Abstract Interpretation with Alien Expressions and Heap Structures

Bor-Yuh Evan Chang[1,⋆] and K. Rustan M. Leino[2]

[1] University of California, Berkeley, California, USA, bec@cs.berkeley.edu
[2] Microsoft Research, Redmond, Washington, USA, leino@microsoft.com

Abstract. The technique of abstract interpretation analyzes a computer program to infer various properties about the program. The particular properties inferred depend on the particular abstract domains used in the analysis. Roughly speaking, the properties representable by an abstract domain follow a domain-specific schema of relations among variables. This paper introduces the congruence-closure abstract domain, which in effect extends the properties representable by a given abstract domain to schemas over arbitrary terms, not just variables. Also, this paper introduces the heap succession abstract domain, which when used as a base domain for the congruence-closure domain, allows given abstract domains to infer properties in a program's heap. This combination of abstract domains has applications, for example, to the analysis of object-oriented programs.

1 Introduction

The automatic reasoning about computer programs from their program text is called *static analysis*. It has applications in, for example, compiler optimizations and program verification. An important form of static analysis is *abstract interpretation* [6, 7], which systematically computes over-approximations of sets of reachable program states. The over-approximations are represented as elements of some given lattice, called an *abstract domain*. The elements of the abstract domain can be viewed as constraints on a set of variables, typically the variables of the program. For example, the polyhedra abstract domain [8] can represent linear-arithmetic constraints like $x + y \leqslant z$.

Often, the constraints of interest involve function and relation symbols that are not all supported by any single abstract domain. For example, a constraint of possible interest in the analysis of a Java or C# program is $\mathsf{sel}(H, o, \mathsf{x}) + k \leqslant \mathsf{length}(a)$ where H denotes the current heap, $\mathsf{sel}(H, o, \mathsf{x})$ represents the value of the x field of an object o in the heap H (written $o.x$ in Java and C#), and $\mathsf{length}(a)$ gives the length of an array a. A constraint like this cannot be represented directly in the polyhedra domain because the polyhedra domain does not support the functions sel and length. Consequently, the polyhedra abstract

⋆ This work was performed while Bor-Yuh Evan Chang was a research intern at Microsoft Research.

R. Cousot (Ed.): VMCAI 2005, LNCS 3385, pp. 147–163, 2005.

(a)	(b)	(c)	(d)

if $0 \leqslant x$ **then**
$\quad y := x$
else
$\quad y := -x$
end
	if $0 \leqslant o.x$ **then**
	$\quad y := o.x$
	else
	$\quad y := -o.x$
	end
		$x := 0\ ;\ y := 0\ ;$
		while $x < N$ **do**
		$\quad y := y + x\ ;$
		$\quad x := x + 1$
		end
			$o.x := 0\ ;\ p.y := 0\ ;$
			while $o.x < N$ **do**
			$\quad p.y := p.y + o.x\ ;$
			$\quad o.x := o.x + 1$
			end

Fig. 1. Two pairs of simple programs demonstrating the difference in what can be inferred without and with the congruence-closure and the heap succession abstract domains.

domain would very coarsely over-approximate this constraint as `true`—the lattice element that conveys no information. This example conveys a general problem for many abstract domains: the abstract domain only understands constraints consisting of variables and its supported function and relation symbols. If a given constraint mentions other, *alien*, function or relation symbols, it is ignored (that is, it is very coarsely over-approximated) by the abstract domain.

Rather than building in special treatment of such alien symbols in each abstract domain, we propose a coordinating *congruence-closure abstract domain*, parameterized by any set of given abstract domains that we shall refer to as *base domains*. The congruence-closure abstract domain introduces variables to stand for subexpressions that are alien to a base domain, presenting the base domain with the illusion that these expressions are just variables. For example, by itself, the polyhedra domain can infer that $0 \leqslant y$ holds after the program in Fig. 1(a), but it can only infer `true` after the program in Fig. 1(b). In contrast, the congruence-closure domain using the polyhedra domain as a base domain can also infer that $0 \leqslant y$ holds after the program in Fig. 1(b).

In this paper, we introduce the congruence-closure abstract domain and detail its operations. The congruence-closure abstract domain gets its name from the fact that it stores congruence-closed equivalence classes of terms. It is these equivalence classes that are represented as variables in the base domains. Equivalence classes may be dissolved as the variables of the program change. So as not to lose too much information, the congruence-closure domain consults its base domains during such updates.

We also introduce a particular base domain, the *heap succession abstract domain*, that is useful in analyzing programs with a heap, such as object-oriented programs (but also applies more generally to arrays and records). The benefit of this domain is demonstrated by the programs in Fig. 1 where program (d) involves updates to the heap. The polyhedra domain can infer that $0 \leqslant x \wedge 0 \leqslant y$ holds after the program in Fig. 1(c), but it can only infer `true` after the program in Fig. 1(d), even when the polyhedra domain is used as a single base domain of the congruence-closure domain. However, if one additionally uses the heap succession domain as a base domain, one can infer that $0 \leqslant o.x \wedge 0 \leqslant p.y$ holds after the program in Fig. 1(d).

2 Abstract Interpretation

In this section, we introduce the basic interface of each abstract domain. In an extended version of this paper [5], we briefly review abstract interpretation [6] and illustrate the use of our abstract-domain interface to infer properties of programs written in a toy imperative language.

Expressions. We assume expressions of interest to be variables and functions applied to expressions:

expressions	Expr	e, p	$::=$	$x \mid f(\vec{e})$
variables	Var	$x, y, \ldots$		
function symbols	FunSym	f		
expression sequences	Expr[ ]	$\vec{e}$	$::=$	$e_0, e_1, \ldots, e_{n-1}$

In programs and examples, we take the liberty of deviating from this particular syntax, instead using standard notation for constants and operators. For example, we write 8 instead of 8() and write $x + y$ instead of $+(x, y)$. A *constraint* is any boolean-valued expression.

Abstract Domains. The basic abstract domain interface is shown in Fig. 2. Each abstract domain provides a type `Elt`, representing the elements of the abstract domain lattice. Each lattice element corresponds to a constraint on variables. This constraint is returned by the `ToPredicate` operation. Conversely, `ToElt`(p) yields the most precise representation for constraint p in the lattice, which may have to lose some information. We do not need to compute `ToElt`, so we have omitted it from the abstract domain interface. In the literature [6], the functions corresponding to `ToElt` and `ToPredicate` are often written as α (abstraction) and γ (concretization), respectively.

```
interface AbstractDomain {
  type Elt ;
  ToPredicate: Elt → Expr ;
  Top: Elt ;
  Bottom: Elt ;
  AtMost: Elt × Elt → bool ;
  Constrain: Elt × Expr → Elt ;
  Eliminate: Elt × Var → Elt ;
  Rename: Elt × Var × Var → Elt ;
  Join: Elt × Elt → Elt ;
  Widen: Elt × Elt → Elt ;
}
```

Fig. 2. Abstract domains.

An abstract domain is required to define a partial ordering on the lattice elements (`AtMost`), `Top` and `Bottom` elements (required to correspond to `true` and `false`, respectively), and `Join` and `Widen` operations. Furthermore, an abstract domain must define operations to add a constraint to an element (`Constrain`), existentially quantify a variable (`Eliminate`), and rename a free variable (`Rename`), all of which may be conservative. See the extended version [5] for a more detailed description of these operations.

In the extended version [5], we also fix a particular imperative language and review how to apply the abstract domain operations to compute over-approximations of reachable states to infer properties about the program. This ends our general discussion of abstract interpretation. Next, we describe the congruence-closure abstract domain.

3 Congruences and Alien Expressions

The congruence-closure abstract domain $\mathcal{C}$ is parameterized by a list of base domains $\vec{\mathcal{B}}$. A lattice element of the congruence-closure domain is either $\bot$, representing $\mathtt{Bottom}_{\mathcal{C}}$, or has the form $\langle G, \vec{B} \rangle$, where G is an equivalence graph (*e-graph*) that keeps track of the names given to alien expressions and $\vec{B}$ is a list containing one non-$\mathtt{Bottom}_{\mathcal{B}_i}$ lattice element from each base domain $\mathcal{B}_i$. The names introduced by the congruence-closure domain to stand for alien expressions appear as variables to the base domains. To distinguish these from the variables used by the client of the congruence-closure domain, we call the newly introduced variables *symbolic values*. Intuitively, a symbolic value represents the value to which a client expression evaluates. Alternatively, one can think of the symbolic value as identifying an equivalence class in the e-graph. Throughout, we use Roman letters to range over client variables and Greek letters to range over symbolic values. The e-graph consists of a set of mappings:

$$\begin{array}{llrcl} \text{mappings} & \mathsf{Mapping} & m & ::= & t \mapsto \alpha \\ \text{terms} & \mathsf{Term} & t & ::= & x \mid f(\vec{\alpha}) \\ \text{symbolic values} & \mathsf{SymVal} & \alpha, \beta, \ldots & & \end{array}$$

In addition to providing the service of mapping alien expressions to symbolic values, the e-graph keeps track of equalities between terms. It represents an equality between terms by mapping these terms to the same symbolic value. For example, the constraint $w = \mathsf{f}(x) \wedge \mathsf{g}(x, y) = \mathsf{f}(y) \wedge w = \mathsf{h}(w)$ is represented by the e-graph

$$w \mapsto \alpha \quad x \mapsto \beta \quad \mathsf{f}(\beta) \mapsto \alpha \quad y \mapsto \gamma \quad \mathsf{g}(\beta, \gamma) \mapsto \delta \quad \mathsf{f}(\gamma) \mapsto \delta \quad \mathsf{h}(\alpha) \mapsto \alpha \quad \text{(Ex. 1)}$$

The e-graph maintains the invariant that the equalities it represents are congruence-closed. That is, if the e-graph represents the terms $\mathsf{f}(x)$ and $\mathsf{f}(y)$ and the equality $x = y$, then it also represents the equality $\mathsf{f}(x) = \mathsf{f}(y)$. For instance, if the e-graph in Ex. 1 is further constrained by $x = y$, then β and γ are unified, which in turn leads to the unification of α and δ, after which the e-graph becomes

$$w \mapsto \alpha \qquad x \mapsto \beta \qquad \mathsf{f}(\beta) \mapsto \alpha \qquad y \mapsto \beta \qquad \mathsf{g}(\beta, \beta) \mapsto \alpha \qquad \mathsf{h}(\alpha) \mapsto \alpha$$

A supplementary description of how these mappings can be viewed as a graph is given in Appendix A.

To compute $\mathtt{ToPredicate}_{\mathcal{C}}(\langle G, \vec{B} \rangle)$, the congruence-closure domain first obtains a predicate from each base domain $\mathcal{B}_i$ by calling $\mathtt{ToPredicate}_{\mathcal{B}_i}(B_i)$. Since the base domains represent constraints among the symbolic values, these predicates will be in terms of symbolic values. The congruence-closure domain then replaces each such symbolic value α with a client expression e, such that recursively mapping the subexpressions of e to symbolic values yields α. In Sec. 3.3, we explain how we ensure that such an e exists for each α. Finally, the congruence-closure domain conjoins these predicates with a predicate expressing the equalities represented by the e-graph. For example, if the congruence-closure domain uses a single base domain $\mathcal{B}_0$ for which $\texttt{ToPredicate}_{\mathcal{B}_0}(B_0)$ returns $\alpha \leqslant \gamma$, then the congruence-closure domain may compute $\texttt{ToPredicate}_{\mathcal{C}}(\langle(\text{Ex. 1}), \vec{B}\rangle)$ as $w = \mathsf{f}(x) \wedge \mathsf{g}(x, y) = \mathsf{f}(y) \wedge w = \mathsf{h}(w) \wedge w \leqslant y$.

In the remainder of this section, we detail the other abstract domain operations for the congruence-closure domain.

3.1 Constrain

The operation $\texttt{Constrain}_{\mathcal{C}}(\langle G, \vec{B}\rangle, p)$ may introduce some new symbolic values and constraints in G and then calls $\texttt{Constrain}_{\mathcal{B}_i}(B_i, p_i)$ on each base domain $\mathcal{B}_i$, where p_i is p with expressions alien to $\mathcal{B}_i$ replaced by the corresponding symbolic value. If any $\texttt{Constrain}_{\mathcal{B}_i}$ operation returns $\texttt{Bottom}_{\mathcal{B}_i}$, then $\texttt{Constrain}_{\mathcal{C}}$ returns $\perp$. Additionally, if the constraint p is an equality, then the congruence-closure domain will make note of it in the e-graph by calling `Union` (discussed below).

In order for the congruence-closure domain to know which subexpressions of p to replace by symbolic values, we extend the interface of abstract domains with the following operation:

$$\texttt{Understands}\colon \mathsf{FunSym} \times \mathsf{Expr}[\,] \to \mathbf{bool}$$

which indicates whether the abstract domain understands the given function symbol in the given context (*i.e.*, the arguments to the function in question). An abstract domain may choose to indicate it "understands" a function symbol even when it only partially interprets it.

To translate the client expression to an expression understandable to a base domain, the congruence-closure domain traverses top-down the abstract syntax tree of the client expression calling `Understands` on the base domain for each function symbol. If the base domain understands the function symbol, then $\mathcal{C}$ leaves it as is. If not, then $\mathcal{C}$ replaces the alien subexpression with a symbolic value and adds this mapping to the e-graph. Hopeful that it will help in the development of good reduction strategies (see Sec. 6), we also let $\mathcal{C}$ continue to call `Understands` on subexpressions of alien expressions and assert equalities with the symbolic value for any subexpression that is understood by the base domain. In fact, this is done whenever a new client expression is introduced into the e-graph as part of the `Find` operation (discussed below).

To illustrate the $\texttt{Constrain}_{\mathcal{C}}$ operation, suppose the congruence-closure domain is given the following constraint:

$$\texttt{Constrain}_{\mathcal{C}}(\langle G, \vec{B}\rangle,\ 2 \cdot x + \mathsf{sel}(H, o, \mathsf{f}) \leqslant |y - z|)$$

If a base domain $\mathcal{B}_i$ is the polyhedra domain, which understands linear arithmetic ($+,\ -,\ \cdot,\ 2,\ \leqslant$ in this example), then the congruence-closure domain makes the following calls on the polyhedra domain $\mathcal{B}_i$:

$$\texttt{Constrain}_{\mathcal{B}_i}(\texttt{Constrain}_{\mathcal{B}_i}(B_i,\ \gamma = \upsilon - \zeta),\ 2 \cdot \chi + \alpha \leqslant \beta)$$

and the e-graph is updated to contain the following mappings:

$$\begin{array}{ccr} x \mapsto \chi & H \mapsto \sigma & \mathsf{sel}(\sigma, \omega, \phi) \mapsto \alpha \\ y \mapsto \upsilon & o \mapsto \omega & |\gamma| \mapsto \beta \\ z \mapsto \zeta & \mathsf{f} \mapsto \phi & \upsilon - \zeta \mapsto \gamma \end{array}$$

We now define the union-find operations on the e-graph. The `Union` operation merges two equivalence classes. It does so by unifying two symbolic values and then merging other equivalence classes to keep the equivalences congruence-closed. Unlike the standard union operation, but akin to the union operation in the Nelson-Oppen congruence closure algorithm that combines decision procedures in a theorem prover [15], doing the unification involves updating the base domains.

The `Find` operation returns the name of the equivalence class of a given client expression, that is, its symbolic value. If the e-graph does not already represent the given expression, the `Find` operation has a side effect of adding the representation to the e-graph. Like `Union`, this operation differs from the standard find operation in that it involves updating the base domains. As noted above, to avoid loss of information by the congruence-closure domain, additional equality constraints between understandable subexpressions and their symbolic values (like $\gamma = \upsilon - \zeta$ in the example above) are given to the base domains. Detailed pseudo-code for $\texttt{Constrain}_{\mathcal{C}}$ along with both `Union` and `Find` are given in the extended version [5].

3.2 Rename and Eliminate

Since the base domains never see client variables, the congruence-closure domain can implement $\texttt{Rename}_{\mathcal{C}}$ without needing to call the base domains. The congruence-closure domain need only update its e-graph to map the new variable to the symbolic value mapped by the old variable (and remove the mapping of the old variable).

Similar to $\texttt{Rename}_{\mathcal{C}}$, we implement $\texttt{Eliminate}_{\mathcal{C}}$ by simply removing the mapping of the given variable (without calling the base domains). This means that base domains may have constraints on symbolic values that are no longer representable in terms of client variables. We postpone eliminating such "garbage values" from the base domains until necessary, as we describe in the next subsection. Pseudo-code for $\texttt{Rename}_{\mathcal{C}}$ and $\texttt{Eliminate}_{\mathcal{C}}$ are also given in the extended version [5].

3.3 Cleaning Up Garbage Values

Garbage values—symbolic values that do not map to any client expressions—can be generated by $\texttt{Eliminate}_{\mathcal{C}}$, $\texttt{Join}_{\mathcal{C}}$, and $\texttt{Widen}_{\mathcal{C}}$. The garbage values would be a problem for $\texttt{ToPredicate}_{\mathcal{C}}$. Therefore, at strategic times, including at the start of the $\texttt{ToPredicate}_{\mathcal{C}}$ operation, the congruence-closure domain performs a garbage collection. Roughly speaking, `Eliminate` with garbage collection is a lazy quantifier elimination operation.

To garbage collect, we use a "mark-and-sweep" algorithm that determines which terms and symbolic values are *reachable* in the e-graph from a client expression; a symbolic value that is not reachable is a garbage value. We define "reachable (from a client expression)" as the smallest relation such that: (a) any client variable is reachable, (b) any function application term whose arguments are all reachable is reachable, and (c) if the left-hand side of a mapping in the e-graph is reachable, then so is the right-hand side of the mapping.

There may be terms in the e-graph that depend on unreachable symbolic values (*i.e.*, that take unreachable symbolic values as arguments). Dropping these may lead to an undesirable loss of information, as we demonstrate in Sec. 4. However, the base domains may have additional information that would allow us to rewrite the term to not use the garbage value. To harvest such information, we extend the abstract domain interface with the following operation:

$$\texttt{EquivalentExpr}\colon \mathsf{Elt} \times \mathsf{Queryable} \times \mathsf{Expr} \times \mathsf{Var} \to \mathsf{Expr}\ \textbf{option}$$

Operation $\texttt{EquivalentExpr}(B, Q, t, \alpha)$ returns an expression that is equivalent to t but does not mention α (if possible). The $\mathsf{Queryable}$ parameter Q provides the base domain an interface to broadcast queries to all other abstract domains about certain predicates, which it might need to yield an equivalent expression.

After marking, the garbage collector picks a candidate garbage value (say α), if any. Then, for every mapping $t \mapsto \beta$ where t mentions α, each base domain is asked for an equivalent expression for t that does not mention α; if one is obtained, then the t in the mapping is replaced by the equivalent expression. The marking algorithm is then resumed there, in case an equivalent expression may have given rise to more reachable terms and symbolic values. After that, if α is still unreachable, all remaining mappings that mention α are removed from the e-graph and $\texttt{Eliminate}_{\mathcal{B}_i}(B_i, \alpha)$ is called on every base domain $\mathcal{B}_i$. At this time, α has either been determined to be reachable after all, or it has been eliminated completely from the e-graph and all base domains. The garbage collector then repeats this process for the next candidate garbage value, if any.

3.4 Congruence-Closure Lattice

Mathematically, we view the congruence-closure domain $\mathcal{C}$ as the Cartesian product lattice [7] over an *equivalences lattice* $\mathcal{E}$ and the base domain lattices. We consider the equivalences lattice $\mathcal{E}$ as the lattice over (empty, finite, and infinite) conjunctions of equality constraints between expressions ordered by logical implication. Elementary lattice theory gives us that both $\mathcal{E}$ and $\mathcal{C}$ are indeed lattices (assuming the base domain lattices are indeed lattices) [4].

However, as with other "standard" e-graph data structures, the e-graph described in previous sections represents only an empty or finite conjunction of ground equalities plus implied congruences, that is, only a proper subset of $\mathcal{E}$. To define the set of equalities implied by an e-graph, we define the evaluation judgment $G \vdash e \Downarrow \alpha$, which says that the e-graph G evaluates the client expression e to the symbolic value α:

$$\boxed{G \vdash e \Downarrow \alpha}$$

$$\frac{G(x) = \alpha}{G \vdash x \Downarrow \alpha}\ \textsf{var} \qquad \frac{G \vdash e_0 \Downarrow \alpha_0 \quad \cdots \quad G \vdash e_{n-1} \Downarrow \alpha_{n-1} \quad G(f(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})) = \alpha}{G \vdash f(e_0, e_1, \ldots, e_{n-1}) \Downarrow \alpha}\ \textsf{fun}$$

This corresponds to intuition that an expression belongs to the equivalence class of expressions labeled by the symbolic value to which it evaluates. We define the equalities implied by an e-graph by introducing the following judgment:

$$\boxed{G \Vdash e_0 = e_1}$$

$$\frac{G \vdash e_0 \Downarrow \alpha \quad G \vdash e_1 \Downarrow \alpha}{G \Vdash e_0 = e_1}\ \textsf{eval} \qquad \frac{G \Vdash e_0 = e_1}{G \Vdash f(e_0) = f(e_1)}\ \textsf{cong}$$

$$\frac{}{G \Vdash e = e}\ \textsf{refl} \qquad \frac{G \Vdash e_1 = e_0}{G \Vdash e_0 = e_1}\ \textsf{symm} \qquad \frac{G \Vdash e_0 = e_1 \quad G \Vdash e_1 = e_2}{G \Vdash e_0 = e_2}\ \textsf{trans}$$

An equality is implied by the e-graph if either both sides evaluate to the same symbolic value, it is a congruence implied by the e-graph, or it is implied by the axioms of equality.

We let $\mathcal{G}$ denote the poset of e-graphs ordered with the partial order from $\mathcal{E}$ (*i.e.*, logical implication). All the operations already described above have the property that given an element representable by an e-graph, the resulting element can be represented by an e-graph. However, $\texttt{Join}_{\mathcal{G}}$ cannot have this property, which is demonstrated by the following example given by Gulwani *et al.* [9]:

$$(x = y) \sqcup_{\mathcal{E}} \big(\mathsf{g}(x) = \mathsf{g}(y) \ \wedge\ x = \mathsf{f}(x) \ \wedge\ y = \mathsf{f}(y)\big) \quad = \bigwedge_{i:\ i \geqslant 0} \mathsf{g}(\mathsf{f}^i(x)) = \mathsf{g}(\mathsf{f}^i(y)) \quad \text{(Ex. 2)}$$

where we write $\sqcup_{\mathcal{E}}$ for the join in the lattice $\mathcal{E}$ and $\mathsf{f}^i(x)$ for i applications of f. This example shows that $\mathcal{G}$ is not a lattice, since for any k, $\wedge_{i:\ 0 \leqslant i \leqslant k} \mathsf{g}(\mathsf{f}^i(x)) = \mathsf{g}(\mathsf{f}^i(y))$ can be represented by an e-graph, but not the infinite conjunction. Thus, $\texttt{Join}_{\mathcal{C}}$ may have to conservatively return an e-graph that is less precise (*i.e.*, higher) than the join in $\mathcal{E}$. These issues are discussed further in Sec. 3.5.

AtMost. Aside from the trivial cases where one or both of the inputs are $\texttt{Top}_{\mathcal{C}}$ or $\texttt{Bottom}_{\mathcal{C}}$, $\texttt{AtMost}_{\mathcal{C}}(\langle G_0, \vec{B_0}\rangle, \langle G_1, \vec{B_1}\rangle)$ holds if and only if $G_1 \Vdash e_0 = e_1$ implies $G_0 \Vdash e_0 = e_1$ for all e_0, e_1 and $\texttt{AtMost}_{\vec{\mathcal{B}}}(\vec{B_0}, \vec{B_1})$. For the e-graphs, we determine if all equalities implied by G_1 are implied by G_0 by considering all "ground" equalities in G_1 (given by two mappings to the same symbolic value) and seeing if a $\texttt{Find}$ on both sides in G_0 yield the same symbolic value (since the e-graph is congruence-closed).

3.5 Join

The primary concern is how to compute the join of two e-graphs, since the overall join for elements of the congruence-closure domain is simply the join of the e-graphs and the join for the base domains (which is obtained by calling $\mathtt{Join}_{\vec{\mathcal{B}}}$ on the base domains). Some may find the graphical view of the e-graph described in Appendix A more intuitive for understanding this algorithm, though it is not necessary. Intuitively, there is a potential symbolic value (*i.e.*, node) in the result e-graph for every pair of symbolic values in the input e-graphs (one from each). Let us denote a symbolic value in the resulting e-graph with the pair of symbolic values from the input e-graphs, though we actually assign a new symbolic value to each unique pair of symbolic values. Then, the resulting e-graph $G = \mathtt{Join}_{\mathcal{G}}(G_0, G_1)$ consists of the following mappings:

$$\begin{array}{rl} x \mapsto \langle \alpha', \beta' \rangle & \quad \text{if } G_0(x) = \alpha' \text{ and } G_1(x) = \beta' \\ f(\overrightarrow{\langle \alpha, \beta \rangle}) \mapsto \langle \alpha', \beta' \rangle & \quad \text{if } G_0(f(\vec{\alpha})) = \alpha' \text{ and } G_1(f(\vec{\beta})) = \beta' \end{array}$$

In Fig. 3, we give the algorithm that computes this join of e-graphs, introduces the new symbolic values in the base domains, and then computes $\mathtt{Join}_{\mathcal{C}}$ as the Cartesian product of the various joins. As we create new symbolic values in the result e-graph, we need to remember the corresponding pair of symbolic values in the input graphs. This is given by two partial mappings M_0 and M_1 that map symbolic values in the resulting e-graph to symbolic values in G_0 and G_1, respectively. $\mathit{Visited}_0$ and $\mathit{Visited}_1$ track the symbolic values that have already been considered in G_0 and G_1, respectively.

The workset W gets initialized to the variables and 0-ary functions that are in common between the input graphs (along with where they map in the input graphs) (line 5, Fig. 3). One can consider the workset as containing terms (*i.e.*, edges) that will be in the resulting e-graph but do not yet have a symbolic value to map to (*i.e.*, a destination node).

Then, until the workset is empty, we choose some term to determine what symbolic value it should map to in the resulting e-graph. For a $\langle t, \alpha_0, \alpha_1 \rangle \in W$, if the pair $\langle \alpha_0, \alpha_1 \rangle$ is one where we have already assigned a symbolic value γ in the resulting e-graph G, then map t to γ in G (line 9). Otherwise, it is a new pair, and we create a new symbolic value (*i.e.*, node) ρ in G, update M_0 and M_1 accordingly, consider α_0 and α_1 visited, and map t to ρ in G (lines 11–15). So that information is not lost unnecessarily (unless chosen to by the base domains), we assert equalities between the symbolic values in the input graphs with the corresponding symbolic values in the result graph (line 12) before taking the join of the base domains. Finally, we find each function in common between G_0 and G_1 from α_0 and α_1, respectively, where all arguments have now been visited (α_0 and α_1 being the last ones). We add each such function to the workset but with the arguments being in terms of the symbolic values of the resulting e-graph (line 16).

We can make a few small optimizations when creating a new symbolic value in the result graph. First, if we have a global invariant that symbolic values are never reused, then α can be used for the symbolic value in the resulting e-graph corresponding to the pair $\langle \alpha, \alpha \rangle$ in the input graphs (rather than getting a fresh symbolic value). Second, for the first symbolic value ρ in the resulting e-graph that maps to α_0 in the input graph G_0, rather than calling

fun $\texttt{Join}_{\mathcal{C}}(\langle G_0, \vec{B_0} \rangle : \mathsf{Elt}, \langle G_1, \vec{B_1} \rangle : \mathsf{Elt}) : \mathsf{Elt} =$
  let $G : \mathsf{EGraph}$ **in**
  let $B_0', B_1' : \mathsf{Elt}[\,] = B_0, B_1$ **in**
  let $M_0, M_1 : \mathsf{SymVal} \to \mathsf{SymVal}$ **in**
  let $\mathit{Visited}_0, \mathit{Visited}_1$: set of SymVal **in**
  let W : set of $\mathsf{Term} \times \mathsf{SymVal} \times \mathsf{SymVal} =$
    $\{\langle x, G_0(x), G_1(x) \rangle \mid x \in \mathrm{domain}(G_0) \wedge x \in \mathrm{domain}(G_1)\}$
    $\cup \{\langle f(), G_0(f()), G_1(f()) \rangle \mid f() \in \mathrm{domain}(G_0) \wedge f() \in \mathrm{domain}(G_1)\}$
  in
  while W is not empty **do**
    pick and remove $(t, \alpha_0, \alpha_1) \in W$;
    if $M_0^{-1}(\alpha_0) \cap M_1^{-1}(\alpha_1) = \{\gamma\}$ **then**
      add $t \mapsto \gamma$ to G
    else
      let $\rho : \mathsf{SymVal} =$ fresh SymVal **in**
      $\vec{B_0'} := \texttt{Constrain}_{\vec{\mathcal{B}}}(\vec{B_0'}, \alpha_0 = \rho)$; $\vec{B_1'} := \texttt{Constrain}_{\vec{\mathcal{B}}}(\vec{B_1'}, \alpha_1 = \rho)$;
      add $\rho \mapsto \alpha_0$ to M_0 and $\rho \mapsto \alpha_1$ to M_1;
      add $t \mapsto \rho$ to G;
      add α_0 to $\mathit{Visited}_0$ and α_1 to $\mathit{Visited}_1$;
      find each $f(\vec{\beta_0}) \in \mathrm{domain}(G_0)$ and $f(\vec{\beta_1}) \in \mathrm{domain}(G_1)$ such that
        $\vec{\beta_0} \subseteq \mathit{Visited}_0 \wedge \alpha_0 \in \vec{\beta_0} \wedge \vec{\beta_1} \subseteq \mathit{Visited}_1 \wedge \alpha_1 \in \vec{\beta_1}$
      and add each $\langle f(\vec{\beta}), G_0(f(\vec{\beta_0})), G_1(f(\vec{\beta_1})) \rangle$ to W such that
        $M_0(\vec{\beta}) = \vec{\beta_0} \wedge M_1(\vec{\beta}) = \vec{\beta_1} \wedge \rho \in \vec{\beta}$
    end if
  end while;
  $\langle G, \texttt{Join}_{\vec{\mathcal{B}}}(\vec{B_0'}, \vec{B_1'}) \rangle$
end fun

Fig. 3. The join for the congruence-closure abstract domain.

$\texttt{Constrain}_{\vec{\mathcal{B}}}(\vec{B_0'}, \alpha_0 = \rho)$, we can call $\texttt{Rename}_{\vec{\mathcal{B}}}(\vec{B_0'}, \alpha_0, \rho)$ since α_0 will not be a symbolic value in the result e-graph (and similarly for G_1).

Soundness. We show that the above join algorithm indeed gives an upper bound. Note that since the $\texttt{Constrain}_{\vec{\mathcal{B}}}$ calls on the base domain simply give multiple names to existing variables, the soundness of $\texttt{Join}_{\mathcal{C}}$ reduces to soundness of the join of the e-graphs (assuming the joins of the base domains are sound). We write $\mathit{Join}_{\mathcal{G}}$ for the algorithm described in Fig. 3 ignoring the base domains. Informally, $\mathit{Join}_{\mathcal{G}}$ is sound if for any equality implied by the resulting e-graph, it is implied by both input e-graphs. The formal statement of the soundness theorem is given below, while its proof is given in our extended paper [5].

Theorem 1 (Soundness of $\texttt{Join}_{\mathcal{G}}$) *Let $G = \texttt{Join}_{\mathcal{G}}(G_0, G_1)$. If $G \Vdash e_0 = e_1$, then $G_0 \Vdash e_0 = e_1$ and $G_1 \Vdash e_0 = e_1$.*

Completeness. Note that different e-graphs can represent the same lattice element. For example, consider the following e-graphs

$$x \mapsto \alpha \quad y \mapsto \alpha \qquad \text{(Ex. 3a)} \qquad x \mapsto \alpha \quad y \mapsto \alpha \quad \mathsf{f}(\alpha) \mapsto \beta \quad \text{(Ex. 3b)}$$

that both represent the constraint $x = y$ (and any implied congruences). For the previous operations, the element that is represented by the result was the same regardless of the form of the e-graph in the input; however, the precision of the join algorithm is actually sensitive to the particular e-graph given as input. For example, the join of the e-graphs shown in Ex. 3a and Ex. 3b with an e-graph representing the constraint $\mathsf{f}(x) = \mathsf{f}(y)$ yields elements `true` and $\mathsf{f}(x) = \mathsf{f}(y)$, respectively, as shown below:

$$\begin{array}{l}\mathtt{Join}_{\mathcal{G}}(\{x \mapsto \alpha, y \mapsto \alpha\}, \{x \mapsto \gamma, y \mapsto \delta, \mathsf{f}(\gamma) \mapsto \varepsilon, \mathsf{f}(\delta) \mapsto \varepsilon\}) = \{x \mapsto \rho, y \mapsto \sigma\} \\ \mathtt{Join}_{\mathcal{G}}(\{x \mapsto \alpha, y \mapsto \alpha, \mathsf{f}(\alpha) \mapsto \beta\}, \{x \mapsto \gamma, y \mapsto \delta, \mathsf{f}(\gamma) \mapsto \varepsilon, \mathsf{f}(\delta) \mapsto \varepsilon\}) \\ \quad = \{x \mapsto \rho, y \mapsto \sigma, \mathsf{f}(\rho) \mapsto \tau, \mathsf{f}(\sigma) \mapsto \tau\}\end{array}$$

A naïve idea might be to extend e-graph (Ex. 3a) to (Ex. 3b) in the join algorithm as necessary; however, the algorithm no longer terminates if the join in the lattice $\mathcal{E}$ is not representable as a finite conjunction of equality constraints plus their implied congruences. Recall that Ex. 2 shows that such a non-representable join is possible.

Ex. 2 does, however, suggest that $\mathtt{Join}_{\mathcal{G}}$ can be made arbitrarily precise though not absolutely precise. In fact, the precision is controlled exactly by what terms are represented in the e-graph. If an equality is represented in both input e-graphs to $\mathtt{Join}_{\mathcal{G}}$, then that equality will be implied by the result e-graph. In fact, a slightly stronger statement holds that says that the equality will also be represented in the result e-graph. Thus, the precision of the join can be controlled by the client by introducing expressions it cares about in the initial e-graph. We state the completeness theorem formally below, while its proof is given in the extended version [5].

Theorem 2 (Relative Completeness of $\mathtt{Join}_{\mathcal{G}}$) *Let* $G = \mathtt{Join}_{\mathcal{G}}(G_0, G_1)$. *If* $G_0 \vdash e_0 \Downarrow \alpha_0$, $G_0 \vdash e_1 \Downarrow \alpha_0$, $G_1 \vdash e_0 \Downarrow \alpha_1$, *and* $G_1 \vdash e_1 \Downarrow \alpha_1$, *then* $G \Vdash e_0 = e_1$.

This theorem, however, does not directly indicate anything about the precision of the entire join $\mathtt{Join}_{\mathcal{C}}$. While without the calls to $\mathtt{Constrain}_{\vec{\mathcal{B}}}$, much information would be lost, it is not clear if as much as possible is preserved. Gulwani *et al.* [9] give the following challenge for obtaining precise combinations of join algorithms. Let $E_0 \stackrel{\text{def}}{=} a = a' \wedge b = b'$ and $E_1 \stackrel{\text{def}}{=} a = b' \wedge b = a'$, then

$$\begin{array}{ll} E_0 \sqcup_{\mathcal{E}} E_1 \equiv \mathsf{true} & E_0 \sqcup_{\mathcal{P}} E_1 \equiv a + b = a' + b' \\ E_0 \sqcup_{\mathcal{E},\mathcal{P}} E_1 \sqsubseteq_{\mathcal{E},\mathcal{P}} \bigwedge_{i:\, i \geqslant 0} f^i(a) + f^i(b) = f^i(a') + f^i(b') & \end{array}$$

where $\mathcal{P}$ is the polyhedra abstract domain and $\mathcal{E}, \mathcal{P}$ is a hypothetical combination of equalities of uninterpreted functions and linear arithmetic. Note that the combined join also yields an infinite conjunction of equalities not representable by our e-graph. Thus, we cannot achieve absolute completeness using the congruence-closure domain with the polyhedra domain as a base domain; however, we do achieve an analogous relative completeness where we obtain all conjuncts where the terms are represented in the input e-graphs. In the table below, we show the e-graphs for E_0 and E_1 with one application of f to each variable explicitly represented and the join of these e-graphs. Consider the input elements for the polyhedra domain to be $\mathtt{Top}_{\mathcal{P}}$. We show the elements after the calls to $\mathtt{Constrain}_{\mathcal{P}}$ during $\mathtt{Join}_{\mathcal{C}}$ and the final result after the polyhedra join.

	C_0		C_1		$\mathtt{Join}_{\mathcal{C}}(C_0, C_1)$	
E-Graph	$a \mapsto \alpha_0$	$b \mapsto \beta_0$	$a \mapsto \alpha_1$	$b \mapsto \beta_1$	$a \mapsto \rho$	$b \mapsto \tau$
	$a' \mapsto \alpha_0$	$b' \mapsto \beta_0$	$b' \mapsto \alpha_1$	$a' \mapsto \beta_1$	$a' \mapsto \sigma$	$b' \mapsto \upsilon$
	$\mathsf{f}(\alpha_0) \mapsto \gamma_0$	$\mathsf{f}(\beta_0) \mapsto \delta_0$	$\mathsf{f}(\alpha_1) \mapsto \gamma_1$	$\mathsf{f}(\beta_1) \mapsto \delta_1$	$\mathsf{f}(\rho) \mapsto \phi$	$\mathsf{f}(\tau) \mapsto \psi$
					$\mathsf{f}(\sigma) \mapsto \chi$	$\mathsf{f}(\upsilon) \mapsto \omega$
Polyhedra	$\alpha_0 = \rho = \sigma$	$\beta_0 = \tau = \upsilon$	$\alpha_1 = \rho = \upsilon$	$\beta_1 = \tau = \sigma$	$\rho + \tau = \sigma + \upsilon$	
(after Constrains)	$\gamma_0 = \phi = \chi$	$\delta_0 = \psi = \omega$	$\gamma_1 = \phi = \omega$	$\delta_1 = \psi = \chi$	$\phi + \psi = \chi + \omega$	

$\mathtt{ToPredicate}_{\mathcal{C}}$ on the result yields $a + b = a' + b' \ \wedge\ \mathsf{f}(a) + \mathsf{f}(b) = \mathsf{f}(a') + \mathsf{f}(b')$, as desired. (Gulwani and Tiwari have indicated that they have a similar solution for this example.) Note that there are no equality constraints in the resulting e-graph; these equalities are only reflected in the base domain. This example suggests that such equalities inferred by a base domain should be propagated back to the e-graph in case those terms exist in the e-graph for another base domain where such a term is alien (akin to equality sharing of Nelson-Oppen [15]).

3.6 Widen

Unfortunately, the above join operation successively applied to an ascending chain of elements may not stabilize (even without consideration of the base domains), as can demonstrated by the following example. Let G_i (for $i \geqslant 0$) be an ascending chain of e-graphs representing $x = \mathsf{f}^{2^i}(x)$. Then,

$$G'_0 = G_0 \qquad G'_1 = \mathtt{Join}_{\mathcal{G}}(G'_0, G_1) = G_1 \qquad G'_2 = \mathtt{Join}_{\mathcal{G}}(G'_1, G_2) = G_2 \qquad \cdots$$

does not reach a fixed point. The above sequence does not converge because a cycle in the e-graph yields an infinite number of client expressions that evaluate to a symbolic value (by following the loop several times). Thus, a non-stabilizing chain can be constructed by joining with a chain that successively rules out terms that follow the loop less than k times (as given above). The same would be true for acyclic graphs with the join algorithm that adds additional terms to the e-graph as necessary to be complete. Therefore, we can define $\mathtt{Widen}_{\mathcal{C}}$ by following the join algorithm described in Fig. 3 except fixing a finite limit on the number of times a cycle can be followed in G_0 (and calling $\mathtt{Widen}_{\vec{\mathcal{B}}}$ on the base domains rather than $\mathtt{Join}_{\vec{\mathcal{B}}}$). Once the e-graph part stabilizes, since the set of symbolic values are fixed up to renaming, the base domains will also stabilize by the stabilizing property of $\mathtt{Widen}_{\vec{\mathcal{B}}}$.

4 Heap Structures

In this section, we specifically consider programs with heaps, such as object-oriented programs. We view a heap as an array indexed by heap locations. Therefore, what we say here more generally applies also to arrays and records.

4.1 Heap-Aware Programs

We consider an imperative programming language with expressions to read object fields ($o.x$) and statements to update object fields ($o.x := e$). To analyze these programs, we explicitly represent the heap by a program variable H. The heap is an array indexed by heap locations $\langle o, \mathsf{x} \rangle$, where o denotes an object identity and x is a field name.

A field read expression $o.x$ in the language is treated simply as a shorthand for $\mathsf{sel}(H, o, \mathsf{x})$. Intuitively, this function retrieves the value of H at location $\langle o, \mathsf{x} \rangle$. Thus, from what we have already said, the congruence-closure domain allows us to infer properties of programs that read fields. For example, using the polyhedra domain as a base domain on the program in Fig. 1(b), we infer arithmetic properties like $y = \mathsf{sel}(H, o, \mathsf{x}) \wedge 0 \leqslant \mathsf{sel}(H, o, \mathsf{x})$ after the statement in the true-branch and $0 \leqslant y$ after the entire program.

The semantics of the field update statement $o.x := e$ is usually defined as an assignment $H := \mathsf{upd}(H, o, x, e)$ (*cf.* [10, 11, 16]), where upd is a function with the following axiomatization:

$$\begin{array}{ll} \mathsf{sel}(\mathsf{upd}(H, o, x, e), o', x') = e & \text{if } o = o' \text{ and } x = x' \\ \mathsf{sel}(\mathsf{upd}(H, o, x, e), o', x') = \mathsf{sel}(H, o', x') & \text{if } o \neq o' \text{ or } x \neq x' \end{array}$$

We choose a slightly different formulation introducing the *heap succession* predicate $H \equiv_{o.x} H'$, which means H' is an updated heap equivalent to H everywhere except possibly at $o.x$. We thus regard the field update statement $o.x := e$ as the following assignment:

$$H := H' \quad \text{where } H' \text{ is such that } H \equiv_{o.x} H' \text{ and } \mathsf{sel}(H', o, x) = e$$

A more precise semantics is given in the extended version [5].

Unfortunately, this is not enough to be useful in the analysis of heap structured programs. Consider the program in Fig. 1(d). Applying the congruence-closure domain with, say, the polyhedra domain as a single base domain gives the disappointingly weak predicate `true` after the entire program. The problem is that an analysis of the field update statement will effect a call to the operation $\mathtt{Eliminate}_{\mathcal{C}}(\langle G, \vec{B} \rangle, H)$ on the congruence-closure domain, which has the effect of losing all the information that syntactically depends on H. This is because no base domain $\mathcal{B}_i$ is able to return an expression in response to the congruence-closure domain's call to $\mathtt{EquivalentExpr}_{\mathcal{B}_i}(B_i, Q, \mathsf{sel}(H, o, \mathsf{x}), H)$ (or more precisely, with expression $\mathsf{sel}(\sigma, \phi, \chi)$ and variable σ that are the corresponding symbolic values).

To remedy the situation, we develop an abstract domain that tracks heap updates. Simply including this abstract domain as a base domain in our congruence-closure abstract domain solves this problem.

4.2 Heap Succession Abstract Domain

A lattice element in the *heap succession abstract domain* $\mathcal{S}$ represents false or a conjunction of heap succession predicates

$$(\exists \ldots \bullet\ H_0 \equiv_{o_0.x_0} H_1 \ \wedge\ H_1 \equiv_{o_1.x_1} H_2 \ \wedge\ \cdots\ \wedge\ H_{n-1} \equiv_{o_{n-1}.x_{n-1}} H_n)$$

for some $n \geqslant 0$, where the H_i, o_i, and x_i are variables, some of which may be existentially bound, and where no H_i is repeated.

The heap succession domain, like any other base domain, works only with variables and implements the abstract domain interface. However, of primary importance is that it can often return useful results to `EquivalentExpr` calls. Specifically, it substitutes newer heap variables for older heap variables in expressions when it is sound to do so, which is exactly what we need. The operation $\texttt{EquivalentExpr}_{\mathcal{S}}(S, Q, t, H)$ returns nothing unless t has the form $\mathsf{sel}(H, o, x)$ and element S contains a successor of heap H. If there is a heap successor H' of H, that is, if S contains a predicate $H \equiv_{p.y} H'$, then $\mathcal{S}$ first determines whether $o \neq p \ \vee\ x \neq y$ (*i.e.*, whether the references o and p are known to be unaliased or the fields are distinct). If it finds that $o \neq p \ \vee\ x \neq y$ and H' is not existentially bound, then the operation returns the expression $\mathsf{sel}(H', o, x)$; otherwise, the operation iterates, this time looking for a heap successor of H'. If x and y denote two different fields (which are represented as 0-ary functions), the condition is easy to determine. If not, the heap succession domain may need to query the other abstract domains via Q to find out if any other abstract domain knows that $o \neq p$.

4.3 Preserving Information Across Heap Updates

We give an example to illustrate how the heap succession domain can allow information to be preserved across heap updates. Consider a heap update statement $o.x := z$ and suppose that before the update, the abstract domains have the information that $p.y = 8$ (*i.e.*, $\mathsf{sel}(H, p, \mathsf{y}) = 8$). After the update to $o.x$, we hope to preserve this information, since the update is to a different field name. Consider the relevant mappings in the e-graph after the update:

$$\begin{array}{cccc} & H \mapsto \sigma' & \mathsf{sel}(\sigma, \psi, \upsilon) \mapsto \alpha & \mathsf{sel}(\sigma', \phi, \chi) \mapsto \zeta \\ p \mapsto \psi & o \mapsto \phi & 8 \mapsto \alpha & z \mapsto \zeta \\ \mathsf{y} \mapsto \upsilon & \mathsf{x} \mapsto \chi & & \end{array}$$

while the heap succession domain has the following constraint: $\sigma \equiv_{\phi.\mathsf{x}} \sigma'$. The old heap σ is now a garbage value. Recall that during garbage collection before σ is eliminated from the base domain, the congruence-closure domain will call $\texttt{EquivalentExpr}_{\mathcal{B}_i}$ to ask each base domain $\mathcal{B}_i$ whether it can give an equivalent expression for $\mathsf{sel}(\sigma, \psi, \upsilon)$ without σ. In this case, the heap succession domain can return $\mathsf{sel}(\sigma', \psi, \upsilon)$ because field name constants x and y are distinct. Thus, the information that $\mathsf{sel}(H, p, \mathsf{y}) = 8$ is preserved. In the same way, the congruence-closure domain with heap succession and polyhedra as base domains computes $0 \leqslant o.x \ \wedge\ N \leqslant o.x \ \wedge\ 0 \leqslant p.y$ after the program in Fig. 1(d).

5 Related Work

Gulwani *et al.* [9] describe several join algorithms for both special cases of the theory of uninterpreted functions and in general. The representation of equality constraints they consider, called an *abstract congruence closure* [1, 2], is a convergent set of rewrite rules of the form $f(c_0, c_1, \ldots, c_{n-1}) \to c$ or $c_0 \to c$ for fresh constants $c, c_0, c_1, \ldots, c_{n-1}$. If the latter form is excluded, then we obtain something analogous to our e-graph where the fresh constants are our symbolic values. In fact, because the latter form can lead to many different sets of rewrite rules for the same set of equality constraints, Gulwani *et al.* quickly define a *fully reduced* abstract congruence closure that precisely excludes the latter form and then only work with fully reduced abstract congruence closures. Our work goes further by introducing the concept of base domains and recognizing that symbolic values can be used to hide alien expressions. Gulwani *et al.* discuss an item of future work to combine their join algorithm for the theory of uninterpreted functions with some other join algorithm (*e.g.*, for linear arithmetic) and a challenge for such a combination. Using the congruence-closure abstract domain with polyhedra as a base domain, we seem to stand up to the challenge (see Sec. 3.5).

Previous research in the area of abstract interpretation and dynamic data structures has centered around *shape analysis* [14], which determines patterns of connectivity between pointers in the heap. Using transitive closure, shape analysis can reason about reachability in the heap and abstracts many heap objects into so-called summary nodes. Our technique of combining abstract domains does not specifically attempt to abstract objects into summary nodes, though it would be interesting to consider the possibility of using such a shape analyzer as a base domain in our technique. In shape analysis, properties of nodes can be encoded as specially interpreted predicates (*cf.* [17, 12]). Our technique differs in that it extends the representable properties of nodes by simply plugging in, as base domains, classic abstract domains that reason only with relations among variables. This feature allows our analysis to obtain properties like $o.f \leqslant p.g$ with an "off-the-shelf" polyhedra implementation.

Logozzo uses abstract interpretation to infer object invariants with several objects but with some restrictions on the possible aliasing among object references [13]. The abstract domains described in this paper might be able to be used as building blocks for another method for inferring object invariants.

6 Conclusion

We have described a technique to extend any abstract domain to handle constraints over arbitrary terms, not just variables, via a coordinating abstract domain of congruences. Moreover, this technique is designed so that abstract domains can be used mostly unmodified and oblivious to its extended reasoning. To implement the congruence-closure domain, we have given a sound and relatively complete algorithm to join e-graphs.

Additionally, we have described the heap succession domain, which allows our framework to handle heap updates. This domain need only be a base domain and thus fits modularly into our framework. Lastly, the handling of heap updates can be improved modularly through other base domains that yield better alias (or rather, unaliased) information.

We have a prototype implementation of our technique in the abstract interpretation engine of the Spec# program verifier, which is part of the Spec# programming system [3], and are in the process of obtaining experience with it.

Our work is perhaps a step toward having a uniform way to combine abstract domains, analogous to the Nelson-Oppen algorithm for cooperating decision procedures [15]. For example, continuing to assign symbolic values to subexpressions of alien expressions, as well as notifying base domains of additional understandable subexpressions suggests some kind of potential sharing of information between abstract domains. The structure of our framework that uses a coordinating abstract domain of congruences is perhaps also reminiscent of Nelson-Oppen. While equality information flows from the congruence-closure domain to the base domains, to achieve cooperating abstract domains, we need to add a way for each base domain to propagate information, like equalities that it discovers, to the congruence-closure domain and other base domains. We believe exploring this connection would be an exciting line of research.

Acknowledgments. We thank Simon Ou and River Sun for helpful discussions about the e-graph join algorithm. Francesco Logozzo provided extensive comments on an early version of some work leading to the present paper. Rob Klapper participated in a preliminary study of applying abstract interpretation to object-oriented programs. Sumit Gulwani and the anonymous referees provided useful comments on earlier drafts. Finally, we thank the rest of the Spec# team at Microsoft Research for various discussions.

References

1. Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In David McAllester, editor, *Conference on Automated Deduction (CADE 2000)*, volume 1831 of *LNAI*, pages 64–78, June 2000.
2. Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, LNCS. Springer, 2004. To appear.
4. Garrett Birkhoff. *Lattice Theory*, volume XXV of *Colloquium Publications*. American Mathematical Society, 1940.
5. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. Technical Report MSR-TR-2004-115, Microsoft Research, November 2004.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth POPL*, pages 238–252, January 1977.

7. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Sixth POPL*, pages 269–282, January 1979.
8. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth POPL*, pages 84–96, January 1978.
9. Sumit Gulwani, Ashish Tiwari, and George C. Necula. Join algorithms for the theory of uninterpreted functions. In *24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, December 2004.
10. C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.
11. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, 1995. Available as Technical Report Caltech-CS-TR-95-03.
12. Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 26–38, 2000.
13. Francesco Logozzo. Separate compositional analysis of class-based object-oriented languages. In *10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, volume 3116 of *LNCS*, pages 332–346. Springer, July 2004.
14. Steven S. Muchnick and Neil D. Jones. Flow analysis and optimization of Lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.
15. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
16. Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.
17. Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

A Graphical View of the E-Graph

We can view the e-graph as a rooted directed graph where the vertices are the symbolic values (plus a distinguished root node) and the edges are the terms. Variables and 0-ary functions are labeled edges from the root node to the symbolic value to which they map. The n-ary functions are multi-edges with the (ordered) source nodes being the arguments of the function and the destination node being the symbolic value to which they map labeled with the function symbol. More precisely, let G be a mapping in Sec. 3, then the corresponding graph is defined as follows:

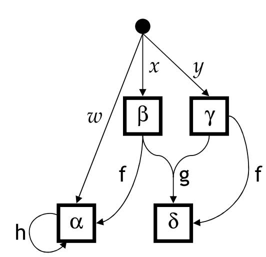

Fig. 4. An e-graph.

$$\begin{aligned}\text{vertices}(G) &= \text{range}(G) \cup \{\bullet\}\\ \text{edges}(G) &= \left\{\bullet \xrightarrow{x} G(x) \;\middle|\; x \in \text{domain}(G)\right\} \cup \left\{\vec{\alpha} \xrightarrow{f} G(f(\vec{\alpha})) \;\middle|\; f(\vec{\alpha}) \in \text{domain}(G)\right\}\end{aligned}$$

where $\bullet$ stands for the distinguished root node, as well as the empty sequence. Fig. 4 gives the graph for Ex. 1.