MEL proof theory is derived from the standard proof theory of equational logic. Its distinctive characteristic is that it allows to infer the memberships of terms to sorts in addition to the standard equalities of terms. Given a specification (Ω, Γ), there are two inference rules that facilitate that. One is a modification of the *modus ponens* rule to deduce a membership from a (conditional) sentence in Γ once its condition has been proven. The second rule is an extensionality rule asserts that equal terms have the same sort.

$$\textsf{Membership:}\qquad \frac{\Gamma \vdash_\Omega (\forall X)\ t = t' \qquad \Gamma \vdash_\Omega (\forall X)\ t : s}{\Gamma \vdash (\forall X)\ t' : s}$$

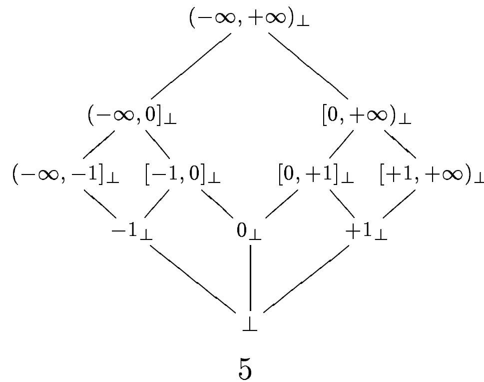
Maude [4,3] is a high-performance rewrite system in the OBJ family [10] that supports both membership equational logic and rewriting logic. Its current version processes up to 3 million rewritings per second on standard hardware. We use Maude notation in this paper not only to represent abstract interpretations in membership equational logic but also to verify them. A few notational conventions are introduced next. Equations and conditional equations are declared via the keywords `eq` and `ceq`, respectively; membership and conditional membership assertions are declared with `mb` and `cmb`. Operations can be declared using *mix-fix* notation. They can also be overloaded; however, this is only syntactic sugar for appropriate conditional membership assertions. For example, `_*_ : Pos Neg -> Neg` can be replaced by `cmb X * Y : Neg if X : Pos and Y : Neg`, where `_*_` is defined on appropriate kinds. Declarations of the form `var X : S` are used to introduce variables; their scope is bounded

by the enclosing *module*, introduced by `fmod ... end`. Kinds need not to be declared explicitly. They are automatically calculated as the connected components of the partial order defined by subsort declarations.

A typical problem of specification formalisms that allow order-sorting and operator overloading is that some terms may have various correct sorts at the same time. The possible sorts of a term can be deduced using the complete deduction system of MEL, i.e., $sorts(t)$ is the set $\{s \in S \mid \Gamma \vdash_\Omega (\forall X)\ t : s\}$. A specification (Ω, Γ) is called *regular* if and only if for each term t, $sorts(t)$ is either empty or has a minimal element w.r.t. the subsort relation. A detailed discussion on regularity can be found in [1], together with decidability results and various syntactic criteria that imply regularity. Maude implements some of these criteria. We can thus assume that each valid specification is regular and that we can calculate the least sort of any term efficiently.

3 Abstract Interpretations as Ω-Algebras

To illustrate our approach, we describe a generalized sign/range abstraction for the real numbers, in which both the positive and the negative numbers are split into *large* and *small*. This interpretation is a simplified version of a more complex one that we implemented in a program synthesis system for the statistical data analysis domain [6]. In this domain, probabilities (i.e., the interval $[0, 1]$) and thus the real numbers "0" and "1" are of specific interest. In order to handle division by zero appropriately, we need to make an assumption about the real numbers. We can consider division either as a partial function and leave the set of real numbers unchanged, or as a total function, in which case we need to add a new element to the real numbers which represents undefinedness. For simplicity, we choose the second assumption; note that MEL also supports partiality, so our choice does not reflect a limitation of the framework. Let $\perp$ be the special symbol denoting "undefined"; we use the same symbol $\perp$ for the set $\{\perp\}$, and let $\mathbb{R}_\perp$ denote the set of real numbers plus $\perp$. The abstract domain then consists of eleven intervals of $\mathbb{R}_\perp$ which are partially ordered by inclusion. Each interval of real numbers also contains $\perp$, reflecting the intuition that the result of an expression can be either a number within that interval or undefined. This domain can, as usual, be represented by the following lattice:



We can now provide the abstract versions of operations. The typical, concise way to do it is by tables, as for example the following table for division [2]:

X/Y	$(-\infty,\infty)$	$(-\infty,0]$	$[0,\infty)$	$(-\infty,-1]$	$[-1,0]$	$[0,+1]$	$[+1,\infty)$	-1	0	$+1$	$\bot$
$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,\infty)$	$\bot$	$(-\infty,\infty)$	$\bot$
$(-\infty,0]$	$(-\infty,\infty)$	$[0,\infty)$	$(-\infty,0]$	$[0,\infty)$	$[0,\infty)$	$(-\infty,0]$	$(-\infty,0]$	$[0,\infty)$	$\bot$	$(-\infty,0]$	$\bot$
$[0,\infty)$	$(-\infty,\infty)$	$(-\infty,0]$	$[0,\infty)$	$(-\infty,0]$	$(-\infty,0]$	$[0,\infty)$	$[0,\infty)$	$(-\infty,0]$	$\bot$	$[0,\infty)$	$\bot$
$(-\infty,-1]$	$(-\infty,\infty)$	$[0,\infty)$	$(-\infty,0]$	$[0,\infty)$	$[+1,\infty)$	$(-\infty,-1]$	$(-\infty,0]$	$(+1,\infty]$	$\bot$	$(-\infty,-1]$	$\bot$
$[-1,0]$	$(-\infty,\infty)$	$[0,\infty)$	$(-\infty,0]$	$[0,+1]$	$[0,\infty)$	$(-\infty,0]$	$[-1,0]$	$[0,+1]$	$\bot$	$[-1,0]$	$\bot$
$[0,+1]$	$(-\infty,\infty)$	$(-\infty,0]$	$[0,\infty)$	$[-1,0]$	$(-\infty,0]$	$[0,\infty)$	$[0,+1]$	$[-1,0]$	$\bot$	$[0,+1]$	$\bot$
$[+1,\infty)$	$(-\infty,\infty)$	$(-\infty,0]$	$[0,\infty)$	$(-\infty,0]$	$(-\infty,-1]$	$[+1,\infty)$	$[0,\infty)$	$(-\infty,-1]$	$\bot$	$[+1,\infty)$	$\bot$
-1	$(-\infty,\infty)$	$[0,\infty)$	$(-\infty,0]$	$[0,+1]$	$[+1,\infty)$	$(-\infty,-1]$	$[-1,0]$	$+1$	$\bot$	-1	$\bot$
0	0	0	0	0	0	0	0	0	$\bot$	0	$\bot$
$+1$	$(-\infty,\infty)$	$(-\infty,0]$	$[0,\infty)$	$[-1,0]$	$(-\infty,-1]$	$[+1,\infty)$	$[0,+1]$	-1	$\bot$	$+1$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

However, it is known that tables are often not sufficient to fully describe the intended behavior of abstract operations. For example, there is no possibility to specify that the abstract value of X/X is $+1_\bot$. Hence, the abstraction becomes less precise than one may wish. We will show in the next section how MEL allows us to use membership assertions to specify such properties.

In the context of MEL, it is more convenient to regard the lattice and the tables associated with an abstract interpretation as an Ω-algebra over an appropriate signature Ω. This general algebraic view gives us the freedom to consider an abstract interpretation as "just one particular model" of an (Ω,Γ)-specification. Since a specification can have many different models, this allows us also to relate different abstractions to each other. In the remainder of this section we concentrate on showing how abstract interpretations can be seen as Ω-algebras. In the next sections we will then show how a MEL specification can naturally be associated with an intended abstract interpretation and how we can prove properties at the level of that specification rather than at the level of the abstract model.

The lattice actually gives some structure to the originally flat concrete domain of real numbers. It is now natural to associate a MEL-signature Ω to the lattice above by providing a sort for each node in the lattice and using subsort relations to reflect the lattice structure. In Maude, this is represented as:

```
sorts  Real     --- ≙ (−∞, +∞)⊥
       Neg      --- ≙ (−∞, 0]⊥
       Neg-l    --- ≙ (−∞, −1]⊥
       Neg-s    --- ≙ [−1, 0]⊥
       Neg-one  --- ≙  − 1⊥
       Pos      --- ≙ [0, +∞)⊥
       Pos-l    --- ≙ [+1, +∞)⊥
       Pos-s    --- ≙ [0, +1]⊥
       Pos-one  --- ≙  + 1⊥
       Zero     --- ≙ 0⊥
       Bottom . --- ≙ ⊥
```

[2] For typographical reasons we omit the $\bot$-subscript here.

```
subsorts Neg Pos < Real .
subsorts Neg-one < Neg-l Neg-s < Neg .
subsorts Pos-one < Pos-s Pos-l < Pos .
subsorts Zero < Neg-s Pos-s .
subsorts Bottom < Neg-one Zero Pos-one .
```

Since the lattice is a single connected component, all sorts belong to the same kind k and all operations are declared on that kind, i.e., we have $*, /, +, - : k \times k \to k$ and $- : k \to k$, or again using the Maude notation,

```
ops _*_, _/_, _+_, _-_ : Real Real -> Real .
op  -_ : Real -> Real .
```

Now, the extended real numbers $I\!R_\perp$ can be viewed as an Ω-algebra A as follows. For each sort s we define as its carrier A_s the appropriate subset of $I\!R_\perp$. For example, we define $A_k = A_{\texttt{Real}} = I\!R_\perp$, $A_{\texttt{Pos}} = I\!R^+_\perp = \{x \in I\!R \mid x \geq 0 \text{ or } x = \perp\}$ and so on, until we get to $A_{\texttt{Bottom}} = \{\perp\}$. A's operations are the usual arithmetic operations on $I\!R_\perp$. Notice that most of the carriers are infinite sets.

However, A is not the only interesting Ω-algebra we can construct. Instead of using the set of the real numbers within the respective intervals as carrier we can as well use the intervals themselves. We thus get $A_{\texttt{Bottom}} = B_{\texttt{Bottom}} = \{\perp\}$ and $A_{\texttt{Pos-one}} = B_{\texttt{Pos-one}} = \{+1, \perp\}$ but $B_{\texttt{Pos-s}} = \{[0, +1], 0, +1, \perp\}$ and $B_{\texttt{Pos-s}} = \{[+1, +\infty], +1, \perp\}$ and so on. Hence, B's carrier B_k is finite—in fact, it consists of exactly the same eleven elements as the lattice. B's operations are defined via operator tables, as for example the division table.

The main difference between the two Ω-algebras A and B above is obviously that B is finite. Hence, it is easy to handle in practice but it looses information about the concrete domain of real numbers, so that many useful properties of real numbers do not hold anymore. A just structures the flat concrete domain but still carries all the burden of analyzing the real numbers directly. Ideally, we would want to keep both algebras, as opposed to just discard A and keep B as it often happens in practice. We next show that MEL is a framework under which we can write a (finite) specification which accepts *both* A and B as models. This specification can be seen as a "finite abstraction" of the concrete domain that can contain as many properties of the domain as needed.

4 Interpreting Abstract Interpretations

We present our technique on the example in the previous section. We start in Maude with a module `ABSTRACTION-REAL` by defining the abstract domain as a partial order via sorts and subsorts:

```
fmod ABSTRACTION-REAL is
  sorts    Real Neg Pos Neg-l Pos-l Neg-s Pos-s
           Neg-one Zero Pos-one Bottom .
  subsorts Neg-one < Neg-l Neg-s < Neg < Real .
  subsorts Zero < Neg-s Pos-s .
  subsorts Pos-one < Pos-s Pos-l < Pos < Real .
  subsorts Bottom < Neg-one Zero Pos-one .
```

Note that we can declare more than one subsort in a `subsorts` declaration; a sort is smaller than any other sort occurring between any subsequent `<` symbol and the next period. The concrete real numbers will be provided in Maude 2.0 in a module `MACHINE-REAL`. This module is imported by the abstract interpretation using the `protecting MACHINE-REAL` clause; the reals are abstracted via a map `|_| : MachineReal -> Real` defined using membership assertions:

```
protecting MACHINE-REAL .
op |_| : MachineReal -> Real .
var R  : MachineReal .
 mb  | 0 | : Zero .
 mb  | 1 | : Pos-one .
 mb | -1 | : Neg-one .
cmb  | R | : Neg-l if R <= -1 .
cmb  | R | : Neg-s if R >= -1 and R <= 0 .
cmb  | R | : Pos-s if R >=  0 and R <= 1 .
cmb  | R | : Pos-l if R >=  1 .
```

Note that `| 0 |`, the abstraction of 0, can be of three sorts: `Zero`, `Neg-s` and `Pos-s`. However, Maude automatically calculates for each term its smallest sort, so `| 0 |` has *the* sort `Zero`.

The next step is to abstract the operations defined on the concrete domain, in our case addition, multiplication, negation, subtraction and division. We do this by making use of operator overloading, which basically represents an elegant way to encode the tables often used to define abstract interpretations:

```
op _+_ : Bottom Real -> Bottom .
op _+_ : Real Bottom -> Bottom .
op _+_ : Neg-one Zero -> Neg-one .
op _+_ : Zero Neg-one -> Neg-one .
op _+_ : Zero Zero -> Zero .
op _+_ : Neg-one Pos-one -> Zero .
op _+_ : Pos-one Neg-one -> Zero .
op _+_ : Pos-one Zero -> Pos-one .
op _+_ : Zero Pos-one -> Pos-one .
op _+_ : Neg-l Neg -> Neg-l .
op _+_ : Neg Neg-l -> Neg-l .
op _+_ : Neg-s Zero -> Neg-s .
op _+_ : Zero Neg-s -> Neg-s .
op _+_ : Pos-s Zero -> Pos-s .
op _+_ : Zero Pos-s -> Pos-s .
op _+_ : Pos-l Pos -> Pos-l .
op _+_ : Pos Pos-l -> Pos-l .
op _+_ : Neg Neg -> Neg .
op _+_ : Neg-l Pos-s -> Neg .
op _+_ : Pos-s Neg-l -> Neg .
op _+_ : Pos Pos -> Pos .
op _+_ : Pos-l Neg-s -> Pos .
op _+_ : Neg-s Pos-l -> Pos .
op _+_ : Real Real -> Real .
```

```
op _*_ : Bottom Real -> Bottom .
op _*_ : Real Bottom -> Bottom .
op _*_ : Neg-one Pos-one -> Neg-one .
op _*_ : Pos-one Neg-one -> Neg-one .
op _*_ : Real Zero -> Zero .
op _*_ : Zero Real -> Zero .
op _*_ : Pos-one Pos-one -> Pos-one .
op _*_ : Neg-one Neg-one -> Pos-one .
op _*_ : Neg-l Pos-l -> Neg-l .
op _*_ : Pos-l Neg-l -> Neg-l .
op _*_ : Neg-s Pos-s -> Neg-s .
op _*_ : Pos-s Neg-s -> Neg-s .
op _*_ : Pos-s Pos-s -> Pos-s .
op _*_ : Neg-s Neg-s -> Pos-s .
op _*_ : Neg-l Neg-l -> Pos-l .
op _*_ : Pos-l Pos-l -> Pos-l .
op _*_ : Neg Pos -> Neg .
op _*_ : Pos Neg -> Neg .
op _*_ : Pos Pos -> Pos .
op _*_ : Neg Neg -> Pos .
op _*_ : Real Real -> Real .
```

In this case, we are able to define the intended abstractions of addition and multiplication completely by operator overloading (i.e., in table form). In practice, however, tables are often not sufficient to properly define abstract versions of operators. Some important domain specific properties of operators cannot be specified only looking at the abstract type of their arguments. For example, there is no way to declare that the abstract type of `X/X` is `Pos-one`

by just analyzing the arity (or the table) of (the abstraction of) division. Fortunately, membership equational logic allows us to declare membership assertions, which then can be used to infer the smallest type of an abstract term:

```
op -_ : Bottom -> Bottom .
op -_ : Pos-one -> Neg-one .
op -_ : Zero -> Zero .
op -_ : Neg-one -> Pos-one .
op -_ : Pos-l -> Neg-l .
op -_ : Pos-s -> Neg-s .
op -_ : Neg-s -> Pos-s .
op -_ : Neg-l -> Pos-l .
op -_ : Pos -> Neg .
op -_ : Neg -> Pos .
op -_ : Real -> Real .

vars X Y : Real .

mb X + (- X) : Zero .

op _-_ : Real Real -> Real .
eq X - Y = X + (- Y) .
```

```
op _/_ : Bottom Real -> Bottom .
op _/_ : Real Zero -> Bottom .
op _/_ : Neg-one Pos-one -> Neg-one .
op _/_ : Pos-one Neg-one -> Neg-one .
op _/_ : Zero Real -> Zero .
op _/_ : Pos-one Pos-one -> Pos-one .
op _/_ : Neg-one Neg-one -> Pos-one .
op _/_ : Neg-l Pos-s -> Neg-l .
op _/_ : Pos-l Neg-s -> Neg-l .
op _/_ : Neg-s Pos-l -> Neg-s .
op _/_ : Pos-s Neg-l -> Neg-s .
op _/_ : Pos-s Pos-l -> Pos-s .
op _/_ : Neg-s Neg-l -> Pos-s .
op _/_ : Neg-l Neg-s -> Pos-l .
op _/_ : Pos-l Pos-s -> Pos-l .
op _/_ : Neg Pos -> Neg .
op _/_ : Pos Neg -> Neg .
op _/_ : Neg Neg -> Pos .
op _/_ : Pos Pos -> Pos .
op _/_ : Real Real -> Real .

mb X / X : Pos-one .
mb (- X) / X : Neg-one .
endfm
```

Notice that we can use the equation "`eq X - Y = X + (- Y) .`" to eliminate the abstract subtraction in a way consistent with its concrete definition, thus reducing the size of the specification.

It may be worth mentioning here that Maude provides internal algorithms to insure the regularity of a signature and that it warned us a few times while testing the overloaded signature above that the signature was not regular. After careful examination, we discovered that our "table" was not consistent, in the sense that certain concrete expressions couldn't have a most precise abstract value. For example, we forgot to add the third declaration of `_*_`; then Maude issued a warning saying that the regularity check failed: indeed, the product of negative one and positive one could have any sort in the set {`Neg-s`,`Neg-l`,`Neg`,`Real`}, but this set has no minimal element. The regularity checks thus helped us cover all the situations. We strongly believe that there is a deep relationship between the regularity of order-sorted overloaded signatures and the existence of most concrete abstract values for abstract interpretations where the abstract values form a lattice, and that this relationship is worth investigating.

The module above can now be used as a decision procedure to abstract expressions. The following are a few examples showing how it works:

```
red | 1 - 1 | .  red | 2 - 2 | .
red | 1 | - | 1 | .  red | 2 | - | 2 | .  red | 1 | + | -1 | .

red | 2 | + | -2 | .
red (| -1 | + | 1 |) + | 1 | .
red | -1 | + (| 1 | + | 1 |) .
```

```
red | -2 | / (| 3 | / | 3 |) .
red (| -2 | * | 3 |) / | 3 | .
```

The first five expressions are all correctly abstracted to `Zero`; that is, they all reduce to terms of sort `Zero`, but for different reasons: the first two reduce to `| 0 |` whose sort is `Zero`, the next two first use the equation `eq X - Y = X + (- Y)` to eliminate the subtraction and then use the membership assertion `mb X + (- X) : Zero`, and the last one first calculates the sorts of `| 1 |` and `| -1 |` and then use the declaration `op _+_ : Pos-one Neg-one -> Zero`. The following is the Maude output for the other five reductions:

```
result Real: | 2 | + | -2 |
result Pos-one: (| -1 | + | 1 |) + | 1 |
result Pos: | -1 | + (| 1 | + | 1 |)
result Neg-1: | -2 | / (| 3 | / | 3 |)
result Neg: (| -2 | * | 3 |) / | 3 |
```

The reductions above reflect the intuition that abstract interpretations loose information. For example, since the most specific abstract values of `2` and `-2` are `Pos-1` and `Neg-1`, respectively, we cannot say anything about the abstract type of `| 2 | + | -2 |`, so the returned type is `Real`. The next two reductions show that the abstraction of addition is not associative anymore, which was an unexpected problem that we encountered in our data analysis synthesis system. Finally, the last two reductions show that other expected equational properties of the concrete domain do not hold at the abstract level either, in this case the equation `eq X / (Y / Z) = (X * Z) / Y`.

5 Proving Properties about Abstract Interpretations

Since by abstraction of a concrete domain one actually looses information, a natural and important question is "how much information is lost?". This is a difficult question both to ask and to answer, and probably new theoretical results are behind it; instead of pursuing theoretical research, we decided to take a pragmatic attitude and develop an environment in which one can test an abstract interpretation as a Maude module for various equational or membership properties that it is expected or not to satisfy. For example, suppose that one considers the following properties (also written as a Maude module):

```
fmod PROPERTIES is
  sorts Pos Neg Real .
  subsorts Pos Neg < Real .
  op  -_ : Real -> Real .
  ops (_+_) (_-_) (_*_) (_/_) : Real Real -> Real .
  vars X Y Z : Real .
  eq X + Y = Y + X .                         ***> true
  eq X * Y = Y * X .                         ***> true
  eq X + (Y + Z) = (X + Y) + Z .             ***> false
  eq X * (Y * Z) = (X * Y) * Z .             ***> true
  eq X * (Y + Z) = (X * Y) + (X * Z) .       ***> false
  eq - (- X) = X .                           ***> true
  eq X * (- Y) = - (X * Y) .                 ***> true
  eq (- X) / Y = - (X / Y) .                 ***> true
  eq (X / Y) / Z = X / (Y * Z) .             ***> true
```

```
  eq X / (Y / Z) = (X * Z) / Y .                 ***> false
  eq (X + Y) / Z = (X / Z) + (Y / Z) .           ***> false

  eq X * X = X * X * X * X .                     ***> true

  mb X * X : Pos .                               ***> true
  mb ((X * X) + (Y * Y)) + (X * Y) : Pos .       ***> false
  mb (X - Y) * (Y - X) : Neg .                   ***> false
endfm
```

All one needs to know in order to write up desired properties of an abstract interpretation are the sorts and the operations one wants to refer to. Notice that the sorts `Pos` and `Neg` are only needed for the last three properties, so one could remove them if one was not interested in the membership properties. The fact that the main sort `Real` has 10 subsorts in our previous abstract interpretation is part of the abstraction, not of its desired properties w.r.t. the concrete domain. Since one may want to test the same properties against many different abstract interpretations of the same domain and to chose one that best fits one's purpose, one should refer to as few sorts specific to only one abstraction as possible when writing properties.

One can now test the properties above against the abstract interpretation presented in the previous section using the simple command

```
red verify PROPERTIES on ABSTRACTION-REAL .
```

after first loading the module which defines the operation `verify_on_` and its semantics, whose implementation is briefly presented in the appendix. The {`true, false`} comments in the module above summarize the execution of this command. In fact, for each of the properties above, Maude detected all counterexamples, i.e., possible abstract values for `X, Y, Z`), that violated it; there were 780 counterexamples in total and it took Maude about 30s on a 1.5GHz linux machine to find all of them. For example, the associativity of addition does not hold when `X : Neg-one`, `Y : Pos-one`, `Z : Pos-one`, and the last membership assertion does not hold when `X : Pos-one`, `Y : Pos`, `Z : Zero`. Interestingly, notice the last equation holds for the presented abstract interpretation, even if it does not hold for the concrete domain of real numbers. The lesson that we learned from experiments with this system is that the loss of information due to an abstraction is reflected not only by lost properties of the concrete domain, but also by properties the abstraction has that do not hold in the concrete domain.

The meta level of Maude was needed to implement the above environment, that is, the functionality of the operation `verify_on_`. At the meta level, the modules are just terms of sort `FModule`, so `verify_on_` has the arity `FModule FModule -> Warnings`, where `Warnings` is a list of warnings. In our current implementation, we traverse the first module and for each equation or membership assertion, we take the second module and generate all possible abstract sorts for variables and then calculate the least sorts of the terms involved (two terms if the statement is an equation and one if it is a membership). If the statement is an equation then we output a warning if and only if the two terms

have different least sorts, and if the statement is a membership assertion then a warning is output if and only if the calculated least sort is not smaller than or equal to the sort specified in the membership statement. The interested reader may consult the appendix for more detail on the implementation.

The presented tool is still in an experimentation stage. One of our major goals was to keep it simple and flexible, such that to easily adapt it for concrete uses. We can currently foresee three applications of this tool, depending on what the specification `PROPERTIES` is:

Abstract interpretation retrieval. If it is a collection of required properties manually or automatically generated in the context of a concrete situation where an abstract interpretation is needed, then one can use our tool to automatically retrieve from a library those abstract interpretations that satisfy the requirements;

Comparison with the concrete domain. If it is an axiomatic presentation of the concrete domain, such as a Peano specification of natural numbers, then one can use the tool to understand or test how much and what kind of information a certain abstract interpretation looses and then decide whether it is acceptable or not for a specific task;

Abstract interpretation refinement. If it is another abstract interpretation of the same concrete domain, then an empty list of warnings means that `ABSTRACTION` refines this second abstraction.

6 Conclusion and Future Work

We presented a new way to represent and verify abstract interpretations. We abstracted a concrete domain by a finite membership equational logic specification, which, under loose semantics, also admits as model the common abstract interpretations whose abstract values form a lattice and whose abstract operations are given by a table. This flexible framework allows one to add needed properties of the concrete domain that cannot be captured by tables via equations and membership assertions. Then we showed how one can use Maude, an efficient rewriting engine that implements membership equational logic, to execute such specifications and thus calculate the most concrete abstract type (or the least sort) of an expression. Building on the meta-capabilities of Maude, we further implemented a tool by which one can actually verify an abstract interpretation, given as a MEL specification, against equational or membership properties. Such a tool can be used to retrieve appropriate abstract interpretations from a library, to characterize the loss of information due to an abstraction, and to show refinements of abstract interpretations.

We use this approach in a programming synthesis tool that generates data analysis C programs from stochastic specifications, in order to symbolically prove the conditions of algorithm schemas before they are applied. However, there is much interesting research to be done to prove the viability of our

approach for a broader range of practical situations. For example, we do not know at this stage how to represent infinite abstract interpretations, such as arbitrary integer intervals. Should the sort representation of the lattice be given up? Or should membership equational logic be extended to allow free generated algebras of sorts? What would regularity mean in such a context? Another important aspect of abstract interpretations that we didn't investigate is the computation of fixed points. Is there any relationship between regularity of MEL signatures and fixpoints?

Acknowledgments: We thank José Meseguer and Steven Eker for encouragements and productive discussions. We also thank Steven Eker for providing us an alpha version of Maude 2.0 which uses the membership assertions in calculating the least sort of a term, so we could test our programs and claims.

References

[1] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[2] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[3] Manuel Clavel, F. Durán, Steven Eker, Patrick Lincoln, N. Martí-Oliet, José Meseguer, and J. F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Proc. 10th Intl. Conf. Rewriting Techniques and Applications*, volume 1631 of *Lect. Notes Comp. Sci.*, pages 240–243, Trento, Italy, July 1999. Springer. System Description.

[4] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic, March 1999. Maude System documentation at `http://maude.csl.sri.com/papers`.

[5] Patrick M. Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977. ACM Press.

[6] Bernd Fischer, Johann Schumann, and Thomas Pressburger. Generating data analysis programs from statistical models (position paper). In Walid Taha, editor, *Proc. Intl. Workshop Semantics Applications, and Implementation of Program Generation*, volume 1924 of *Lect. Notes Comp. Sci.*, pages 212–229, Montreal, Canada, September 2000. Springer.

[7] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[8] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

[9] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[10] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[11] José Meseguer. Membership algebra as a logical framework for equational specification. In *Proceedings, WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[12] David A. Plaisted. Theorem proving with abstraction. *Art. Intell.*, 16(1):47–108, 1981.

[13] Willem Visser, S. Park, and John Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000. To appear.

A Some Implementation Details

In this appendix we give a few details on how we implemented the operation `verify_on_ : FModule FModule -> Bool` that verifies an abstract interpretation against user defined properties, both represented as Maude modules. The interestead reader can download all the Maude source code discussed in this paper at `http://ase.arc.nasa.gov/grosu/download/absint-code.html`.

We first implemented some auxiliary operations on top of Maude's meta-level. The code below is not executable; because of space limitations we replaced straightforward code by three dots:

```
fmod META-AUXILIARY is including META-LEVEL .
  ...
  sort VarDeclTuple .
  op |_| : VarDeclSet -> VarDeclTuple .
  ...
  op getVarTuples : FModule VarDeclSet -> VarDeclTupleList .
  var M : FModule .  var Var  : VarDecl .  var Vars : VarDeclSet .
  eq getVarTuples(M, none) = | none | .
  eq getVarTuples(M, Var Vars)
       = mergeVarDeclSet(getVarsFromVar(M, Var),getVarTuples(M, Vars)) .

  op mergeVarDeclSet : VarDeclSet VarDeclTupleList -> VarDeclTupleList .
  op mergeVarDecl : VarDecl VarDeclTupleList -> VarDeclTupleList .
  var Ts  : VarDeclTupleList .
  eq mergeVarDeclSet(none, Ts) = nil .
  eq mergeVarDeclSet(Var Vars, Ts)
       = mergeVarDecl(Var, Ts) mergeVarDeclSet(Vars, Ts) .
  eq mergeVarDecl(Var, nil) = nil .
  eq mergeVarDecl(Var, | Vars | Ts) = | Var Vars | mergeVarDecl(Var, Ts) .
```

```
    op getVarsFromVar : FModule VarDecl -> VarDeclSet .
    vars V S : Qid .
    eq getVarsFromVar(M, var V : S .)
         = getVarsFromNameAndSorts(V, lesserSorts(M, S)) .
    ...
  endfm
```

Notice the use of the builtin operation `lesserSorts` which takes a module and a sort name and gives a set of all smaller sorts. This set of sorts is then used to generate all possible combinations of declarations of variables, which will be subsequently used to calculate the least sorts of the terms involved in properties.

The next module implements the main operation. It imports both the auxiliary operations and a module defining warnings which we skipped; the warnings are currently some adhoc messages concatenated by `@`.

```
  fmod VERIFY-TYPE is including META-AUXILIARY + WARNING .
    op verify_on_ : FModule FModule -> Warnings .
    vars ABS, PROP : FModule .
    eq verify PROP on ABS =
      verifyEqns(getName(ABS), getVarTuples(ABS, getVars(PROP)), getEqns(PROP))
      @ verifyMbs(getName(ABS), getVarTuples(ABS, getVars(PROP)), getMbs(PROP)) .

    op verifyEqns : Qid VarDeclTupleList EquationSet -> Warnings .
    op verifyEqn  : Qid VarDeclTupleList Equation -> Warnings .
    var Name : Qid .  var Ts : VarDeclTupleList .  var Vars : VarDeclSet .
    var Eqn : Equation .  var Eqns : EquationSet .
    eq verifyEqns(Name, Ts, none) = ok .
    eq verifyEqns(Name, Ts, Eqn Eqns) =
      verifyEqn(Name, Ts, Eqn) @ verifyEqns(Name, Ts, Eqns) .
    eq verifyEqn(Name, nil, Eqn) = finishedEqn(Eqn) .
    eq verifyEqn(Name, | Vars | Ts, Eqn) =
      (if leastSort(buildModule(Name, Vars),  leftHandSide(Eqn)) ==
          leastSort(buildModule(Name, Vars), rightHandSide(Eqn))
       then ok else errorEqn?(Vars, Eqn) fi) @ verifyEqn(Name, Ts, Eqn) .

    op verifyMbs : Qid VarDeclTupleList MembAxSet -> Warnings .
    op verifyMb  : Qid VarDeclTupleList MembAx -> Warnings .
    var Mb : MembAx .  var Mbs : MembAxSet .
    eq verifyMbs(Name, Ts, none) = ok .
    eq verifyMbs(Name, Ts, Mb Mbs) =
      verifyMb(Name, Ts, Mb) @ verifyMbs(Name, Ts, Mbs) .
    eq verifyMb(Name, nil, Mb) = finishedMb(Mb) .
    eq verifyMb(Name, | Vars | Ts, Mb) =
      (if sortLeq(buildModule(Name, Vars),
                  leastSort(buildModule(Name, Vars),  leftHandSide(Mb)),
                  rightHandSide(Mb))
       then ok else errorMb?(Vars, Mb) fi) @ verifyMb(Name, Ts, Mb) .
  endfm
```

All equations and membership assertions are thoroughly tested on the specification representing the abstract interpretation. In the case of equations, the least sorts of both terms are calculated (using the builtin operation `leastSort`) and if they are different, a warning is issued. In the case of membership assertions, the least sort of the declared term is calculated and then compared with the declared sort (using the builtin operation `sortLeq`); if it is not smaller then a warning is generated.