Program Verification with Interacting Analysis Plugins

Nathaniel Charlton

Department of Computing, Imperial College London

Abstract. In this paper we propose and argue for a modular framework for interprocedural program analysis, where multiple program analysis tools are combined in order to exploit the particular advantages of each. This allows for "plugging together" such tools as required by each verification task and makes it easy to integrate new analyses. Our framework automates the sharing of information between plugins using a first order logic with transitive closure, in a way inspired by the open product of Cortesi et al..

We describe a prototype implementation of our framework, which performs static assertion checking on a simple language for heap-manipulating programs. This implementation includes plugins for three existing approaches — predicate abstraction, 3-valued shape analysis and a decidable pointer analysis — and for a simple type system. We demonstrate through a detailed example the increase in precision that our approach can provide. Finally we discuss the design decisions we have taken, in particular the tradeoffs involved in the choice of language by which the plugins communicate, and identify some future directions for our work.

Keywords: abstraction, software verification, plugins, open product

1. Introduction

Finite-state model checking is a widely used method of formal verification in which one creates a finite model of some system's behaviour and then establishes properties of that system by exhaustively exploring the model's state space. Because all reachable states are examined, model checking gives very strong assurances of correctness compared to conventional testing techniques. Efficient "symbolic" model checking algorithms, based on BDDs, can now handle systems with upwards of 10^{20} states [BCM+92]. Traditional targets for model checking have been control systems and communication protocols.

Applications programs written in everyday programming languages tend to be infinite-state, and so model checking is not immediately applicable, but with the size, complexity and prevalence of such programs increasing all the time, methods of ensuring their reliability are of paramount importance. Thus it is unsurprising that recent work has explored the application of the technique to infinite-state software written

Please cite this document as "Nathaniel Charlton. Program Verification with Interacting Analysis Plugins. Technical Report 2006/11, Department of Computing, Imperial College London, United Kingdom, ISSN 1469-4174, September 2006."

in languages such as Java. The key to this is the process of abstraction: from the source program we produce an approximate, abstract program, which omits some of the detail of the original, but has finite-state behaviour and is therefore amenable to model checking.

To obtain meaningful results, we insist that our abstract program be a conservative approximation, or *simulation*, of the original, i.e. that all execution paths in the real program are possible in the abstract system (and possibly more). This ensures that safety properties checked on the abstract system "carry over" to the concrete one [BG03]. Abstract Interpretation [CC92] provides the necessary theoretical framework to show that such analyses are correct.

Checking software in this way has achieved some success: for example, communication protocols, garbage collectors and libraries implementing data structures have been verified (e.g. [HS96, DDP99, MS01] respectively). Well-known abstraction methods include polyhedral analysis e.g. [CC04], predicate abstraction [BR01], shape analysis [LAMS04] and various pointer analyses e.g. [MS01]. But although each of these methods works well for particular classes of programs and properties, all have "blind spots" where they are ineffective. Therefore it remains the case that "real" (applications) programs as written by ordinary programmers are beyond the scope of verification. The problem is that designing abstraction schemes is difficult: retaining too much irrelevant information results in high computational cost, but if relevant facts are thrown away it will not be possible to verify the desired property.

Contributions: Our aim in this paper is to introduce and argue for a modular framework for abstraction-based program verification, in which multiple program analysis tools are combined in order to exploit and amplify the particular advantages of each. This allows different subsets of the analyses to be "plugged together" as required by each verification task and makes it easy to integrate new ones.

As it runs, a program analysis tool produces intermediate results describing possible program states; these results are usually thought of as belonging to a lattice of abstract values. Because the intermediate results produced by different analyses are in general not independent, in theory such analyses can benefit from using each others' findings. But these interactions can be subtle and hence difficult and time-consuming to implement. The key aspect of our framework is that we seek to take advantage of such interactions automatically: inspired by the *open product* [CLCVH00] of Cortesi et al. we allow our plugins to exchange information using a first order logic with transitive closure. In [CLCVH00] the open product is applied to the detection of shallow properties of logic programs, for the purposes of optimisation; here we rework the idea in the context of verifying behavioural specifications of imperative programs.

We have produced a prototype implementation of a static assertion checker, which targets a simple language with imperative and (limited) object-oriented features. We allow recursion, but in order to make the verification problem more manageable we do not deal with inheritance, exceptions or concurrency, all of which introduce subtle difficulties.

Rest of paper: The rest of the paper is structured as follows:

- Section 2 provides a brief account of the relevant background to our work, and the motivation for our new framework.
- In section 3 we define our framework precisely. We give syntax and semantics to the simple programming language we analyse, define the notion of an *analysis plugin* which is central to our work and describe the mechanism by which these plugins cooperate. We give our algorithm for plugin-based interprocedural analysis, and sketch our proofs of termination and soundness results.
- In section 4 we describe our prototype implementation of an assertion checker, which includes plugins for three existing techniques: predicate abstraction, 3-valued shape analysis and a decidable pointer analysis (as in [BR01, LAMS04, MS01] respectively).
- In section 5 we work through an example verification which demonstrates the increase in precision which our framework can provide.
- Finally section 6 presents the thinking behind the various design decision we have taken, and the tradeoffs involved, and suggests future work.

This paper revises and extends the material from our workshop paper [Cha06]. Several aspects of our work, including the syntax and semantics of the programming language we target (subsections 3.1 to 3.4), and our algorithm for plugin-based analysis (subsection 3.6), are recounted more formally (based in fact on our formalisation of these aspects in the proof assistant Isabelle). New sections address the use of type systems as

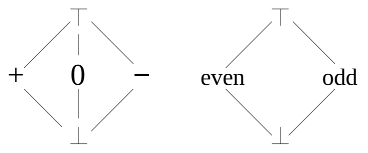


Fig. 1. The sign and parity abstraction lattices

plugins (subsection 4.4), and discuss our choice of intermediate language (subsection 6.1). When discussing our TVLA plugin (subsection 4.2) we now present explicitly the translations between our common logic and TVLA's logic.

2. Background

2.1. Abstract Interpretation and verification

The theory of Abstract Interpretation [CC92] is a general formal treatment of abstraction. It sets out conditions that an abstraction scheme ought to meet, and provides an assortment of results and algorithms that apply whenever these are satisfied.

To illustrate, consider a program with a single integer variable. The state of the program at any point of execution is an element of $\mathbb{Z}$, and the set of possible states reachable at any particular program point is an element of the complete lattice $\mathbb{P}(\mathbb{Z})$ (the set of subsets of the integers, ordered by inclusion). Finding the set of reachable states for each program point requires an iterative, least-fixed-point calculation, but this may never terminate because $\mathbb{P}(\mathbb{Z})$ is infinite. Abstract Interpretation tells us to approximate $\mathbb{P}(\mathbb{Z})$ with a finite (or finite-height) lattice L, where we will have guaranteed termination.

One such lattice is the *sign abstraction lattice* L_{sign} shown in Figure 1 (left).

The idea is that the elements $\{+, 0, -\}$ retain the sign of the integer variable while throwing away its exact value. The parity lattice on the right of Figure 1 is similar, recording only whether the integer is even or odd. The top element $\top$ represents no information about the integer variable, and the bottom element represents inconsistent information (and should only occur for unreachable program points). Precise meaning is given to the elements of the abstract lattice by giving a concretisation function γ, in this case a function of type $L_{sign} \to \mathbb{P}(\mathbb{Z})$:
The set $\gamma(a)$ is the set of concrete states represented by the abstract value a. From γ we define the abstraction function $\alpha : \mathbb{P}(\mathbb{Z}) \to L_{sign}$ which maps each concrete state set to its best abstract overapproximation (or *conservative approximation*, or *safe approximation*). For instance, both $\top$ and $+$ conservatively approximate the concrete state set $\{1, 2\}$, because $\gamma(\top) = \mathbb{Z} \supseteq \{1, 2\}$ and $\gamma(+) = \{n : n > 0\} \supseteq \{1, 2\}$, but we take $\alpha(\{1, 2\}) = +$ because $+$ is more precise. On the other hand, for $\{0, 1, 2\}$ the only available conservative approximation is $\top$, so $\alpha(\{0, 1, 2\}) = \top$.

Program statements such as assignments can be given meaning through a *transfer function* on the concrete state space. For example, the transfer function $f : \mathbb{Z} \to \mathbb{Z}$ associated with `x := x - 1` is $f(n) = n-1$. With each such f we associate an abstract transfer function $f^{\#}$ which operates on the abstract lattice and mimics the effect of f. For `x := x - 1`, we could have

$$\begin{array}{lll} f^{\#}(+) = \top & f^{\#}(0) = - & f^{\#}(\top) = \top \\ f^{\#}(-) = - & & f^{\#}(\bot) = \bot \end{array}$$

Note that there is a kind of information loss in the above $f^{\#}$, at $+$: since 1 and 3 are both represented by $+$, and $f(1) = 0$ and $f(3) = 2$, the best we can safely do is to set $f^{\#}(+) = \top$. These abstract transfer functions generate an abstract transition system which (under appropriate safety conditions on γ and the $f^{\#}$s) is a conservative approximation or simulation of the program's real behaviour, so that for every real execution path in the program there is a corresponding path in the abstract system (but not necessarily vice versa). Now we can search the abstract transition system for paths leading to “bad” or error states. If no such paths exist, we can conclude that executing the original program never leads to an error state. In particular we can do assertion checking, by transforming each assertion into code which tests the asserted condition and jumps to a special error label when it fails to hold.

Having seen the general setting, we now consider some specific instances of abstraction schemes.

2.2. Predicate abstraction

The idea of predicate abstraction is to group the concrete program states into equivalence classes based on the values they give to a finite collection of predicates. We choose *abstraction predicates* $P_1, \ldots, P_n$, and then abstract each state s to the formula $\Psi \triangleq \Psi_1 \wedge \ldots \wedge \Psi_n$ where

$$\Psi_i \triangleq \begin{cases} P_i \text{ if } P_i \text{ is true in s} \\ \neg P_i \text{ if } P_i \text{ is false in s} \end{cases}$$

(Throughout, we use $\triangleq$ for meta-equality when we wish to distinguish this from the equality predicate inside a logic, which is always written $=$.) Such formulae are called *monomials.* The meaning of the monomial $\Psi \triangleq \Psi_1 \wedge \ldots \wedge \Psi_n$, i.e. the concretisation $\gamma(\Psi)$, is simply the set of states in which the formula Ψ holds. Each monomial over n abstraction predicates can be succinctly represented as a vector of n bits, where the ith bit records the polarity of P_i.

Transitions between these abstract states can be calculated using a satisfiability checker for the logic in which the Φ_is are written. Hence, predicate abstraction combines model checking with verification by theorem proving.

Example 2.1. Returning to our statement `x := x - 1` in a program with a single integer variable, let us choose two abstraction predicates $P_1 \triangleq x^2 = 1$ and $P_2 \triangleq x > 0$. Using x_0 to denote the value of the variable x before executing the statement, the formula $x = x_0 - 1$ expresses the statement's effect. There is a transition from $P_1 \wedge P_2$ to $\neg P_1 \wedge \neg P_2$ because the formula

$$\begin{array}{ll} & (P_1 \wedge P_2))[V \backslash V_0] \wedge (\neg P_1 \wedge \neg P_2) \wedge x = x_0 - 1 \\ \triangleq & {x_0}^2 = 1 \wedge x_0 > 0 \wedge \neg x^2 = 1 \wedge \neg x > 0 \wedge x = x_0 - 1 \end{array}$$

is satisfiable, by the concrete states $x_0 = 1$ and $x = 0$. (We write $[V \backslash V_0]$ for substitution of every free variable v by its "initial" counterpart v_0.) On the other hand, a transition to $\neg P_1 \wedge P_2$ is not possible, since the following is unsatisfiable.

$$\begin{array}{ll} & (P_1 \wedge P_2))[V \backslash V_0] \wedge (\neg P_1 \wedge P_2) \wedge x = x_0 - 1 \\ \triangleq & {x_0}^2 = 1 \wedge x_0 > 0 \wedge \neg x^2 = 1 \wedge x > 0 \wedge x = x_0 - 1 \end{array}$$

SLAM [BR01] (now the basis of Microsoft's Static Driver Verifier [Mic04]) and the similar BLAST [HJMS02] implement predicate abstraction for C programs. Recursion is handled properly by constructing a *summary* of each procedure, which is then shared across all call sites.

These tools work well for checking (control-dominated) interface usage properties of device drivers, and require no provision of invariants by the user, but are ineffective when it comes to programs manipulating linked data structures such as linked lists and trees. To treat such structures effectively, we need to reason about reachability in an object graph: when inserting a node n into a list, for instance, the postcondition may state that "n is reachable from program variable v by following a sequence of 'next' pointers". It is well known that first order logic (FO), on which SLAM and BLAST are based, is unable to express such properties.

In the next subsection we look at one approach to describing object graphs, namely using logics with reachability operators. Others include graph grammars [FM97] and using judiciously chosen local invariants and "ghost fields" [MN05].

2.3. Describing object graphs – logics with reachability constructs

We can add reachability to first order logic by allowing *transitive closure* formulae of the form

$$TC_{[a,b]}\,[\Phi(a,b)]\,(x,y)$$

which are true just when there is some finite sequence of points starting at x and ending at y, and such that for each point a in the sequence the point b following it satisfies $\Phi(a,b)$. The resulting logic is called first order logic with transitive closure, or FO(TC) (e.g. [Imm87]). FO(TC) is desirable because it is very expressive; we can write conditions like

Only objects transitively reachable from x by f fields have had their g fields modified (recall that v_0 denotes the previous value of v):

$$\forall o \quad g(o) \neq g_0(o) \quad \rightarrow \quad TC_{[a,b]}\,[f(a)=b]\,(x,o)$$

Although there exists no complete proof procedure for FO(TC), recent work [LAIR$^+$05] suggests that one can do effective reasoning for FO(TC) using a first order theorem prover, by heuristically selecting a set of first order axioms which soundly describe transitive closure. Alternatively, one can play the customary game of carefully restricting the logic and/or the class of models, hoping to find a logic which is decidable yet sufficiently powerful to be worthwhile. A variety of decidable logics with reachability are known, such as

- WS2S, essentially a weak second order logic for trees [KM01], can only handle tree-like models and has high complexity, but is very expressive. The Pointer Assertion Logic Engine (PALE) discussed in subsection 2.4 is based on WS2S.
- $\exists\forall(DTC^+[E])$, a decidable subset of FO(TC) where one is allowed to take the transitive closure of a single binary relation symbol E, has the main advantage that it can describe data structures more general than trees [IRR$^+$04].
- The guarded fixed point logic μGF [GW99] (which includes the modal μ-calculus) can also handle more general data structures, but can only express limited kinds of reachability.

An issue not explored here is that in some of these logics we can derive weakest preconditions for program statements, and in some we cannot.

Yet a third approach is to avoid theorem proving and decision procedures altogether and take a model-based approach, using 3-valued models to represents sets of 2-valued ones. This is how the TVLA system discussed in subsection 2.5 works.

2.4. PALE

The Pointer Assertion Logic Engine (PALE) [MS01] is used to verify that procedures manipulating graph type data structures preserve their consistency. A *graph type* data structure [MS01] consists of some acyclic tree *backbones* augmented by some well-behaved "extra" pointers governed by a datatype invariant. A linked list where each node has a pointer to the last node is a graph type, as is a binary tree where the leaves are threaded into a cyclic list.

Graph type stores can be encoded conveniently as models of the tree logic WS2S because the tree structure needed to handle the backbones is already built in. PALE accepts programs in a C-like language, ignoring arithmetic statements. The programmer must provide loop invariants and a special graph type declaration for each type used, such as the one for linked lists in Figure 2.

The 'next' fields form the backbone, and the 'prev' fields are extra pointers, constrained by the declaration `pointer prev:Node[this^Node.next={prev}]` to be the inverses of the 'next' fields (`^` is the "backwards" operator, so `this^Node.next` denotes starting at `this` and going one step backwards along field `Node.next`). PALE generates verification conditions in WS2S and sends them to the MONA tool [KM01] which decides them using tree automata. Thus – within its limited domain of application – the shape analysis of PALE is utterly precise.

```
type Node = {
  bool value;
  data next:Node;
  pointer prev:Node[this^Node.next={prev}];
}
```

Fig. 2. An example of a graph type declaration for the PALE tool. Here we declare nodes to contain a boolean data field `value` and pointer fields 'next' and 'prev', which we constraint to be inverses of each other with the declaration `pointer prev:Node[this^Node.next={prev}]`. (Here `^Node.next` indicates a backward step along the 'next' field.)

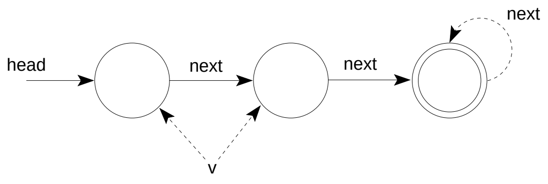


Fig. 3. An abstract heap, representing a linked list of length ≥ 3, with v pointing to the first or second element, or null

2.5. TVLA

TVLA [LAMS04], the Three-Valued Logic Analyser, is similar to PALE in that it is a system for tracking the shapes of object graphs, abstracting away data fields such as integers. TVLA gives only approximate results whereas PALE is fully precise, but on the other hand it can handle arbitrary object graphs and can infer loop invariants so that these need not be provided.

TVLA treats an object graph as a model of a predicate logic with unary and binary predicates. The domain of interpretation represents the set of allocated objects. For each (object-typed) program variable v there is a unary predicate V which holds only at the object pointed to by v. Similarly, pointer fields are represented by binary predicates.

To give the semantics for a statement S one provides an *update rule* for each predicate that S changes. These rules express the values of the predicates after execution of S in terms of their values beforehand and can use transitive closure. For instance `v := u.f` has the update rule $V(o) = \exists p(U(p) \wedge F(p, o))$.

Abstraction is achieved by moving to a 3-valued logic, where there is an extra truth value *Unknown* (or $\frac{1}{2}$) in addition to the usual *True* and *False*. In abstract states, such as the one in Figure 3, predicates may take the value *Unknown*, which is depicted as a dashed line. *Summary nodes*, drawn with a double circle, represent a whole group of one or more concrete nodes. The abstract heap in Figure 3 represents all linked lists of length three or more starting at 'head' and where v points to the first or second element or is null. Sound abstract transfer functions are obtained automatically simply by interpreting the update rules over three truth values rather than two.

2.6. Combinators for abstractions

This section outlines research on defining and implementing combinations of abstractions, properly called *products*. The general situation is that one has two abstract lattices L_1, L_2 with concretisation functions γ_1, γ_2 and abstraction functions α_1, α_2. For the sake of illustration we consider the product of the sign and parity lattices in Figure 1.

The simplest kind of product is the *direct product* [CC79, CMB$^+$95], where one takes as elements of the product lattice all the pairs $(a, b) \in A \times B$. The concretisation of (a, b) is simply $\gamma_1(a) \cap \gamma_2(b)$. Unfortunately this may introduce many redundant elements. What does $(\text{odd}, 0)$ mean? Since the intersection of the odd integers and $\{0\}$ is empty, it means the same as $(\bot, \bot)$. The latter is more precise when taken one component at a time, so we want to use it instead whenever we see $(\text{odd}, 0)$. Similarly $(\top, 0)$ should be reduced to $(\text{even}, 0)$. Removing the redundant elements in this way gives the desirable and well-behaved *reduced product* [CC92].

How should we construct the abstract transfer functions on the reduced product? One possibility is to simply apply pointwise the transfer functions provided by A and B, giving $(a, b) \mapsto (f_1^{\#}(a), f_2^{\#}(b))$. For `x := x - 1` we have $f_1^{\#}(\text{even}) = \text{odd}$ and $f_2^{\#}(0) = -$, and hence $(\text{even}, 0) \mapsto (\text{odd}, -)$. This amounts (almost) to running the two analyses in parallel *with no interaction between them.*

We can do better, however, if we *allow A and B to interact cooperatively.* The pointwise method gives $(\text{even}, +) \mapsto (\text{odd}, \top)$. But we know that if an integer n is even and $n > 0$, then $n \geq 2$. Therefore $n-1$ must be positive and odd, and the more precise $(\text{even}, +) \mapsto (\text{odd}, +)$ is preferable. Such cooperation is formalised by intersecting the concretisations from the underlying lattices A and B, that is, by defining

$$(a, b) \mapsto \big(\alpha_1(S), \alpha_2(S)\big) \text{ where } S = f(\gamma_1(a) \cap \gamma_2(b))$$

The key question is how to *implement* such cooperation. The preceding definition does not tell us how – it is written in terms of operations on the concrete lattices and in general we cannot compute with these. Implementing the transfer functions directly is difficult and *non-modular*:

- The implementor must have complete knowledge of the structures of all the lattices, of which there may in general be $n > 0$
- The constraints encoded by elements of the various lattices may interact in subtle ways, making the implementation hard to get right
- The implementation needs to be re-done every time we change the combination of analyses used.

2.7. Open products

The *open product* [CLCVH00] attempts to combine abstractions automatically and modularly, which the reduced product does not, while still allowing cooperation between them.

The fundamental idea is to allow abstraction schemes to exchange information via a system of queries. We fix a set Q of queries about the program state. Each lattice L is now endowed with a query-answering function $I : Q \times L \to \{\mathit{True}, \mathit{Unknown}, \mathit{False}\}$ where $I(q, a)$ can be *True* (resp. *False*) if the query q is true (resp. false) in all concrete states represented by a, and is *Unknown* otherwise. Abstract transfer functions now take the query-answering function provided by the other lattice as an extra argument, and may use it to produce more precise results. In the example of the previous subsection, if the abstract transfer function $f_2^{\#}$ of the sign analysis was able to send the query "Is it possible that x is 1?" to the parity analysis, we would get the desired result $f_2^{\#}(+) = +$. In [CLCVH00] open products are applied to the (shallow) optimisation of logic programs, but it appears that they have not been used with imperative programs, or for verification.

2.8. Related work

The Hob project [KLZR05] also employs multiple analysis plugins for verification, applying each where it is needed. It shares some of the goals of our work, such as making the best use of analysis techniques with narrow domains of applicability (e.g. PALE). The analyses in Hob also exchange information using a common logic, but using the theory of boolean algebra with Presburger arithmetic [KNR05] instead of FO(TC).

An important difference, however, is that in Hob exactly one plugin is used for each "module" of the source program, and interaction occurs only at the module boundaries and not for each program statement. The authors strive to "decouple" the analyses whereas we seek to integrate them more tightly, to increase precision. Similar themes are found also in [NEFE03, CL05, Hub03, SYY03].

3. Formal framework for interactive plugin-based analysis

In this section we define our interactive analysis framework precisely. We begin by specifying the class of programs we target. We provide small-step operational semantics for these programs, which we have formalised in the higher order logic (HOL) proof assistant Isabelle [NPW02].

We then introduce the notion of an analysis plugin, including the interface which plugins must implement, and requirements on plugins which will give rise to a sound analysis (we have also formalised these in Isabelle/HOL). We define a combination operator on plugins, which makes them work cooperatively, sharing information using FO(TC).

To complete the development we present our algorithm for plugin-based interprocedural analysis, and sketch our proofs of termination and soundness results.

```
static method Fact(n)
  {
    vars i, f;

    i := n;
    f := 1;

    while (i > 0)
      {
        f := f * i;
        i := i - 1;
      }

    assert f >= 1;

    return f;
  }
```

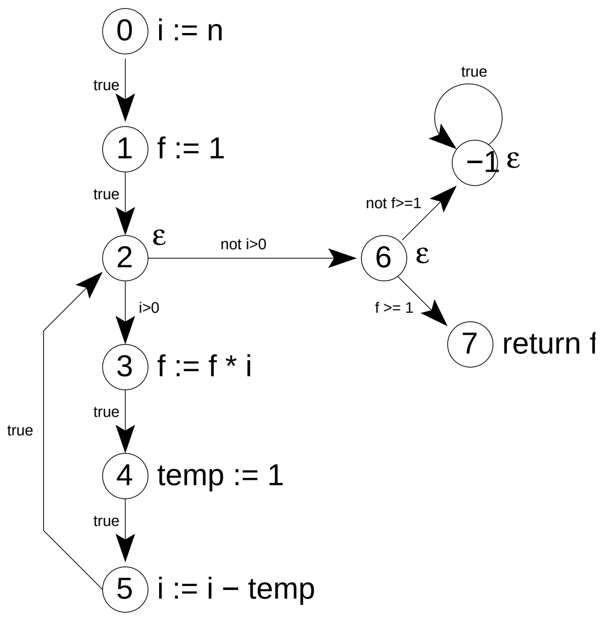


Fig. 4. An example method, for calculating factorials. The left part shows the method as it might be represented in source code, annotated with a behavioural assertion; the right part shows the control flow graph which we work with.

3.1. Representation of programs

Here we define precisely the kind of programs to be analysed in our framework. Rather than dealing with source code and its associated nuisances, we shall simply define programs in terms of their flow control graphs.

Fix once and for all countably infinite disjoint sets *VarNames*, *FieldNames*, *MethodNames* and *ClassNames*, and a distinguished element $this \in VarNames$.

Definition 3.1. A *statement* has one of the following forms:

- $u := v$ (variable copying)
- $u := n$ (assign integer constant)
- $u := v_1 \otimes v_2$ (arithmetic)
- $u := v.f$ (field read)
- $u.f := v$ (field write)
- $u :=$ New c (object creation)

- $u := v.m(p_1, \ldots, p_n)$ (method call)
- return v (method return)
- ε (do nothing)

where $u, v_i, p_i \in VarNames$, $m \in MethodNames$, $c \in ClassNames$ and throughout the paper $\otimes \in \{+, -, \times\}$. We call the first six forms above *assignment statements*.

Definition 3.2. A *control flow graph* is a graph as in Figure 4. Ordinary nodes are numbered and labelled with statements, and edges are labelled with *guards*, which are formulae of the program logic $\mathscr{L}$ (whose definition we defer to a later subsection 3.3). Nodes labelled with return statements must have no out-edges, and those labelled with call statements must have one out-edge labelled with *True*. Other nodes can have either a single out-edge with guard *True*, or two out-edges, one labelled with some condition P and the other with $\neg P$. A special *error node* is always numbered -1, labelled with ε and has a self-loop.

For a control flow graph G, the notation $G \ni n_1, s \xrightarrow{\Phi} n_2$ will denote that in the graph G there is an edge with guard Φ from a node numbered n_1 and labelled with statement s to a node numbered n_2. (We will omit s when we aren't interested in the label.)

The left of Figure 4 shows a small example program, written in a Java-like syntax that would be given to a compiler, and annotated with a behavioural assertion. The right hand side shows the corresponding control flow graphs which we work with. These are close to what a compiler would produce internally from the source code. Because our control flow graphs may contain only a minimal set of statement forms, there is not necessarily an exact correspondence between statements in the source code and those in the graph. Note also how edge guards are used to encode both control statements and assertions, the latter by means of guarded edges to the error node.

Definition 3.3. A *method* is a 4-tuple $M = (m, [p_1, \ldots, p_j], [l_1, \ldots, l_k], G)$ where: $m \in \mathit{MethodName}$ is the method's name, $p_i \in \mathit{VarNames}$ are the formal parameters, $l_i \in \mathit{VarNames}$ are its local variables and G is its control flow graph.

We will use appropriately named projection functions when dealing with tuples: the functions Π^M_{Name}, Π^M_{Formals}, Π^M_{Locals} and Π^M_{Graph} will project the respective components of a method.

Definition 3.4. A *class* is a 3-tuple $C = (c, [f_1, \ldots, f_j], [M_1, \ldots, M_k])$ where: $c \in \mathit{ClassNames}$ is the class' name, $f_i \in \mathit{FieldNames}$ are its fields and M_i are its methods.

The corresponding projection functions are Π^C_{Name}, Π^C_{Fields} and Π^C_{Methods}.

Definition 3.5. A *program* is a list of classes. We will write $\mathit{Methods}(P)$ for the set of all methods in all classes of P.

In order to be able to define the semantics of programs fully precisely, which use of the proof assistant Isabelle (and good taste, generally) requires us to do, we define fourteen healthiness conditions which make a program *well-formed*. In the main these conditions are unsurprising (a program cannot contain two classes with the same name, statements cannot refer to variables which don't exist in the current scope, etc.) and are already met by programs accepted by compilers.

However, one condition deserves special mention: Since we have chosen not to address inheritance and subtyping of classes, we insist that no two distinct classes declare methods or fields of the same name.

On a point of notation, since classes and methods are uniquely named in a well-formed program, we shall afford ourselves the liberty of writing c when we really mean "the class whose name is c", and m in place of "the method whose name is m".

3.2. Representation of program states

Definition 3.6. An *environment* is a function from $\mathit{VarNames}$ to $\mathbb{Z}$. Let Env be the set of all environments.

Definition 3.7. A *stack frame* is a triple (m, n, env) where $m \in MethodNames$ names the current method, $n \in \mathbb{N} \cup \{-1\}$ is the node number of the current control location (or -1 which is the special *error location*), and env is the environment. A *stack* is a non-empty list of stack frames whose head is taken to be the top of the stack. We name the respective projections Π^F_{Method}, $\Pi^M_{\mathit{Location}}$ and Π^M_{Env}.

Definition 3.8. An *object record* is a pair (c, F) where $c \in ClassNames$ says which class the object is an instance of, and F is a partial function from $\mathit{FieldNames}$ to $\mathbb{Z}$ giving values to the fields of the object. Let $ObjRec$ be the set of all object records. We name the respective projections Π^O_{Class} and Π^O_{Fields}.

Definition 3.9. A *heap* is a partial function from $\mathbb{N}_{>0}$ to $ObjRec$. We use 0 as the null address (and will use 0 and *null* interchangeably from now on). Let Heap be the set of all heaps.

Definition 3.10. Finally, a *program state* is a pair (s, h) where s is a stack and h is a heap. Let $State$ be the set of all program states. We name the respective projections Π^S_{Stack} and Π^S_{Heap}.

As with programs, we require that states satisfy additional conditions to be well-formed. These conditions are parametrised by the program P, and include for example, for the heap h: If $h(a) = (c, F)$ then P includes a class C with name c whose fields are exactly $\mathit{dom}(F)$. We have formalised these conditions in Isabelle, but as they are somewhat tedious we don't reproduce them here.

$$
\begin{array}{ll}
\text{term} ::= v & \text{(program variable)} \\
\quad \mid v_0 & \text{("initial" program variable)} \\
\quad \mid X & \text{(logical variable)} \\
\quad \mid n & \text{(integer constant)} \\
\quad \mid \mathit{null} & \text{(integer zero)} \\
\quad \mid \text{term} \otimes \text{term} & \text{(arithmetic)} \\
\quad \mid \text{term} \bullet \text{field} & \text{(field lookup)} \\
\quad \mid \text{term} \bullet \text{field}_0 & \text{(initial field lookup)}
\end{array}
$$

$$\text{literal} ::= \text{term} = \text{term} \mid \text{term} < \text{term} \mid \mathit{Allocd}^c(\text{ term }) \mid \mathit{Allocd}_0^c(\text{ term })$$

$$\Phi ::= \text{literal} \mid \mathit{True} \mid \neg\Phi \mid \Phi \wedge \Phi \mid \exists X\ \Phi \mid TC_{[A,B]}\,[\Phi(A,B)]\,(\mathit{term}, \mathit{term})$$

Fig. 5. Grammar of the logic $\mathscr{L}$ which describes program states. $\mathscr{L}$ is a first order logic with a transitive closure operator TC. Here $c \in \mathit{ClassNames}$.

3.3. Our logic for program states

We now present the logic $\mathscr{L}$ which is used to describe program states. In fact, $\mathscr{L}$ will do triple duty, being used to express:

- conditional statements in programs,
- assertions about desired program behaviour and
- information exchanged between cooperating analysis plugins.

We have chosen to use a first order logic with transitive closure, or FO(TC). This important design decision will be discussed in subsection 6.1. The syntax of the logic is given in Figure 5. We reserve the right to use standard abbreviations e.g. $\forall X\Phi$ for $\neg\exists X\neg\Phi$.

Formulae are interpreted over *pairs* of states, each consisting of an "earlier" and "later" state. This allows us to express the effect of a program statement by relating the state after its execution to the state before. Program variables subscripted with 0 refer to the earlier state, and those without to the later one. Similarly the $\mathit{Allocd}^c(x)$ predicate, which expresses that an object of class c is allocated at memory address x, comes in two versions. (Given a particular program, we will also write $\mathit{Allocd}(x)$ as shorthand for the disjunction of $\mathit{Allocd}^c(x)$ over all classes c.) Quantification is allowed only over *logical variables* which are kept separate from program variables and capitalised. Informally, the transitive closure operator TC works as follows: $TC_{[A,B]}\,[\Phi(A,B)]\,(t_1,t_2)$ says that from t_1 we can "reach" t_2 via some path of intermediate points, such that for each point A in the path the point B following it satisfies $\Phi(A,B)$.

Figure 6 defines the semantics $[\![-]\!]$. Rather than defining $[\![\Phi]\!]$ on $\mathit{State} \times \mathit{State}$ as might be expected, we first define it on $(\mathit{Env} \times \mathit{Heap}) \times (\mathit{Env} \times \mathit{Heap})$, since formulae can only describe local variables in the current scope and the heap. Local variables further up the stack cannot be described, nor can the control location. We then lift the semantics to full states in the obvious way.

Now, $(s_0, s) \in [\![\Phi]\!]_\rho$ means that Φ is true of the pair of states (s_0, s) where ρ gives values to the logical variables. When Φ contains no free variables we omit ρ. In some situations, such as when evaluating a guard Φ of a conditional statement, we have only one state to consider, and Φ will be free of 0-subscripts. In such cases we write simply $s \in [\![\Phi]\!]$.

3.4. Small-step program semantics

We define the semantics for each node of the program P as a binary relation on states: for a node in a method named m, numbered n and labelled with statement s, the relation $\mathit{nodeSem}(P, m, n, s) \subseteq \mathit{State} \times \mathit{State}$ relates states before execution at that node with the corresponding states afterwards. For example,

Example 3.11. The rule for applying an arithmetic operator $\otimes \in \{+, -, \times\}$ is:

Interpretation of terms:

$$\begin{array}{lcl}
I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, v) & = & \mathit{env}(v) \\
I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, v_0) & = & \mathit{env}(v_0) \\
I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, X) & = & \rho(X) \\
I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, n) & = & n \\
I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, \mathit{null}) & = & 0 \\
I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_1 \otimes t_2) & = & I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_1) \otimes I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_2)
\end{array}$$

$$I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t \bullet f) = \begin{cases} \left(\Pi^O_{\mathit{Fields}}(h(x))\right)(f) & \text{if } x \in \mathit{dom}(h) \text{ and } f \in \mathit{dom}\left(\Pi^O_{\mathit{Fields}}(h(x))\right) \\ \mathit{arbitrary} & \text{otherwise} \end{cases}$$
$$\text{where } x = I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t)$$

$$I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t \bullet f_0) = \begin{cases} \left(\Pi^O_{\mathit{Fields}}(h_0(x))\right)(f) & \text{if } x \in \mathit{dom}(h_0) \text{ and } f \in \mathit{dom}\left(\Pi^O_{\mathit{Fields}}(h_0(x))\right) \\ \mathit{arbitrary} & \text{otherwise} \end{cases}$$
$$\text{where } x = I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t)$$

Semantics of formulae:

$$\begin{array}{lll}
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![t_1 = t_2]\!]_\rho & \text{iff} & I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_1) = I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_2) \\
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![t_1 < t_2]\!]_\rho & \text{iff} & I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_1) < I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_2) \\
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![\mathit{Allocd}^c(t)]\!]_\rho & \text{iff} & \exists o \text{ s.t. } h\left(I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t)\right) = o \text{ and } \Pi^O_{\mathit{Class}}(o) = c \\
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![\mathit{Allocd}^c_0(t)]\!]_\rho & \text{iff} & \exists o \text{ s.t. } h_0\left(I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t)\right) = o \text{ and } \Pi^O_{\mathit{Class}}(o) = c \\
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![\neg\Phi]\!]_\rho & \text{iff} & ((\mathit{env}_0, h_0), (\mathit{env}, h)) \notin [\![\Phi]\!]_\rho \\
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![\Phi_1 \wedge \Phi_2]\!]_\rho & \text{iff} & ((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![\Phi_1]\!]_\rho \cap [\![\Phi_2]\!]_\rho \\
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![\exists X \Phi]\!]_\rho & \text{iff} & \exists n \in \mathbb{Z} \text{ s.t. } ((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![\Phi]\!]_{\rho \oplus \{X \mapsto n\}}
\end{array}$$

$$\begin{array}{l}
((\mathit{env}_0, h_0), (\mathit{env}, h)) \in [\![TC_{[A,B]}\,[\Phi(A,B)]\,(t_1, t_2)]\!]_\rho \\
\qquad \text{iff for some } n_1, \ldots, n_k \in \mathbb{Z}: \\
\qquad I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_1) = n_1, I((\mathit{env}_0, h_0), (\mathit{env}, h), \rho, t_2) = n_k \\
\qquad \text{and for } i = 1 \ldots k-1, (s_0, s) \in [\![\Phi]\!]_{\rho \oplus \{A \mapsto n_i, B \mapsto n_{i+1}\}}
\end{array}$$

Fig. 6. Semantics of the logic $\mathscr{L}$ which describes program states. Formulae of $\mathscr{L}$ are interpreted over pairs of states, an "earlier" state and a "later" state, and thus can describe the effects of atomic statements. The operator $\oplus$ denotes function override.

$$\begin{array}{lcl}
\mathit{nodeSem}(m, n_1, u := v_1 \otimes v_2, P) & := & \{((m, n_1, e_1) : xs, h), ((m, n_2, e_2) : xs, h) \mid \\
 & & \quad 1.\ \Pi^M_{\mathit{Graph}}(m) \ni n_1 \xrightarrow{g} n_2 \\
 & & \quad 2.\ e_2 = e_1 \oplus \{u \mapsto e_1(v_1) \otimes e_1(v_2)\} \\
 & & \quad 3.\ (e_2, h) \in [\![g]\!] \\
 & & \}
\end{array}$$

Informally, this rule states that:

1. the control flow graph for the current method m contains an edge from the current location n_1 to the next location n_2, with guard g,
2. the new environment e_2 is the same as the old one, except it has been updated at the variable assigned to, u, with the result of the arithmetic operation, and
3. the guard g is met in the new state.

The heap h, the stack frames further up the stack, xs, and the current method m all remain unaffected.

The full set of rules is given in Figures 7 and 8. Transitions to error states occur only for field reads and writes, i.e. statements $u = x.f$ and $x.f = v$, when the variable x either does not point to any allocated object, or points to an object for which the field f is not defined. Successful terminations are given self-loops for convenience; because the error node also has a self-loop, this frees us from considering terminating runs without changing the reachable states of the program.

The (small step) semantics of the entire program, $\mathit{semantics}(P) \subseteq \mathit{State} \times \mathit{State}$ is just the union

$$\mathit{semantics}(P) := \bigcup_{m,n,s} \mathit{nodeSem}(P, m, n, s)$$

taken over all nodes in the program.

We show in Isabelle that the semantics $\mathit{semantics}(P)$ of a well-formed program P relates each well-formed state to exactly one state, which is also well-formed. Thus $\mathit{semantics}(P)$ is in effect a function on and into well-formed states. We find it cleaner to have this functionality as something which we *prove* from relational definitions, rather than assume from the outset, because this reassures us that our definitions are reasonable, and fits better with a formalisation in Isabelle/HOL.

Definition 3.12. A *trace* in program P is a pair (p_0, p) in the transitive closure $\mathit{semantics}^*(P)$ where p_0 is well-formed. (It follows automatically that p is well-formed).

Here we represent traces with their first and last states. This differs slightly from the standard representation of traces as sequences of states, but we find it more convenient, and one can always conjour up the intermediate states where necessary by appealing to the definition of transitive closure.

Definition 3.13. A (Φ, m)-*trace* in P is a trace (p_0, p) where $\Pi^S_{\mathit{Stack}}(p_0) = [(m, 0, e)]$ for some environment e, and $p_0 \in [\![\Phi]\!]$. (Informally, the trace starts with an invocation of method m in a state described by Φ.) Additionally we say the trace is *to state* p.

3.5. Our interface for analysis plugins

In this subsection we present and discuss the notion of an *analysis plugin* which is central to our work. Intuitively an analysis plugin is a program analysis tool which has been appropriately wrapped for integration into our system.

We wish our plugins to do analysis in a local fashion, i.e. to transform abstract values based on a single statement at a time. We enforce this discipline by not giving plugins access to the global program, just the current statement and a limited amount of global information, which we call the *context* and define as follows:

Definition 3.14. The *context* of a node n in the program P comprises:

1. the name of the method containing n
2. a list of the classes in the program and their fields
3. a list of the methods in each class, and their formal parameters and local variables

We use *Context* to indicate the set of all contexts.

Definition 3.15. An *analysis plugin* is a module implementing the interface of Figure 9. Informally the role of each interface component is as follows (we will ignore the type *Config* and the function χ for now):

$$\mathit{nodeSem}(m, n, \varepsilon, P) := \{((m, n, e) : xs, h)\ ,\ ((m, n, e) : xs, h)\}$$

$$\begin{aligned}\mathit{nodeSem}(m, n_1, u := v, P) := \{&((m, n_1, e_1) : xs, h)\ ,\ ((m, n_2, e_2) : xs, h)\ |\\ &1.\ \Pi^M_{\mathit{Graph}}(m) \ni n_1 \xrightarrow{g} n_2\\ &2.\ e_2 = e_1 \oplus \{u \mapsto e_1(v)\}\\ &3.\ (e_2, h) \in [\![g]\!]\\ \}&\end{aligned}$$

$$\begin{aligned}\mathit{nodeSem}(m, n_1, u := n, P) := \{&((m, n_1, e_1) : xs, h)\ ,\ ((m, n_2, e_2) : xs, h)\ |\\ &1.\ \Pi^M_{\mathit{Graph}}(m) \ni n_1 \xrightarrow{g} n_2\\ &2.\ e_2 = e_1 \oplus \{u \mapsto n\}\\ &3.\ (e_2, h) \in [\![g]\!]\\ \}&\end{aligned}$$

$$\begin{aligned}\mathit{nodeSem}(m, n_1, u := v_1 \otimes v_2, P) := \{&((m, n_1, e_1) : xs, h)\ ,\ ((m, n_2, e_2) : xs, h)\ |\\ &1.\ \Pi^M_{\mathit{Graph}}(m) \ni n_1 \xrightarrow{g} n_2\\ &2.\ e_2 = e_1 \oplus \{u \mapsto e_1(v_1) \otimes e_1(v_2)\}\\ &3.\ (e_2, h) \in [\![g]\!]\\ \}&\end{aligned}$$

$$\begin{aligned}\mathit{nodeSem}(m, n_1, u := v \bullet f, P) := \{&((m, n_1, e_1) : xs, h)\ ,\ ((m, n_2, e_2) : xs, h)\ |\\ &\text{There exists } o \in \mathit{ObjRec} \text{ s.t.}\\ &1.\ \Pi^M_{\mathit{Graph}}(m) \ni n_1 \xrightarrow{g} n_2\\ &2.\ h(e_1(v)) = o\\ &3.\ f \in \mathit{dom}\left(\Pi^O_{\mathit{Fields}}(o)\right)\\ &4.\ e_2 = e_1 \oplus \left\{u \mapsto \left(\Pi^O_{\mathit{Fields}}(o)\right)(f)\right\}\\ &5.\ (e_2, h) \in [\![g]\!]\\ \}&\\ \bigcup&\\ \{&((m, n_1, e) : xs, h)\ ,\ ((m, -1, e) : xs, h)\ |\\ &e(v) \notin \mathit{dom}(h) \text{ or there exists } o \in \mathit{ObjRec} \text{ s.t.}\\ &h(e(v)) = o \text{ and } f \notin \mathit{dom}\left(\Pi^O_{\mathit{Fields}}(o)\right)\\ \}&\end{aligned}$$

Fig. 7. Small-step semantics of programs. The meaning of each program statement is expressed as a binary relation on the set of concrete states. The remainder of the rules are shown in Figure 8.

$\mathit{nodeSem}(m, n_1, u \bullet f := v, P) \quad := \quad \{((m, n_1, e) : xs, h_1) \ , \ ((m, n_2, e) : xs, h_2) \mid$

There exist $o_1, o_2 \in \mathit{ObjRec}$ s.t.

1. $\Pi^M_{\mathit{Graph}}(m) \ni n_1 \xrightarrow{g} n_2$
2. $h_1(e(u)) = o_1$
3. $f \in \mathit{dom}\left(\Pi^O_{\mathit{Fields}}(o_1)\right)$
4. $h_2 = h_1 \oplus \{e(u) \mapsto o_2\}$
5. $\Pi^O_{\mathit{Class}}(o_1) = \Pi^O_{\mathit{Class}}(o_2)$
6. $\Pi^O_{\mathit{Fields}}(o_2) = \Pi^O_{\mathit{Fields}}(o_1) \oplus \{f \mapsto e(v)\}$
7. $(e, h_2) \in [\![g]\!]$

$\}$

$\bigcup$

$\{((m, n_1, e) : xs, h) \ , \ ((m, -1, e) : xs, h) \mid$

$e(u) \notin \mathit{dom}(h)$ or there exists $o \in \mathit{ObjRec}$ s.t.

$h(e(u)) = o$ and $f \notin \mathit{dom}\left(\Pi^O_{\mathit{Fields}}(o)\right)\}$

$\mathit{nodeSem}(m, n_1, u := \text{ New } c, P) \quad := \{((m, n_1, e_1) : xs, h_1) \ , \ ((m, n_2, e_2) : xs, h_2) \mid$

There exist $o \in \mathit{ObjRec}$ and a s.t.

1. $\Pi^M_{\mathit{Graph}}(m) \ni n_1 \xrightarrow{g} n_2$
2. $a > 0$
3. $a \notin \mathit{dom}(h_1)$
4. $\forall a'$, if $0 < a' < a$ then $a' \in \mathit{dom}(h_1)$
5. $h_2 = h_1 \oplus \{a \mapsto o\}$
6. $e_2 = e_1 \oplus \{u \mapsto a\}$
7. $\Pi^O_{\mathit{Class}}(o) = c$
8. $\Pi^O_{\mathit{Fields}}(o)$ is defined exactly at $\Pi^C_{\mathit{Fields}}(c)$ and is constantly 0 there
8. $(e_2, h_2) \in [\![g]\!]$

$\}$

$\mathit{nodeSem}(m_1, n_1,$
$u := v.m_2(p_1, \ldots, p_k), P) \quad := \quad \{((m_1, n_1, e_1) : xs, h) \ , \ ((m_2, 0, e_2) : (m_1, n_1, e_1) : xs, h) \mid$

1. $j = k$
2. $e_2(x) = \begin{cases} e_1(p_i) & \text{if } x = f_i \\ e_1(v) & \text{if } x = \mathit{this} \\ 0 & \text{otherwise.} \end{cases}$

where $[f_1, \ldots, f_j] = \Pi^M_{\mathit{Formals}}(m_2)$

$\}$

$\mathit{nodeSem}(m_1, n_1, \text{ return } r, P) \quad := \quad \{((m_1, n_1, e_1) : (m_2, n_0, e_0) : xs, h) \ , \ ((m_2, n_2, e_2) : xs, h) \mid$

1. $\Pi^M_{\mathit{Graph}}(m_2) \ni n_0, (u = v.m(p_1, \ldots, p_k)) \xrightarrow{g} n_2$ for some $v, m, p_1, \ldots, p_k$
2. $e_2 = e_0 \oplus \{u \mapsto e_1(r)\}$
3. $(e_2, h) \in [\![g]\!]$

$\}$

$\bigcup \{((([(m_1, n_1, e)], h) \ , \ ([(m_1, n1_, e)], h))\}$

Fig. 8. The remaining rules of the small-step semantics of programs (continues from Figure 7).

datatype T (* type of abstract values *)
datatype *Config* (* type of the settings to configure the plugin *)

consts

(* concretisation function: gives meaning to abstract values - never actually implemented *)
$\gamma : \mathit{Context} \times T \to \mathbb{P}(\mathit{Env} \times \mathit{Heap})$

(* characteristic function of configurations - says which abstract values go with a given configuration *)
$\chi : \mathit{Config} \to \mathbb{P}(T)$

(* 'share' provides information for other plugins, about the execution of a given statement *)
(* 'succ' computes abstract successors for the execution of a given statement *)
$\mathit{share} : \mathit{Config} \times \mathit{Context} \times T \times \mathit{Statement} \to \mathscr{L}$
$\mathit{succ} : \mathit{Config} \times \mathit{Context} \times T \times \mathit{Statement} \times \mathscr{L} \to \mathbb{P}(T)$

(* These are like 'share' and 'succ' but for method calls. The parameters are
1. configuration in calling method
2. context of calling method
3. abstract state at call point
4. variable on which the method is invoked
5. name of called method
6. actual parameters
(7. formula of information shared by other plugins)
(8. configuration at called method) *)

$\mathit{share}_C : \mathit{Config} \times \mathit{Context} \times T \times \mathit{VarNames} \times \mathit{MethodNames} \times \mathit{VarNames}\ \text{list} \to \mathscr{L}$
$\mathit{succ}_C : \mathit{Config} \times \mathit{Context} \times T \times \mathit{VarNames} \times \mathit{MethodNames} \times \mathit{VarNames}\ \text{list} \times \mathscr{L} \times \mathit{Config} \to \mathbb{P}(T)$

(* These are like 'share' and 'succ' but for method returns. The parameters are
1. configuration in called method
2. context of called method
3. abstract state at return point
4. variable whose value is returned
5. configuration in calling method
6. name of calling method
7. abstract state at call point
8. variable used to store result in calling method
(9. formula of information shared by other plugins)

$\mathit{share}_R : \mathit{Config} \times \mathit{Context} \times T \times \mathit{VarNames} \times \mathit{Config} \times \mathit{MethodNames} \times T \times \mathit{VarNames} \to \mathscr{L}$
$\mathit{succ}_R : \mathit{Config} \times \mathit{Context} \times T \times \mathit{VarNames} \times \mathit{Config} \times \mathit{MethodNames} \times T \times \mathit{VarNames} \times \mathscr{L} \to \mathbb{P}(T)$

(* generate possible abstract states at start of execution *)
$\mathit{init} : \mathit{Config} \times \mathit{Context} \times \mathscr{L} \to \mathbb{P}(T)$

Fig. 9. The interface which analysis plugins must implement in order to be integrated into our system. The role of each component is discussed more fully in Definition 3.15.

- The datatype T is the type of the abstract values used by the plugin.
- The (notional) concretisation function γ gives meaning to the abstract values (elements of T), just as it did in subsection 2.1.
- Calling $\mathit{share}(\mathit{cf}, \mathit{ctxt}, a, s)$ asks the plugin to share an $\mathscr{L}$-formula Φ which is valid in all concrete states represented by the abstract state a (i.e. Φ is entailed by a) and might be useful to other plugins when computing successors for the statement s.
- Calling $\mathit{succ}(\mathit{cf}, \mathit{ctxt}, a, s, \Phi)$ computes the set of abstract states the program may reach by executing the assignment statement s in a concrete state represented by a and satisfying the formula Φ. In practice Φ will be the information gathered from the other plugins by *share*. (Thus *succ* is similar to the abstract transfer functions $f^{\#}$ of subsection 2.1.)
- The functions share_C and succ_C fulfill the same purpose as *share* and *succ*, except that they handle method calls instead of assignments, and take parameters describing the call.
- Returning from a method call is again treated similarly, by share_R and succ_R, except that two abstract values must be supplied instead of one: one describing the callee's state at the return point, and one describing the caller's state when the call was made. Approximately, constraints on the heap after the return are taken from the first, whereas constraints on the caller's local variables are taken from the second.
- *init* is used to start off the analysis; $\mathit{init}(\mathit{cf}, \mathit{ctxt}, \Phi)$ returns abstract values representing the possible initial states of the program, when it is started in context *ctxt* in a state satisfying Φ.

Note that plugins abstract only the current environment and the heap; the values of local variables further up the stack (i.e. in calling contexts) are not abstracted (and cannot be accessed by formulae of $\mathscr{L}$) because they do not influence execution in the current method. This approach allows us to use procedure summarisation (as in [BR00]) to analyse programs with recursive methods.

In general, plugins may be "reconfigurable" or "tunable". For predicate abstraction, for example, we may choose any finite set of abstraction predicates. Thus we make each plugin's functions parametric in a set *Config* of configurations; these are set on a per-method basis. The function χ is the "characteristic function" of configurations: it interprets a configuration to the subset of abstract values which are appropriate for that configuration.

Example 3.16. To illuminate the preceding definition we show how to frame the monomial predicate abstraction technique mentioned in subsection 2.2 as a plugin. All the plugin's components are collected in Figure 10.

A configuration for predicate abstraction is just a choice of n abstraction predicates $P_1, \ldots, P_n$. The plugin's abstract datatype contains all formulae which are conjunctive at the top level:

$$T := \{\Psi \mid \Psi \text{ is a conjunction of } \mathscr{L} - \text{formulae}\}$$

but for each configuration, only formulae made from that configuration's predicates are allowed:

$$\chi(\{P_0, \ldots, P_n\}) := \{\Psi_0 \wedge \ldots \wedge \Psi_n \mid \text{ each } \Psi_i \text{ is } P_i \text{ or } \neg P_i\}$$

Concretisation is easy: since the abstract values are formulae of $\mathscr{L}$, we map each to the set of states where it is true:

$$\gamma(\mathit{ctxt}, \Psi) := [\![\Psi]\!]$$

Defining *share* is likewise simple - we simply share the entire abstract value (replacing each variable v with the corresponding initial variable v_0, to indicate that we are describing the state before statement execution):

$$\mathit{share}(\mathit{cf}, \mathit{ctxt}, \Psi, s) := \Psi[V \backslash V_0]$$

Transitions between monomials are computed with a satisfiability checker as usual in predicate abstraction (see Example 2.1). To see whether executing statement s in a state described by Ψ might leave us in a state described by Θ, i.e. to see whether $\Theta_1 \wedge \ldots \wedge \Theta_n \in \mathit{succ}(\mathit{cf}, \mathit{ctxt}, \Psi, s, \Phi)$, we check the satisfiability of

$T := \{\Psi \mid \Psi \text{ is a conjunction of } \mathscr{L} - \text{formulae}\}$

$\chi(\{P_0, \ldots, P_n\}) := \{\Psi_0 \wedge \ldots \wedge \Psi_n \mid \text{ each } \Psi_i \text{ is } P_i \text{ or } \neg P_i\}$

$\gamma(\mathit{ctxt}, \Psi) := [\![\Psi]\!]$

$\mathit{init}(\{P_0, \ldots, P_n\}, \mathit{ctxt}, \Phi) := \{\Psi \triangleq \Psi_0 \wedge \ldots \wedge \Psi_n \mid \Psi \wedge \Phi \text{ is satisfiable and each } \Psi_i \text{ is } P_i \text{ or } \neg P_i\}$

$\mathit{share}(\mathit{cf}, \mathit{ctxt}, \Psi, s) := \Psi[V \backslash V_0]$

$\mathit{succ}(\{P_0, \ldots, P_n\}, \mathit{ctxt}, \Psi, s, \Phi) :=$
$\{\Psi' \triangleq \Psi'_0 \wedge \ldots \wedge \Psi'_n \mid \Psi' \wedge \Phi \wedge \mathit{Post}(s, \mathit{ctxt}) \text{ is satisfiable and each } \Psi'_i \text{ is } P_i \text{ or } \neg P_i\}$

$\mathit{share}_C(\mathit{cf}, \mathit{ctxt}, \Psi, v, m, [p_1, \ldots, p_k]) := \Psi[V \backslash V_0]$

$\mathit{succ}_C(\{P_0, \ldots, P_n\}, \mathit{ctxt}, \Psi, v, m, [p_1, \ldots, p_k], \Phi, \{P'_0, \ldots, P'_j\}) :=$
$\{\Psi' \triangleq \Psi'_0 \wedge \ldots \wedge \Psi'_j \mid \Psi' \wedge \Phi \wedge \mathit{Post}_C(v, m, [p_1, \ldots, p_k], \mathit{ctxt}) \text{ is satisfiable and each } \Psi'_i \text{ is } P'_i \text{ or } \neg P'_i\}$

$\mathit{share}_R(\{P_0, \ldots, P_n\}, \mathit{ctxt}, \Psi_{\mathit{callee}}, v_{\mathit{callee}}, \{P'_0, \ldots, P'_j\}, m_{\mathit{caller}}, \Psi_{\mathit{caller}}, v_{\mathit{caller}}) := (\Psi_{\mathit{callee}} \wedge \Psi_{\mathit{caller}})[V \backslash V_0]$

$\mathit{succ}_R(\{P_0, \ldots, P_n\}, \mathit{ctxt}, \Psi_{\mathit{callee}}, v_{\mathit{callee}}, \{P'_0, \ldots, P'_j\}, m_{\mathit{caller}}, \Psi_{\mathit{caller}}, v_{\mathit{caller}}, \Phi) :=$
$\{\Psi' \triangleq \Psi'_0 \wedge \ldots \wedge \Psi'_j \wedge \mid \Psi' \wedge \Phi \wedge \mathit{Post}_R(v_{\mathit{callee}}, v_{\mathit{caller}}, m_{\mathit{caller}}, \mathit{ctxt}) \text{ is satisfiable and each } \Psi'_i \text{ is } P'_i \text{ or } \neg P'_i\}$

Fig. 10. How to wrap the predicate abstraction technique from section 2.2 as a plugin. See Example 3.16 for explanation. ($[V \backslash V_0]$ indicates substitution of every free variable v by its "initial" counterpart v_0.)

$$(\Psi_1 \wedge \ldots \wedge \Psi_n)[V \backslash V_0] \wedge \mathit{Post}(s) \wedge \Theta_1 \wedge \ldots \wedge \Theta_n \wedge \Phi$$

The formula $\mathit{Post}(s, \mathit{ctxt})$ expresses the effect of statement s in context ctxt; e.g. if ctxt has only the variables x and y in scope,

$$\mathit{Post}(\texttt{x := x + 2}, \mathit{ctxt}) \quad \triangleq \quad x = x_0 + 2 \wedge y = y_0$$

Method calls and returns are treated similarly. For calls, the Post_C formula connects the actual and formal parameters, and for returns Post_R connects the value returned in the callee to the variable waiting to receive it in the caller. For example, if a method declared by `method m(a, b)` is called by `m(x, 10)` the *Post* formula will be $a = x_0 \wedge b = 10$. Recall that satisfiability of FO(TC) formulae is undecidable. However, we can still obtain a safe analysis, by assuming that formulae are satisfiable when we cannot show otherwise. This means that, when we cannot determine whether a given transition exists, we assume that it does.

When we come to apply our plugins to programs, we will need to address two concerns:

1. Will our analysis always terminate?
2. Will our analysis deliver sound results?

We now present conditions on plugins that will guarantee these desirable properties.

One way to ensure termination is to insist that the type T be finite, but we consider this to be too restrictive. Consider predicate abstraction: because there are infinitely many possible abstraction predicates, there are infinitely many monomials that can be formed from them. But given *any particular choice* of n abstraction predicates, there are only finitely many monomials (in fact 2^n of them), so termination is assured. Hence, we require χ to produce only finite subsets. Of course, we must make associated restrictions to ensure that the plugin's functions only return abstract values which are appropriate for the configuration; the condition for the *succ* function is

$$\forall \mathit{cf} \in \mathit{Config}, \mathit{ctxt} \in \mathit{Context}, a \in T, s \in \mathit{Statement}, \Phi \in \mathscr{L} : \mathit{succ}(\mathit{cf}, \mathit{ctxt}, a, s, \Phi) \subseteq \chi(\mathit{cf})$$

For all $cf, cf_1, cf_2 \in \mathit{Config}$, $\mathit{ctxt} \in \mathit{Context}$, $a, a_1, a_2 \in T$, $v, v_1, v_2 \in \mathit{VarName}$, $m \in \mathit{MethodName}$ and $\Phi \in \mathscr{L}$:

1. $\chi(cf)$ is finite
2. $\mathit{init}(cf, \mathit{ctxt}, \Phi) \subseteq \chi(cf)$
3. $\mathit{succ}(cf, \mathit{ctxt}, a, s, \Phi) \subseteq \chi(cf)$
4. $\mathit{succ}_C(cf_1, \mathit{ctxt}, a, v, m, \mathit{params}, \Phi, cf_2) \subseteq \chi(cf_2)$
5. $\mathit{succ}_R(cf_1, \mathit{ctxt}, a_1, v_1, cf_2, m, a_2, v_2, \Phi) \subseteq \chi(cf_2)$

Fig. 11. Finiteness axioms for plugins. These ensure termination of our plugin-based analysis.

Figure 11 gives these *finiteness axioms* in full.

We now turn our attention to soundness. Recall that the analyses carried out by our plugins will in general be approximate but conservative, rather than precise. In definition **??** we had conditions relating $f_i^{\#}$ and $\preccurlyeq$; here these are paralleled by the *soundness axioms* listed in Figure 12. We now illustrate those for assignment statements.

Suppose our analysis has reached a node n labelled with an assignment statement s, with context *Context* and configuration *cf*, and the abstract state is $a \in T$. Let c be any concrete state that a represents, i.e. let $(\mathit{env}, h) \in \gamma(a)$ where *env* and h are the current environment and heap in c (that is, $\mathit{env} = \Pi^F_{\mathit{Env}}(\mathit{head}(\Pi^S_{\mathit{Stack}}(c)))$ and $h = \Pi^S_{\mathit{Heap}}(c)$). (Recall that only the heap and the environment in the top stack frame are abstracted.) Let $(c, c') \in \mathit{semantics}(P)$, where state c' has environment env' and heap h'.

Firstly we need to be sure that the formula exported by the plugin, via *share*, really does describe the execution of s, i.e. we require

$$((\mathit{env}, h), (\mathit{env}', h')) \in [\![\mathit{share}(cf, \mathit{ctxt}, a, s)]\!]$$

This is condition B in Figure 12. Secondly, we need to know that, provided the formula imported from the other plugins is correct, the new abstract states computed by *succ* conservatively model the effect of the statement s:

If $((\mathit{env}, h), (\mathit{env}', h')) \in [\![\Phi]\!]$ then for some $a' \in \mathit{succ}(cf, \mathit{ctxt}, a, s, \Phi)$ we have $(\mathit{env}', h') \in \gamma(a')$

This is condition E. The conditions for calls, returns and initialisation are similar. (Although the soundness axioms are expressed in terms of states with minimal-depth stacks, of length one or two, we can effectively add in arbitrarily many extra stack frames, because the program semantics only refers to the top one or two.)

The machinery built up in this section does not include a treatment of *ghost fields* which are a popular means of specifying and reasoning about class behaviour (as in [MN05]). We intend to rectify this in future work, allowing each plugin to add ghost fields to classes.

3.6. Our worklist algorithm for interprocedural plugin-based analysis

Figures 13 and 14 give the worklist algorithm we use for interprocedural analysis. Based on [BR00] we use summarisation to handle recursive methods without requiring that methods be annotated with pre- and post-conditions. In the next subsection we will give a combination operator on plugins which makes them cooperate; thus it suffices here to present the algorithm with a single plugin.

The abstract states used in the analysis are of the form (m, n, a_0, a) where: the method name $m \in \mathit{MethodName}$ and node number $n \in \mathbb{Z}$ give the control location and $a \in T$ is one of the plugin's abstract values, representing the heap and environment. Abstract states also carry with them a very limited part of their history: the value $a_0 \in T$ records how the heap and environment looked when the current method was entered. This allows calls and returns to be matched up.

As the analysis proceeds, a set A of abstract states which approximates the reachable concrete states is built up and explored, using the worklist to keep track of which abstract states still need to be visited.

When finding the abstract successors of a method call (from line 21), it may be the case that A already

A If $(env, h) \in [\![\Phi]\!]$ then $\exists a \in init(cf, ctxt, \Phi)$ s.t. $(env, h) \in \gamma(ctxt, a)$.

B If:

1. $(([m, n_0, env_0], h_0)\ ,\ ([m, n, env], h)) \in semantics(P)$
2. in program P the method named m contains an edge $n_0, s \xrightarrow{True} n$ with s an assignment statement
3. $(env_0, h_0) \in \gamma(ctxt, a)$

then $((env_0, h_0), (env, h)) \in [\![share(cf, ctxt, a, s)]\!]$

C If:

1. $(([m, n, env], h)\ ,\ ([(m', 0, env'), (m, n, env)], h)) \in semantics(P)$
2. in program P the method named m contains a node numbered n and labelled with $u := v.m'(p_1, \dots, p_n)$
3. $(env, h) \in \gamma(ctxt, a)$

then $(env, h), (env', h)) \in [\![share_C(cf, ctxt, a, v, m', [p_1, \dots, p_n])]\!]$

D If:

1. $(([(m, n, env), (m', n', env'')], h)\ ,\ ([(m', n'', env')], h) \in semantics(P)$
2. in program P the method named m contains a node numbered n and labelled *return* r
3. in P the method named m' contains an edge $n', s \xrightarrow{True} n''$ where s has form $u := v.m(p_1, \dots, p_n)$
4. $(env, h) \in \gamma(ctxt, a)$
5. $(env', h) \in \gamma(ctxt', a')$

then $((env, h), (env'', h)) \in [\![share_R(cf, ctxt, a, r, cf', m', a', u)]\!]$

E If:

1. $(([m, n_0, env_0], h_0)\ ,\ ([m, n, env], h)) \in semantics(P)$
2. in program P the method named m contains an edge $n_0, s \xrightarrow{True} n$ with s an assignment statement
3. $(env_0, h_0) \in \gamma(ctxt, a_0)$
4. $((env_0, h_0), (env, h)) \in [\![\Phi]\!]$

then $\exists a \in succ(cf, ctxt, a_0, s, \Phi)$ s.t. $(env, h) \in \gamma(ctxt, a)$.

F If:

1. $(([m, n, env], h)\ ,\ ([(m', 0, env'), (m, n, env)], h)) \in semantics(P)$
2. in program P the method named m contains a node numbered n and labelled with $u := v.m'(p_1, \dots, p_n)$
3. $(env, h) \in \gamma(ctxt, a)$
4. $((env, h), (env', h)) \in [\![\Phi]\!]$

then $\exists a' \in succ_C(cf, ctxt, a, v, m', [p_1, \dots, p_n], \Phi, cf')$ s.t. $(env', h) \in \gamma(ctxt', a')$.

G If:

1. $(([(m, n, env), (m', n', env')], h)\ ,\ ([(m', n'', env'')], h) \in semantics(P)$
2. in program P the method named m contains a node numbered n and labelled *return* r
3. in P the method named m' contains an edge $n', s \xrightarrow{True} n''$ where s has form $u := v.m(p_1, \dots, p_n)$
4. $(env, h) \in \gamma(ctxt, a)$
5. $(env', h) \in \gamma(ctxt', a')$
6. $((env, h), (env', h)) \in [\![\Phi]\!]$

then $\exists a'' \in succ_R(cf, ctxt, a, r, cf', m', a', u, \Phi)$ s.t. $(env'', h) \in \gamma(ctxt', a'')$.

Fig. 12. Soundness axioms for plugins. These ensure that our plugin-based analysis gives correct results.

```
                    ▷ $P$ is the well-formed program to check. $m_0$ is the "main" method from which analysis starts.
                    ▷ The formula $\Phi_0$ constrains the initial state. $S$ configures the plugin (on a per-method basis).

    procedure ANALYSE($P : \mathit{Program}, m_0 : \mathit{MethodName}, \Phi_0 : \mathscr{L}, S : \mathit{MethodName} \to \mathit{Config}$)
 5:
        vars $\mathit{worklist}, A : \mathbb{P}(\mathit{MethodName} \times \mathbb{Z} \times T \times T)$

        let $I = \{(m_0, 0, a, a) \mid a \in \mathit{init}(S(m_0), \mathit{ctxt}(P, m_0), \Phi_0)\}$ in
            $\mathit{worklist} := I$
10:         $A := I$

        while $\mathit{worklist} \neq \emptyset$ do
            let
                $w \in \mathit{worklist}$
15:             $(m, n, a, a_0) = w$
            in
                $\mathit{worklist} := \mathit{worklist} - \{w\}$

                case (statement labeling node $n$ of method $m$) of
20:
                    method call: $u := v.m_c(p_1, \ldots, p_n) \Rightarrow$
                        let
                            $\mathit{callSuccessors} = \mathit{succ}_C(S(m), \mathit{ctxt}(P, m), a, v, m_c, [p_1, \ldots, p_n], \mathit{True}, S(m_c))$
                        in
25:                         if $\mathit{callSuccessors} = \emptyset$ then
                                $A := A - \{w\}$
                            else
                                let
                                    $n_r$ is the number of the return statement, return $v_r$, in $m_c$
30:                                 $n'$ is the target of the out-edge from node $n$ in $m$
                                    $\mathit{existingReturns} = \{\hat{a} \mid \exists \hat{a_0} \in \mathit{callSuccessors}$ s.t. $(m_c, n_r, \hat{a}, \hat{a_0}) \in A - \mathit{worklist}\}$
                                    $\mathit{returnSuccessors} =$
                                        $\bigcup_{\hat{a} \in \mathit{existingReturns}} \{(m, n', a', a_0) \mid a' \in \mathit{succ}_R(S(m), \mathit{ctxt}(P, m), a, u, S(m_c), m, \hat{a}, v_r, \mathit{True})\}$
                                in
35:                                 $\mathit{worklist} := \mathit{worklist} \cup (\{(m_c, 0, \hat{a_0}, \hat{a_0} \mid \hat{a_0} \in \mathit{callSuccessors}\} \cup \mathit{returnSuccessors} - A)$
                                    $A := A \cup \{(m_c, 0, \hat{a_0}, \hat{a_0} \mid \hat{a_0} \in \mathit{callSuccessors}\} \cup \mathit{returnSuccessors}$

                    method return: return $v \Rightarrow$
                        let
40:                         $\mathit{callerPoints} = \left\{(\bar{m}, \bar{n}, \bar{n}', u) \mid \mathit{Graph}(\bar{m}) \ni \bar{n}, u := \bar{v}.m(p_1, \ldots, p_n) \xrightarrow{\mathit{True}} \bar{n}' \text{ for some } \bar{v}, p_1, \ldots, p_n\right\}$
                            $\mathit{callerStates} = \{(\bar{m}, \bar{n}, \bar{a}, \bar{a_0}, \bar{n}', u) \mid (\bar{m}, \bar{n}, \bar{n}', u) \in \mathit{callerPoints}, (\bar{m}, \bar{n}, \bar{a}, \bar{a_0}) \in A,$ and
                                $a_0 \in \mathit{succ}_C(S(\bar{m}), \mathit{ctxt}(P, \bar{m}), \bar{a}, u, m, [p_1, \ldots, p_n], \mathit{True}, S(m))\}$
                            $\mathit{successors} =$
                                $\bigcup_{(\bar{m}, \bar{n}, \bar{a}, \bar{a_0}, \bar{n}', u) \in \mathit{callerStates}} \{(\bar{m}, \bar{n}', a', \bar{a_0}) \mid a' \in \mathit{succ}_R(S(m), \mathit{ctxt}(P, m), a, v, S(\bar{m}), \bar{m}, \bar{a}, u, \mathit{True})\}$
45:                     in
                            $\mathit{worklist} := \mathit{worklist} \cup (\mathit{successors} - A)$
                            $A := A \cup \mathit{successors}$

                            ...continued...
50:
```

Fig. 13. The pseudocode for our interprocedural plugin-based analysis algorithm, using a worklist and summarisation. Continues in Figure 14.

contains a complete abstract trace through the called method (resulting from invocation from another call site), in which case the resulting returns are processed straight away.

Unlike the algorithm in [BR00], A does not quite grow monotonically: when an abstract state is visited it may be found to be inconsistent (like the state $(\text{odd}, 0)$ from subsection 2.1), and can then be safely removed from A. The tests on lines 25, 57 and 68 detect this: if no abstract successors are generated, then there are no concrete successors (by the soundness axioms), and because we have already observed that the semantics are functional, the initial abstract state must be inconsistent. (This does *not* apply to successors of returns, because these are calculated from *two* abstract states — one in the caller and one in the callee — and it could just be that the corresponding caller state is not generated yet.)

```
        field read or write statement $s$ on variable $v$ (potentially dangerous) $\Rightarrow$
          let
            $edges = \left\{(n', \Phi) \mid Graph(m) \ni n \xrightarrow{\Phi} n'\right\}$
            $successors = \bigcup_{(n',\Phi)\in edges} \{(m, n', a', a_0) \mid a' \in succ\,(S\,(m)\,, ctxt\,(P, m)\,, a, s, \Phi \wedge Allocd_0(v_0))\}$
55:         $memErrorSuccessors = \{(m, -1, a', a_0) \mid a' \in succ\,(S\,(m)\,, ctxt\,(P, m)\,, a, \varepsilon, \neg Allocd_0(v_0))\}$
          in
            if $successors \cup memErrorSuccessors = \emptyset$ then
              $A := A - \{w\}$
            else
60:           $worklist := worklist \cup ((successors \cup memErrorSuccessors) - A)$
              $A := A \cup successors \cup memErrorSuccessors$

        any other statement $s \Rightarrow$
          let
65:         $edges = \left\{(n', \Phi) \mid Graph(m) \ni n \xrightarrow{\Phi} n'\right\}$
            $successors = \bigcup_{(n',\Phi)\in edges} \{(m, n', a', a_0) \mid a' \in succ\,(S\,(m)\,, ctxt\,(P, m)\,, a, s, \Phi)\}$
          in
            if $successors = \emptyset$ then
              $A := A - \{w\}$
70:         else
              $worklist := worklist \cup (successors - A)$
              $A := A \cup successors$

  return $A$
```

Fig. 14. The (rest of the) pseudocode for our interprocedural plugin-based analysis algorithm, using a worklist and summarisation. Continues from Figure 13. Procedure summaries are implicit in the set A which is built to over-approximate the reachable (concrete) states. Here $ctxt(P, m)$ and $Graph(m)$ give the context (Definition 3.14) and control flow graph respectively of the method named m in program P.

We deal with potentially dangerous memory accesses by splitting our analysis into two branches, one branch where the memory access succeeds (line 54), which we distinguish by allowing the plugin to assume $Allocd_0(v_0)$, and another branch where the memory access fails (line 55). In this second branch we give the plugin the formula $\neg Allocd_0(v_0)$, whereupon the plugin must try to close this branch by establishing that $\neg Allocd_0(v_0)$ contradicts the plugin's abstract state.

The following two theorems show that the algorithm terminates, and that when it does so, the set A which has been built "covers" or over-approximates the set of reachable concrete states.

Theorem 3.17. Termination: The algorithm in Figure 13 always terminates.

Proof (sketch): First we augment the code with an auxilliary variable B which is updated in the same way as A except that inconsistent states are not removed, and thus B grows monotonically. Define

$$N := |P| \times \left(\max_{m \text{ in } P} |\chi\,(S\,(m))|\right)^2$$

where $|P|$ is the number of nodes (i.e. control locations) in P. It can be seen that N is an upper bound for $|B|$. On each iteration, B increases or remains unchanged. When B stays the same, no new abstract states are generated and therefore the worklist becomes smaller (because one element was removed from it, and no new ones were added). Therefore the ranking function

$$f(B, worklist) := (N - |B|, |worklist|)$$

for the standard lexicographic ordering of $\mathbb{N}^2$ shows termination. ∎

Theorem 3.18. Soundness: Let $c = ((m, n, env) : xs, h)$ be a concrete state such that there exists a $(\Phi, m_0)-$trace to a c in P. Then, once the analysis has terminated, for some $a, a_0 \in T$ we have $(m, n, a_0, a) \in A$ and $(env, h) \in \gamma(a)$.

Proof (sketch): Let (c_0, c) be a (Φ, m_0)-trace in P. By the definition of transitive closure, there exists a sequence $c_1, c_2, \ldots, c_n = c$ such that for $0 < i \leq n, (c_i, c_{i+1}) \in \mathit{semantics}(P)$. We proceed by induction on the length n of this sequence.

If $n = 1$, then c must have the form $([(m, 0, e)], h)$, and $(e, h) \in [\![\Phi_0]\!]$. By the soundness axiom for *init*, there exists $a \in \mathit{init}(S(m_0), \mathit{ctxt}(P, m_0), \Phi_0)$ such that $(\mathit{env}, h) \in \gamma(a)$. We see that the algorithm adds $(m_0, 0, a, a)$ to A (and will not remove it, because it is consistent since its concretisation contains at least c_1), so we are done.

Now let $n > 1$. Let $c_{n-1} = ((m, n, e) : xs, h)$. Our induction hypothesis gives us $w = (m, n, a_{n-1}, a_0) \in A$ such that $(e, h) \in \gamma(a)$. There are now three cases, depending on what kind of statement labels the node numbered n in the method named n:

- Case 1 - Assignment statement s:
  Because the worklist is finally empty, at some iteration w was removed from the worklist. At this stage, by the soundness axiom for *succ*, an abstract state has been generated to "cover" c_n. Furthermore, this covering state will never be removed, because it is not inconsistent (it's concretisation includes at least c_n) and only inconsistent states are removed.
- Case 2 - Call statement: same reasoning as for case 1, using the soundness axiom for succ_C instead of for *succ*.
- Case 3 - Return statement:
  By going back through the sequence of concrete states, matching up calls and returns, we find the concrete state c_k, with $k \leq n - 1$, at which the call is made from which the present statement returns. By applying the induction hypothesis to c_k, we obtain $w' \in A$ which abstracts the call state c_k. Both w and w' are eventually removed from the worklist; there are two subcases: A. the removal of w happens later, and B. the removal of w' happens later. In both situations the soundness axiom for succ_R produces a covering a_n, as in cases 1 and 2.

In particular the previous theorem shows:

Corollary 3.19. If $\{(m', -1, a_0, a) \in A\} = \emptyset$ then executing m_0 in a state satisfying Φ never leads to an error.

We are currently working on mechanizing these proofs in Isabelle. This, together with proofs that our plugins meet the soundness axioms, opens up the possibility of generating machine-checkable proofs of program correctness from successful runs of the analyser (as in [SYY03], but using higher order logic instead of first order).

3.7. Combining analysis plugins

The following definition explains how to combine two plugins, to make them cooperate using FO(TC) as a common language; the result is another plugin, which can be combined again, or used in the analysis algorithm.

Definition 3.20. Given plugins $\mho_1$ and $\mho_2$, we construct their *combination* $\mho_1 \diamond \mho_2$ as:

- $T := T_1 \times T_2$
- $\mathit{Config} := \mathit{Config}_1 \times \mathit{Config}_2$
- $\gamma\left((\mathit{cf}_1, \mathit{cf}_2), (a_1, a_2)\right) := \gamma_1(\mathit{cf}_1, a_1) \cap \gamma_2(\mathit{cf}_2, a_2)$
- $\chi((\mathit{cf}_1, \mathit{cf}_2)) := \chi_1(\mathit{cf}_1) \times \chi_2(\mathit{cf}_2)$
- $\mathit{init}(P, (\mathit{cf}_1, \mathit{cf}_2), \mathit{ctxt}, \Phi) := \mathit{init}_1(\mathit{cf}_1, \mathit{ctxt}, \Phi) \times \mathit{init}_2(\mathit{cf}_2, \mathit{ctxt}, \Phi)$
- $\mathit{share}((\mathit{cf}_1, \mathit{cf}_2), \mathit{ctxt}, (a_1, a_2), s) := \mathit{share}_1(\mathit{cf}_1, \mathit{ctxt}, a_1, s) \wedge \mathit{share}_2(\mathit{cf}_2, \mathit{ctxt}, a_2, s)$
- $\mathit{succ}((\mathit{cf}_1, \mathit{cf}_2), \mathit{ctxt}, (a_1, a_2), s, \Phi) := \mathit{succ}_1(\mathit{cf}_1, \mathit{ctxt}, a_1, s, \Theta) \times \mathit{succ}_2(\mathit{cf}_2, \mathit{ctxt}, a_2, s, \Theta)$
  where $\Theta \triangleq \quad \Phi \wedge \mathit{share}((\mathit{cf}_1, \mathit{cf}_2), \mathit{ctxt}, (a_1, a_2), s)$

- $share_C((cf_1, cf_2), ctxt, (a_1, a_2), v, m, [p_1, \ldots, p_j]) :=$
  $share_{C1}(cf_1, ctxt, a_1, v, m, [p_1, \ldots, p_j]) \wedge share_{C2}(cf_2, ctxt, a_2, v, m, [p_1, \ldots, p_j])$

- $succ_C((cf_1, cf_2), ctxt, (a_1, a_2), v, m, [p_1, \ldots, p_j], \Phi, ctxt') :=$
  $succ_{C1}(cf_1, ctxt, a_1, v, m, [p_1, \ldots, p_j], \Theta, ctxt') \times succ_{C2}(cf_2, ctxt, a_2, v, m, [p_1, \ldots, p_j], \Theta, ctxt')$
  where $\Theta \triangleq \quad \Phi \wedge share_C((cf_1, cf_2), ctxt, (a_1, a_2), v, m, [p_1, \ldots, p_j])$

- $share_R((cf_1, cf_2), ctxt, (a_1, a_2), r, (cf'_1, cf'_2), m, (a'_1, a'_2), u) :=$
  $share_{R1}(cf_1, ctxt, a_1, r, cf'_1, m, a'_1, u) \wedge share_{R2}(cf_2, ctxt, a_2, r, cf'_2, m, a'_2, u)$

- $succ_R((cf_1, cf_2), ctxt, (a_1, a_2), r, (cf'_1, cf'_2), m, (a'_1, a'_2), u, \Phi) :=$
  $succ_{R1}(cf_1, ctxt, a_1, r, cf'_1, m, a'_1, u, \Theta) \times succ_{R2}(cf_2, ctxt, a_2, r, cf'_2, m, a'_2, u, \Theta)$
  where $\Theta \triangleq \quad \Phi \wedge share_R((cf_1, cf_2), ctxt, (a_1, a_2), r, (cf'_1, cf'_2), m, (a'_1, a'_2), u)$

When asked to share a formula, $\mho_1 \diamond \mho_2$ asks each of $\mho_1$ and $\mho_2$ for a formula, and conjoins the results.
When generating successors, $\mho_1$ and $\mho_2$ are given each others' shared formulae, as well as the incoming formula Φ, and asked to generate their own sets of successors. All pairs of these are then returned. We do not try to eliminate inconsistent pairs here, because this happens in a later iteration of the analysis anyway, when the pairs have their own successors generated.

Remark 3.21. If $\mho_1$ and $\mho_2$ satisfy the finiteness and soundness axioms, then so does their combination $\mho_1 \diamond \mho_2$.

Intuitively, it seems from Definition 3.20 that, when combining $n > 1$ plugins with $\diamond$, it doesn't matter in which order or which way round we combine them. We believe that it is possible to give an appropriate notion of equivalence between plugins, whence our combination operator $\diamond$ can be shown to be commutative, associative and idempotent, thereby making the space of plugins into a semilattice. Further, we suspect that the ordering in this semilattice coincides with a natural notion of refinement, where plugins which are higher in the refinement order produce tighter analysis results.

4. Description of Prototype Implementation

We have realised the framework just described as a prototype static assertion checker, which we report on in this section. The checker accepts programs written in a Java-like syntax and annotated with assertions, translates them to a control flow graph-based representation as in subsection 3.1, and analyses them with the algorithm of Figure 13. The user may give a constraint on the state at the beginning of execution, corresponding to the environment in which the code will be run.
The checker looks for assertion violations and memory errors (reading or writing a field using an address that is not allocated, or is allocated to an object of the wrong class). The outcome is either success, in which case the program is error-free, or an abstract counterexample trace. Such an abstract trace may correspond to a real error trace in the program, or may be infeasible.
By studying an infeasible trace, the user can try to work out where the abstraction is lacking, reconfigure the plugins accordingly and run the checker again. Improving the abstraction is called *abstraction refinement*, and the whole process is the *abstract-check-refine cycle* (e.g. [BR02, HJMS02]). Each plugin may be enabled or disabled, and enabled plugins are configured on a per-method basis. Plugins will guess configurations where they are not given, to serve as a reasonable starting point for the abstract-check-refine cycle.
Presently there is no support for automatic abstraction refinement (see future directions in subsection 6.2). However, to aid in the understanding of abstract traces, they can be output hierarchically in XML format. At the top level, the tree contains the sequence of abstract states making up the trace. Within each abstract state, one can see the formulae exchanged and the values of each of the plugins. At the lowest level, one can see exactly the execution of the underlying analyses, such as the invocations of the theorem prover for predicate abstraction. Thus the user may collapse irrelevant parts of the trace and focus on interesting features. Some screenshots are included in section 5 (Figures 19 and 20).
The prototype is implemented in Standard ML (which is very close to the language used by Isabelle,

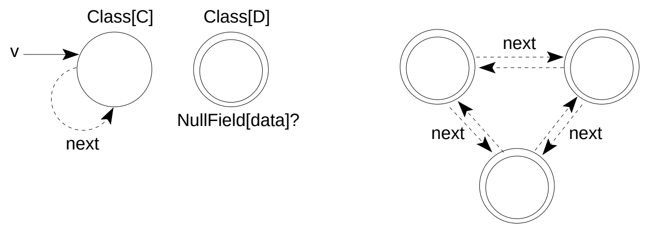


Fig. 15. Two abstract (3-valued) heaps from the TVLA plugin. The heap on the right abstracts exactly the three-colourable heaps of size ≥ 3, a class which cannot be defined in FO(TC) unless NL = NP. The question mark postfixed to a unary predicate indicates it has the unknown truth value, or $\frac{1}{2}$.

making the gap between the implementation and the formalisation small). Currently there are four plugins, three of them based on existing software:

1. predicate abstraction plugin, using an existing theorem prover
2. 3-valued shape analysis using TVLA
3. graph-type pointer analysis using PALE
4. a plugin based on a simple type system

We will spend the rest of this section describing the existing plugins and the analyses they provide, as well as how they are implemented.

4.1. Predicate abstraction plugin

Using the theorem prover Simplify [Nel80] this plugin provides predicate abstraction as set out in Example 3.16. The default configuration (i.e. set of abstraction predicates) for each method is taken to be the set of guards appearing on edges in that method (except *True*, and including only one of P and $\neg P$) and can be added to by manually reconfiguring the plugin. Transitive closure is provided for with a few simple first order axioms in the style of [LAIR$^+$05].

4.2. TVLA plugin

As explained in subsection 2.5, TVLA represents heaps as models of a three-valued logic with unary and binary predicates. Our models use the following "core" predicates:

- For each program variable v, the nullary predicate $\mathit{NullVar}[v]$ indicates that the variable v is null, and the unary predicate $\mathit{Var}[v](o)$ indicates that v points to the object o.
- For each field f, the unary predicate $\mathit{NullField}[f](o)$ indicates that the field f is null in object o, and the binary predicate $\mathit{Field}[f](o_1, o_2)$ indicates that the f field of object o_1 points to object o_2.
- For each class C, the unary predicate $\mathit{Class}_C(o)$ indicates that object o is an instance of C.

The $NullVar[v]$ predicates are necessary because a variable which does not point to some object is not automatically null - perhaps the variable is being used for arithmetic, for instance.

Figure 15 shows two examples. The 3-valued heap on the left represents any heap containing exactly one object of class C, pointed to by the variable v and whose next field may not point to any object other than itself, and one or more objects of class D, whose data fields may not point to any object. The question mark postfixed to a unary predicate indicates it has the unknown truth value, or $\frac{1}{2}$.

The abstract heap on the right in Figure 15 represents all three-colourable heaps of size ≥ 3, a class which cannot be defined in FO(TC) [Imm99] unless NL = NP. We include it to emphasize that although communication between plugins must be in FO(TC), the (private) abstract values which each plugin uses are unrestricted and may, as here, express constraints which cannot be translated exactly into FO(TC).

Configurations for the TVLA plugin include:

1. instrumentation predicates, which make the abstraction more precise by tracking additional properties (often reachability and heap-sharing properties) [SRW02],

2. sharing patterns (which we discuss shortly)

The update rules for handling program statements are much the same as in the standard TVLA examples, so we concentrate here on how shared information expressed in $\mathscr{L}$ can be translated for use by TVLA and vice versa. In fact, since TVLA's logic is also a first order logic with transitive closure, formulae of $\mathscr{L}$ are nearly in the right form already. The two difficulties are that:

1. The universes over which the logics are interpreted are different. In $\mathscr{L}$ variables range over $\mathbb{Z}$, whereas in TVLA's logic they range over (just) the allocated objects.
2. Fields are encoded with functions in $\mathscr{L}$ but with relations in TVLA.

However, for formulae in which variables are suitably "guarded" by the *Allocd* predicates the problem of differing universes goes away, so these formulae can be translated in a largely structural fashion. Let Φ be a first order formula built from the following grammar (a subset of $\mathscr{L}$):

$$
\begin{array}{rl}
e ::= & 0 \mid v \mid X \mid f(e) \\
\\
\Phi ::= & \mathrm{e} = \mathrm{e} \\
& \mid \mathit{True} \mid \neg\Phi \mid \Phi \wedge \Phi \\
& \mid \exists X\ (\mathit{Allocd}^c(X) \wedge \Phi) \\
& \mid TC_{[A,B]}\,[\mathit{Allocd}^c(A) \wedge \mathit{Allocd}^c(B) \wedge \Phi(A,B)]\,(e,e)
\end{array}
$$

Apart from the cases of equality and transitive closure, the translation is by structural recursion:

$$
\begin{array}{l}
(\neg\Phi)^\dagger \triangleq \neg(\Phi^\dagger) \\
(\Phi_1 \wedge \Phi_2)^\dagger \triangleq (\Phi_1^\dagger) \wedge (\Phi_2^\dagger) \\
(\exists X\ (\mathit{Allocd}^c(X) \wedge \Phi))^\dagger \triangleq \exists X\ \big(\mathit{Class}_c(X) \wedge \Phi^\dagger\big)
\end{array}
$$

Translating equalities is more involved due to both of the difficulties identified above. There are exactly three situations in which $e_1 = e_2$ holds:

- both e_1 and e_2 are null
- both e_1 and e_2 contain the address of the same allocated object
- neither e_1 nor e_2 is null or contains the address of an allocated object, but they are nevertheless equal

For each expression e we define two formulae: N_e which holds when e evaluates to null, and $R_e(o)$ which holds when e evaluates to the address o of an allocated object. In full:

$$
\begin{array}{ll}
N_0 := \mathit{True} & N_v := \mathit{NullVar}[v] \\
R_0(o) := \mathit{False} & R_v(o) := \mathit{Var}[v](o) \\
\\
N_X := \mathit{False} & N_{f(e)} := \exists o.\,(R_e\,(o) \wedge \mathit{NullField}[f]\,(o)) \\
R_X(o) := X = O & R_{f(e)}(o) := \exists o'.\,(R_e\,(o') \wedge \mathit{Field}[f]\,(o', o))
\end{array}
$$

(Recall that N_e and R_e are not exhaustive, corresponding to the third situation above: perhaps e refers to a variable used for arithmetic, or gets an arbitrary value, such as when performing a field read on null). We can now translate $e_1 = e_2$ soundly as:

$$
(e_1 = e_2)^\dagger \quad \triangleq \quad (N_{e_1} \wedge N_{e_2}) \vee \exists o\,(R_{e_1}\,(o) \wedge R_{e_2}\,(o)) \vee (\neg\,(N_{e_1} \vee N_{e_2} \vee \exists o R_{e_1}\,(o) \vee \exists o R_{e_2}\,(o)) \wedge\ \tfrac{1}{2})
$$

The presence of $\frac{1}{2}$ here is unavoidable: because our TVLA plugin only tracks integers which are 0 or point to some object, we cannot distinguish between integers for which this is not the case.

Example 4.1. Translation of $\neg(v = 0)$ from $\mathscr{L}$ to TVLA's logic.

$$
\begin{aligned}
& (\neg (v = 0))^{\dagger} \\
\triangleq\ & \neg (v = 0)^{\dagger} \\
\triangleq\ & \neg ((N_v \wedge N_0) \vee \exists o\, (R_v\,(o) \wedge R_0\,(o)) \vee (\neg\,(N_v \vee N_0 \vee \exists o R_v\,(o) \vee \exists o R_0\,(o)) \wedge\ \tfrac{1}{2}))
\end{aligned}
$$

Fill in the right values for N_v, N_0, R_v and R_0:

$$
\triangleq\ \neg\,((\mathit{NullVar}[v] \wedge \mathit{True}) \vee \exists o\,(\mathit{Var}[v]\,(o) \wedge \mathit{False}) \vee (\neg\,(\mathit{NullVar}[v] \vee \mathit{True} \vee \exists o\,\mathit{Var}[v]\,(o) \vee \exists o \mathit{False}) \wedge\ \tfrac{1}{2}))
$$

This trivially simplifies, using the relationships between $\wedge, \vee$ and *True*, *False*, to:

$$
\begin{aligned}
\triangleq\ & \neg\,(\mathit{NullVar}[v] \vee (\neg\,(\mathit{NullVar}[v] \vee \mathit{True} \vee \exists o\,\mathit{Var}[v]\,(o) \vee \exists o \mathit{False}) \wedge\ \tfrac{1}{2})) \\
\triangleq\ & \neg \mathit{NullVar}[v]
\end{aligned}
$$

It remains to say how to translate transitive closure.

$$
\begin{aligned}
& \left(TC_{[A,B]}\,[\mathit{Allocd}^c(A) \wedge \mathit{Allocd}^c(B) \wedge \Phi(A,B)]\,(e_1,e_2)\right)^{\dagger} \\
\triangleq\ & \exists x_1 x_2 . R_{e_1}(x_1) \wedge R_{e_2}(x_2) \wedge TC_{[A,B]}\left[\mathit{Class}_c(A) \wedge \mathit{Class}_c(B) \wedge \Phi(A,B)^{\dagger}\right](x_1,x_2)
\end{aligned}
$$

The *succ* operations make use of the translated formulae $\Phi^{\dagger}$ by adding them to the TVLA model as *consistency rules.* This causes TVLA to reject abstract heaps incompatible with the formula. Additionally, sometimes we can cause TVLA to sharpen a value from $\frac{1}{2}$ to a definite value (*True* or *False*) by first using $\Phi^{\dagger}$ as a *focus formula* (see [SRW02]); the focus operation effectively splits an abstract heap where $\Phi^{\dagger}$ has value $\frac{1}{2}$ into the cases where it is definitely true or definitely false.[1] Formulae which are not in the required form for translation, but include top-level conjuncts which are, will be partially translated.

For example, running the no-operation statement ε on the heap in the left of Figure 15 with $\mathit{next}(v) = \mathit{null}$ will cause the heap to be rejected, whereas instead using $\mathit{next}(v) = v$ will sharpen the dashed line for the next field to a solid one.

The translation given above can be reused to induce transfer of information in the other direction, when running *share*, share_C and share_R. Configurations for TVLA specify patterns of $\mathscr{L}$-formulae for potential sharing. For example, the default configurations contain the following sharing patterns:

- $V = \mathit{null}$,
- $V \neq \mathit{null}$,
- $U = V$, and
- $U \neq V$

where U, V stand for any variables in scope.

For each candidate Φ, we form the translation $\Phi^{\dagger}$ and check whether $\neg\Phi^{\dagger}$ is consistent with the TVLA model. If not, then Φ is "entailed" by the TVLA model, and becomes a conjunct in the formula exported by *share*. We have not yet explored a direct translation from TVLA to $\mathscr{L}$.

Even though the TVLA system doesn't "understand" an integer constraint such as $\mathit{data}(x) < \mathit{data}(y)$, it is in principle possible to draw inferences useful to TVLA from them: for instance $\mathit{data}(x) < \mathit{data}(y)$ tells us that x and y cannot be equal. We don't currently use such inferences, and we do not know whether a most precise translation exists from $\mathscr{L}$ to TVLA's logic.

4.3. PALE plugin

Recall from subsection 2.4 that the PALE tool analyses only *graph type* heaps (those based around a tree backbone), for which a declaration must be given, saying which fields form the backbone and how additional

[1] Focusing cannot be performed for every $\Phi^{\dagger}$; this is discussed in [SRW02].

fields must behave. Thus each configuration for the PALE plugin contains such a declaration (Figure 2 is an example). The other component of configurations is a set of sharing patterns; as for the TVLA plugin we identify a subset of $\mathscr{L}$ constraints that we can translate into PALE's logic, and also use that to induce a translation in the other direction.

The abstract values $T = \{\mathit{Good}, \mathit{All}\}$ for PALE simply record whether the graph type declaration given has been respected. The value *Good* concretises to the set of heap-environment pairs which meet the declaration, whereas *All* concretises to all heap-environment pairs (and so conveys nothing). A considerable limitation is that for the PALE plugin to do any analysis, the target program's entire heap must be a graph type. If only a part of the heap is of the right form, we would like to apply PALE to only this part, but currently we cannot.

4.4. Making a simple type system into a plugin

In the programming language we target, all variables contain integer values. However, we can broadly classify variables according to their use: some are used for integer data values (and have arithmetic operators applied to them) and some are used to store the addresses of objects (and have field reads and writes and method calls applied to them). If the programmer accidentally mixes these uses, it is likely that running the program will produce run-time errors (represented, for us, by a transition to the error location).

A standard way to prevent such errors is to type-check the code at compilation time. Here we present a simple type system which prevents mixing "data" integers with "address" integers. The only types are *Int*, used for data integers, and *ref*(c), used for addresses of objects of class c.

$$\mathit{Type} := \{\mathit{Int}\} \cup \{\mathit{ref}(c) \mid c \in \mathit{ClassName}\}$$

A *type assignment* Δ for the program P is a triple of the following components:

$\Delta_M : \mathit{MethodName} \to \mathit{Type}$ gives a type to the value returned by each method

$\Delta_F : \mathit{Fieldname} \to \mathit{Type}$ gives a type to each field

$\Delta_V : \mathit{VarName} \times \mathit{MethodName} \to \mathit{Type}$ gives a type to a given variable (local variable or formal parameter) of a given method

The judgement $\Delta, m \vdash s$ means that the statement s appearing in the method named m is correctly typed by Δ. Figure 16 gives rules for typing the various statement forms. We say that Δ is *valid* for P if for every statement s labeling some node in a method m of P, we have $\Delta, m \vdash s$.

Definition 4.2. A program state p of program P is said to *meet the type assignment* Δ if, informally:

- **Variables**: For every variable v in scope, if $\Delta(v) = \mathit{ref}(c)$ then either v is null, or v contains the address of an allocated object of class c.
- **Objects**: For every address $a \in \mathbb{Z}$, if a is the address of an allocated object o of class c, f is a field of c and $\Delta_F(f) = \mathit{ref}(c')$ then either the f-field of o is null, or contains the address of an allocated object of class c'.[2]

This is expressed precisely in $\mathscr{L}$ as:

$$\Psi_m^{\mathit{variables}} \triangleq \bigwedge_{v \in \Pi^M_{\mathit{Locals}}(m)} \quad \bigwedge_{\{c \in \mathit{ClassNames} \;\mid\; \Delta_V(m,v) = \mathit{ref}(c)\}} v = 0 \vee \mathit{Allocd}^c(v)$$

$$\Psi^{\mathit{objects}} \triangleq \bigwedge_{c,c' \in \mathit{ClassNames}} \forall X \bigwedge_{\{f \in \Pi^C_{\mathit{Fields}}(c) \;\mid\; \Delta_F(f) = \mathit{ref}(c')\}} \mathit{Allocd}^c(X) \to \Big(f(X) = 0 \vee \mathit{Allocd}^{c'}(f(X))\Big)$$

$$\Psi_m \triangleq \quad \Psi_m^{\mathit{variables}} \wedge \Psi^{\mathit{objects}}$$

[2] Strictly it is only necessary to constrain the fields of objects which are reachable from program variables. For simplicity we don't do this.

$$\frac{\Delta_V(m,u) = \Delta_V(m,v)}{\Delta, m \vdash u := v}$$

$$\frac{\Delta_V(m,u) = \mathit{ref}(c)}{\Delta, m \vdash u := 0}$$

$$\frac{\Delta_V(m,u) = \mathit{Int}}{\Delta, m \vdash u := n}$$

$$\frac{\Delta_V(m,u) = \Delta_V(m,v_1) = \Delta_V(m,v_2) = \mathit{Int}}{\Delta, m \vdash u := v_1 \otimes v_2}$$

$$\frac{\begin{array}{c}\Delta_V(m,u) = \Delta_F(f)\\ \Delta_V(m,v) = \mathit{ref}(c)\end{array}}{\Delta \vdash u := v.f}\, f \in \Pi^C_{\mathit{Fields}}(c)$$

$$\frac{\begin{array}{c}\Delta_F(f) = \Delta_V(m,v)\\ \Delta_V(m,u) = \mathit{ref}(c)\end{array}}{\Delta \vdash u.f := v}\, f \in \Pi^C_{\mathit{Fields}}(c)$$

$$\frac{\Delta_V(m,u) = \mathit{ref}(c)}{\Delta, m \vdash u := \text{new } c}$$

$$\frac{\begin{array}{c}\Delta_V(m,u) = \Delta_M(m')\\ \Delta_V(m,v) = \Delta_V(m',\mathit{this}) = \mathit{ref}(c)\\ \text{for } i = 1,\ldots,n,\ \Delta_V(m,p_i) = \Delta_V(m',a_i)\end{array}}{\Delta, m \vdash u := v.m'(p_1,\ldots,p_n)}\, m' \in \Pi^C_{\mathit{Methods}}(c) \text{ and } [a_1,\ldots,a_n] = \Pi^M_{\mathit{Formals}}(m')$$

$$\frac{\Delta_V(m,v) = \Delta_M(m)}{\Delta, m \vdash \text{return } v}$$

$$\frac{}{\Delta, m \vdash \varepsilon}$$

Fig. 16. Typing rules for a simple type system (some with side conditions). The judgment $\Delta, m \vdash s$ means that the statement s appearing in the method named m is correctly typed by the type assignment Δ.

The following theorem shows that for well-typed programs, certain run-time errors can never occur, namely those where a variable is expected to contain the address of an object of some class C_1 but actually points to one of a different class C_2. Therefore, checking for these errors at runtime is unnecessary.

Theorem 4.3. Behaviour of correctly-typed programs: Let Δ be a valid type assignment for a program P. Let state p_0 meet Δ and let (p_0, p) be a trace in P. Then p meets Δ.

We will now turn this type system into a plugin. We take our configurations to be the type assignments (insisting that every method of the program receives the same configuration), and our abstract states to be just *Yes*(Δ), which says that the type assignment Δ is valid and is in force, and *No* which says that no type assignment is in force. Consequently, the concretisation function γ is defined in terms of the appropriate Ψ_m.

The *init* operation performs two checks. Firstly, it type-checks the entire program against the given Δ,

$$\gamma(\mathit{ctxt}, \mathit{No}) := [\![\mathit{True}]\!]$$
$$\gamma(\mathit{ctxt}, \mathit{Yes}(\Delta)) := [\![\Psi_{\mathit{currentMethod}(\mathit{ctxt})}]\!]$$

$$\mathit{init}(\Delta, m_0, \Phi) := \begin{cases} \mathit{Yes}(\Delta) & \text{if } \mathit{TypeCheck}(P, \Delta) \text{ and } \mathit{TheoremProver}(\Phi \rightarrow \Psi_{m_0}) \\ \mathit{No} & \text{otherwise.} \end{cases}$$

$$\mathit{share}(\Delta, \mathit{ctxt}, a, s) := \begin{cases} v = 0 \vee \mathit{Allocd}^c(v) & \text{if } a = \mathit{Yes}(\Delta) \text{ and } s \text{ is } u := v.f \text{ or } v.f := u \\ \mathit{True} & \text{otherwise.} \end{cases}$$

$$\mathit{succ}(\Delta, \mathit{ctxt}, a, s, \Phi) := \{a\}$$

$$\mathit{share}_C(...) := \mathit{True}$$
$$\mathit{succ}_C(\Delta, \mathit{ctxt}, a, ...) := \{a\}$$

$$\mathit{share}_R(...) := \mathit{True}$$
$$\mathit{succ}_R(\Delta, \mathit{ctxt}, a, ...) := \{a\}$$

$$\chi(\Delta) := \{\mathit{Yes}(\Delta), \mathit{No}\}$$

Fig. 17. Definition of a plugin based on a simple type system.

using the rules from Figure 16[3]. If the program type-checks, then by Theorem 4.3, executing the program preserves the property of meeting Δ. However, it remains to be checked that the initial state meets Δ, so *init* invokes the theorem prover Simplify to see whether the appropriate Ψ follows from the initial constraint Φ_0. If both checks pass, then *init* returns $\mathit{Yes}(\Delta)$, otherwise *No*.

Functions *succ*, succ_C and succ_R do not have any work to do, because the whole program has been type-checked right at the beginning; they merely return the same value they are given. The purpose of the plugin lies in *share*. Suppose we are in $\mathit{Yes}(\Delta)$ mode and have reached a (potentially dangerous) field read $u := v.f$. Through *share* the plugin contributes the formula $v = 0 \vee \mathit{Allocd}^c(v)$. As long as another plugin can produce the fact that the variable is non-null, the analysis will be content that no memory error is possible. The detailed example in section 5 will show this in action.

We conclude this section with a few remarks.

- This particular type system doesn't provide anything that can't be obtained by predicate abstraction, using the appropriate Ψs above as abstraction predicates. But the type system plugin does work more efficiently: it performs a cheap static check once, rather than causing the theorem prover to do extra work at every iteration.
- If a program doesn't type-check, it *doesn't mean that the program can reach an error state* - perhaps the program is fine, but a more dynamic (expensive) analysis such as predicate abstraction is required to establish this. For example, using predicate abstraction with the Ψ formulae above allows us to temporarily break and reestablish the invariant e.g. by $x := x + 1; x := x - 1$ on an "address" variable x. We think it makes sense to have both approaches available and cooperating. (Similar comments apply to constant propagation, which can also be encoded using predicate abstraction.)
- Finally, the above type system is clearly simple and the information it contributes to the analysis is weak. We would like to work on integrating more interesting type systems, such as the non-null types from the Cyclone language [JMG+02], and ownership type systems (e.g. [DM05]) which we return to in section 6.2.

5. Detailed example

In this section we trace what happens when we run our analysis tool on the program P in Figure 18. Suppose we use just the predicate abstraction, TVLA and type system plugins, and as our initial constraint we have

[3] Because *init* has access to the whole program P, what we are constructing here is strictly not a plugin but a family of plugins parametrised by P.

```
    class C
    {
      field data;

      static method main()
        {
          vars x, y;

0         x := new C;
1         y := new C;

2         x.data := 10;
3         y.data := 11;

4         assert x.data < y.data;
        }
    }
```

Fig. 18. A small program used to demonstrate the benefit of exchanging formulae between plugins.

$$\Phi_0 \triangleq \quad x = \mathit{null} \wedge y = \mathit{null} \wedge \forall X.(\mathit{Allocd}^C(X) \rightarrow \mathit{False})$$

which says that x and y are null, and the heap is empty. We configure the plugins as follows:

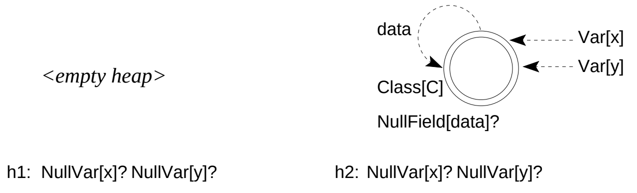
- Predicate abstraction: we choose abstraction predicates $cf_{PA} := \{P_1 \triangleq \mathit{data}(x) = 10, P_2 \triangleq \mathit{data}(x) < \mathit{data}(y)\}$
- TVLA: we take no instrumentation predicates, and choose our sharing patterns to be the defaults, i.e. $V = \mathit{null}$, $U = V$ and their negations.
- Type system: we assign the variables x and y the type $\mathit{ref}(C)$, and give the type Int to the field data. We will call this type assignment Δ.

When the analysis begins, the first thing that happens is that *init* is invoked on each plugin. For predicate abstraction, we have

$$\begin{aligned} & \mathit{init}(cf_{PA}, \mathit{ctxt}, \Phi_0) \\ \triangleq\ & \{P_1 \wedge \neg P_2, \neg P_1 \wedge \neg P_2\} \\ \triangleq\ & \{\mathit{data}(x) = 10 \wedge \neg \mathit{data}(x) < \mathit{data}(y), \neg \mathit{data}(x) = 10 \wedge \neg \mathit{data}(x) < \mathit{data}(y)\} \end{aligned}$$

The expression $\mathit{data}(x)$ is "undefined" and gets an arbitrary value (as per Figure 6), and so both P_1 and $\neg P_1$ are consistent. However, we cannot have P_2: because Φ_0 says that both x and y are null, $P_2 \wedge \Phi_0$ entails $\mathit{data}(\mathit{null}) < \mathit{data}(\mathit{null})$ and is therefore unsatisfiable [4] (and the theorem prover is able to show this).

Now we consider TVLA. Internally, the TVLA plugin generates the two initial heaps h_1 and h_2:

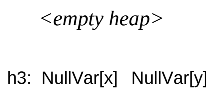


[4] In $\mathscr{L}$ we may state that $\mathit{data}(\mathit{null}) = \mathit{data}(\mathit{null})$ even though $\mathit{data}(\mathit{null})$ is "undefined" and thus interpreted as an arbitrary value (see Figure 6), because whatever that arbitrary value is, it is the same on both sides. While one may certainly have a debate about the best way to handle partial operations, that is not the purpose of this paper.

which concretise to the empty heap and all non-empty heaps respectively. The plugin translates Φ_0 into TVLA's logic as:

$$\mathit{NullVar[x]} \wedge \mathit{NullVar[y]} \wedge \forall X(\mathit{class[C]}(X) \rightarrow \mathit{False})$$

This translated formula causes the right hand heap h_2 to be eliminated, and the left hand heap h_1 is sharpened to



The type system is able to type-check the program and, since Φ_0 tells us that x and y are initially null, is able to decude that these variables start out with values of the required type. So we have $\mathit{init}(\Delta, \mathit{ctxt}, \Phi_0) = \{\mathit{Yes}(\Delta)\}$. Therefore, the initial worklist is:

$$\{(\mathit{main}, 0, (P_1 \wedge \neg P_2, h_3, \mathit{Yes}(\Delta))\,(P_1 \wedge \neg P_2, h_3, \mathit{Yes}(\Delta)))\,, (\mathit{main}, 0, (\neg P_1 \wedge \neg P_2, h_3, \mathit{Yes}(\Delta))\,(\neg P_1 \wedge \neg P_2, h_3, \mathit{Yes}(\Delta)))\}$$

Suppose that, at the first iteration of the algorithm, the analyser chooses the first of these, $(\mathit{main}, 0, P_1 \wedge \neg P_2, P_1 \wedge \neg P_2)$, to remove from the worklist and process. The first step for the analyser is to request a shared formula from each plugin. The predicate abstraction plugin simply shares its entire monomial:

$$\begin{aligned} & \mathit{share}(\mathit{cf}_{PA}, \mathit{ctxt}, P_1 \wedge \neg P_2, x := \text{ new } C) \\ \triangleq\ & P_1 \wedge \neg P_2 \\ \triangleq\ & \mathit{data}(x) = 10 \wedge \neg \mathit{data}(x) < \mathit{data}(y) \end{aligned}$$

The TVLA plugin expands the sharing patterns in its configuration, $V = \mathit{null}$, $U = V$ and their negations, into candidates for sharing. For example, $V \neq \mathit{null}$ expands to $x \neq \mathit{null}$ and $y \neq \mathit{null}$. These are translated and tested against the heap, resulting in the shared formula

$$\mathit{share}(\mathit{cf}_{TVLA}, \mathit{ctxt}, h_3, x := \text{ new } C) \quad \triangleq \quad x = \mathit{null} \wedge y = \mathit{null} \wedge x = y$$

Because the statement being executed is not a potentially dangerous memory access, the type system plugin doesn't contribute anything:

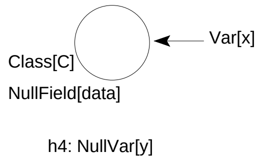
$$\mathit{share}(\Delta, \mathit{ctxt}, \mathit{Yes}(\Delta), x := \text{ new } C) \quad \triangleq \quad \mathit{True}$$

Overall the shared information is

$$\mathit{data}(x) = 10 \wedge \neg \mathit{data}(x) < \mathit{data}(y) \wedge x = \mathit{null} \wedge y = \mathit{null} \wedge x = y \wedge \mathit{True}$$

In this case, the shared formulae do not bring about any improvement. The abstract successors added to the worklist are

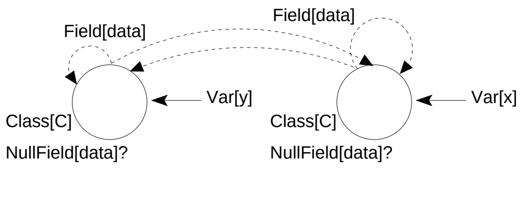
$$\begin{aligned} \{\, & (\mathit{main}, 1, (P_1 \wedge \neg P_2, h_3, \mathit{Yes}(\Delta))\,(\neg P_1 \wedge P_2, h_4, \mathit{Yes}(\Delta)))\,, \\ & (\mathit{main}, 1, (P_1 \wedge \neg P_2, h_3, \mathit{Yes}(\Delta))\,(\neg P_1 \wedge \neg P_2, h_4, \mathit{Yes}(\Delta)))\} \end{aligned}$$



Observe that this time P_1 isn't generated, only $\neg P_1$, because new objects have their fields initialised to zero.

Now let us skip forward to a more interesting iteration, where sharing makes a crucial difference. Eventually the analysis will arrive at the following abstract state, about to execute $y.data := 11$:

$$(main, 3, (P_1 \wedge \neg P_2, h_3, Yes(\Delta))\,(P_1 \wedge \neg P_2, h_5, Yes(\Delta)))$$



h5:

In order to check for a potential memory error when updating the field *data* at y, the analyser invokes *succ* in two distinct "branches": one with the formula $\neg Allocd^C(y)$ and targeting the error node -1, representing a memory error, and one with $Allocd^C(y)$ and targeting the next node 4, representing safe execution.

The branch representing an error will be closed as follows. The type system will contribute $y = null \vee Allocd^C(y)$ and TVLA will contribute $y \neq null$. This is not consistent with $\neg Allocd^C(y)$, so no successor states will be generated.

Now, in the other (ordinary) branch, in the next state we will have to validate the assertion $data(x) < data(y)$ (i.e. P_2). TVLA cannot do this alone because it "doesn't understand" integers. On the other hand, predicate abstraction alone will produce some states with $\neg P_2$ because it "doesn't know" that we don't have $x = y$, and therefore "reasons" that the update to $y.data$ may update $x.data$ as well. But the combination of the two of them works: TVLA shares $x \neq y$ and this allows predicate abstraction to infer that $x.data$ is unaffected, and so the assertion is established.

The screenshots in Figures 19 and 20 show the counterexample trace produced by our tool if we don't use the type system plugin; the analysis cannot determine that the memory access $x.data = 10$ is safe, and so produces a successor at the error state.

At the top level, the trace features three repeating components:

1. **Constraints:** These are the abstract states along the execution. When expanded, this shows the abstract state within each plugin being used; these per-plugin components can themselves be expanded to show more detail.
2. **Statements or conditions:** These show the program statements being executed (abstractly), or in the case of assertion and control structures, the condition being analysed.
3. **Product successor computations:** When expanded, these show the details of how the analyser gets from one abstract state to the next.

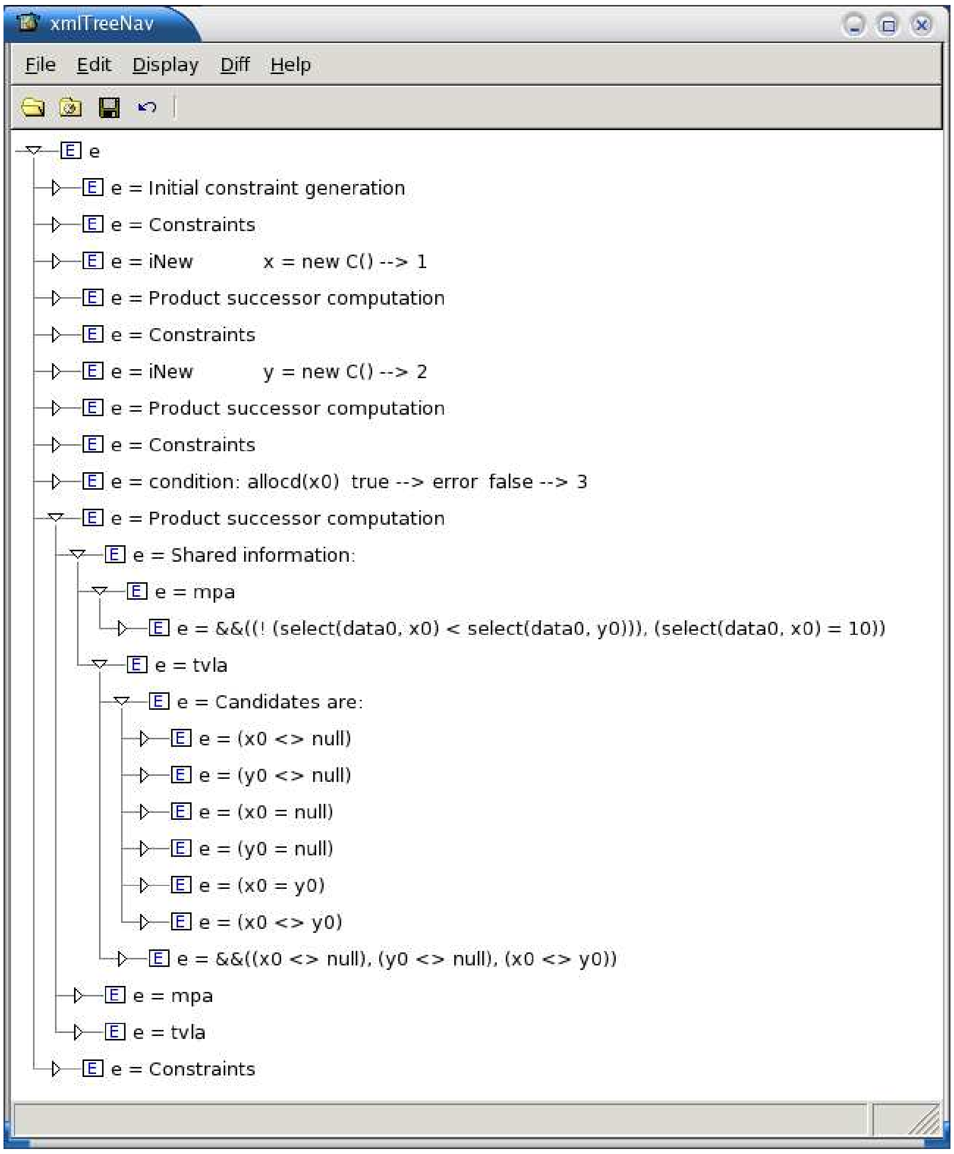
In Figure 19, we have expanded a successor computation, to show the information being shared by each plugin ("mpa" stands for monomial predicate abstraction). We can see predicate abstraction sharing an entire monomial, and the TVLA plugin generating and testing candidate formulae, three of which end up being shared.

Figure 20 shows the predicate abstraction plugin generating and testing successor states, i.e. monomials. By expanding one of the prospective successor monomials we can see the corresponding invocation of the theorem prover Simplify.

6. Discussion and future work

6.1. Choice of intermediate language

The first design issue we considered was whether to have a single intermediate language that all plugins use, or allow each pair of plugins to have their own private communication mechanism. The latter should allow more precise and concise exchange of information, particularly between plugins whose abstract values are already "close together" in expressiveness, but "distant" from the proposed single intermediate language. However, there is a serious disadvantage in terms of implementation effort: the implementor of a new analysis



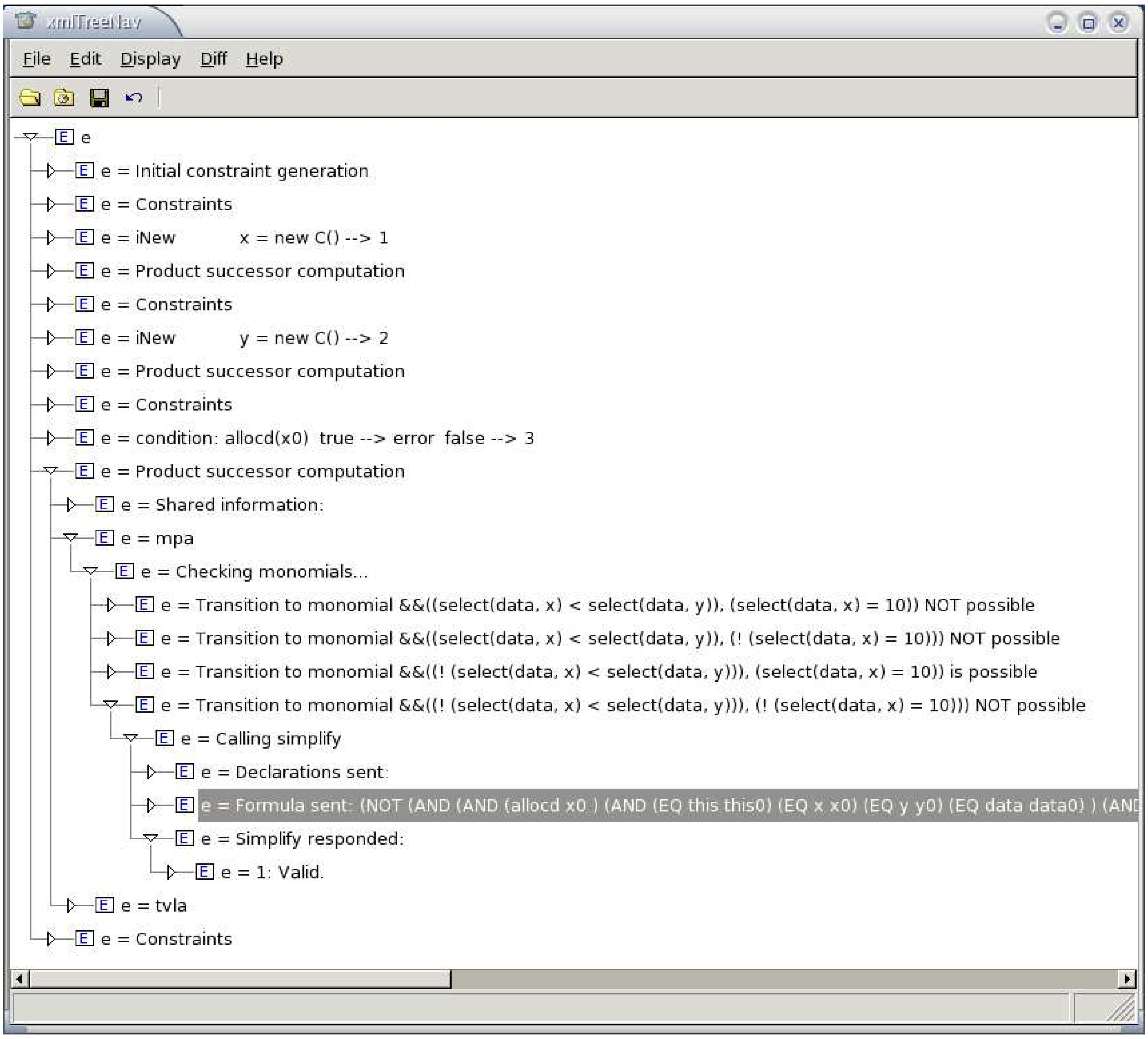
```
e
  e = Initial constraint generation
  e = Constraints
  e = iNew        x = new C() --> 1
  e = Product successor computation
  e = Constraints
  e = iNew        y = new C() --> 2
  e = Product successor computation
  e = Constraints
  e = condition: allocd(x0)  true --> error  false --> 3
  e = Product successor computation
    e = Shared information:
      e = mpa
        e = &&((! (select(data0, x0) < select(data0, y0))), (select(data0, x0) = 10))
      e = tvla
        e = Candidates are:
          e = (x0 <> null)
          e = (y0 <> null)
          e = (x0 = null)
          e = (y0 = null)
          e = (x0 = y0)
          e = (x0 <> y0)
        e = &&((x0 <> null), (y0 <> null), (x0 <> y0))
    e = mpa
    e = tvla
  e = Constraints
```

Fig. 19. A screenshot of our tool displaying an abstract error trace. Here we have expanded a successor computation to show the information being shared by each plugin. We can see predicate abstraction sharing an entire monomial, and the TVLA plugin generating and testing candidate formulae.

may be faced with a large number of languages - one for every other plugin he wishes his to work with - and this undermines exactly the modularity we are striving to achieve. For us this is decisive.

Having settled on using one intermediate language rather than many, we must of course choose which language. Our choice of FO(TC) is essentially an educated guess; we offer the following arguments in support of it:

1. **Some notion of reachability is clearly necessary**. It has been observed (e.g. [IRR$^+$04, BCO04]) that the ability to express the reachability of data via particular variables and fields is essential for



```
e
  e = Initial constraint generation
  e = Constraints
  e = iNew          x = new C() --> 1
  e = Product successor computation
  e = Constraints
  e = iNew          y = new C() --> 2
  e = Product successor computation
  e = Constraints
  e = condition: allocd(x0)  true --> error  false --> 3
  e = Product successor computation
    e = Shared information:
    e = mpa
      e = Checking monomials...
        e = Transition to monomial &&((select(data, x) < select(data, y)), (select(data, x) = 10)) NOT possible
        e = Transition to monomial &&((select(data, x) < select(data, y)), (! (select(data, x) = 10))) NOT possible
        e = Transition to monomial &&((! (select(data, x) < select(data, y))), (select(data, x) = 10)) is possible
        e = Transition to monomial &&((! (select(data, x) < select(data, y))), (! (select(data, x) = 10))) NOT possible
          e = Calling simplify
            e = Declarations sent:
            e = Formula sent: (NOT (AND (AND (allocd x0 ) (AND (EQ this this0) (EQ x x0) (EQ y y0) (EQ data data0) ) (AN
            e = Simplify responded:
              e = 1: Valid.
    e = tvla
  e = Constraints
```

Fig. 20. A screenshot of our tool displaying an abstract error trace. We have focused on the predicate abstraction plugin generating and testing successor states, i.e. monomials. By expanding one of the prospective successor monomials we can see the corresponding invocation of the theorem prover Simplify.

analysing and verifying programs using dynamic data structures[5]. FO(TC) includes a very general notion of reachability.

2. **Sound FO(TC) reasoning can be done with existing first order provers**. By encoding transitive closure subformulae using first order predicates, sound (but necessarily incomplete) reasoning about FO(TC) formulae can be done; this allows us to use the wide variety of existing first order provers. This approach has received attention recently [LAIR+05], but dates back to [Nel83].
3. **FO(TC) can express some structural/spatial reasoning**. In [BCO04], decidable fragments of separation logic are given for reasoning about lists and trees. Given the nature of separation logic one might

[5] In this light, however, [MN05] is interesting because it shows what can still be said about data structures without using reachability.

expect these to be "intrinsically second order", but in fact the decidable fragments from [BCO04] (and more) can be translated into a decidable fragment of FO(TC) (this fact is briefly mentioned in [YRS+06]).

4. **Ownership appears related to transitive closure.** Ownership has been proposed as a crucial concept in making the verification of object-oriented programs scalable. Suppose O_1 and O_2 are objects at the top level of a program, and that O_2 owns an object P. The *owner-as-dominator* formulation of ownership (e.g. [Wre03]) says that P must be unreachable from O_1, except along paths which go via O_2. This means that O_1 cannot access P without the approval of P's owner O_2.

In any case, note that our notion of analysis plugin, and our analysis algorithm, use minimal properties of $\mathscr{L}$: that it can express truth, conjunction and the allocatedness of (the address in) a program variable. Thus the intermediate language can be changed very freely.

What other languages might we use? Second order logic, or even full higher order logic, spring to mind. These would certainly allow greater expressiveness, and would (in principle) ease the problem of getting information from the plugins into the intermediate language; however transfer in the other direction would become correspondingly more difficult. There appears to be little known about automated theorem proving in second and higher order logics. Technically, sound reasoning about higher order logics can be encoded in FO (see e.g. [NR03]) so an analogue of point 2. above applies, but the encodings appear too awkward to use, whereas the encodings of FO(TC) are quite clean.

In terms of the particular plugins we have so far, we speculate that second order logic might be a better fit with the PALE tool, but would fit less well with TVLA. With respect to point 3. above, it would be interesting to see if there is a similar connection between FO(TC) and the heap description formalisms based on graph grammars.

6.2. Future directions

Efficiency and sharing: Our prototype currently suffers from being rather slow. We suspect this is caused mainly by plugins sharing formulae when it isn't necessary: running the analyses together but exchanging no formulae is no slower than running them separately, and may in fact be faster because execution paths ruled out by one of the tools need not be considered by the others.

Exchanging information is necessarily more work, but we hope to minimise the additional cost by making sharing smarter, so that formulae are only propagated to where they are useful. Of course, working out when sharing is beneficial and when it is unnecessary is a daunting problem. Perhaps we can use an approach in the spirit of *lazy abstraction* [HJMS02], which adds abstraction predicates just where they are necessary. We certainly hope to do better than the current coarse scheme (recall that the predicate abstraction plugin shares entire monomials, while the TVLA and PALE plugins are given simple templates for properties to share).

Counterexample-guided abstraction refinement: Our current prototype contains no support for the CEGAR (counterexample-guided abstraction refinement) paradigm (e.g. [BR02, HJMS02]), in which an infeasible abstract trace generated by a failed verification is used to refine the abstraction. If verification fails, the user must manually reconfigure the plugins. Running several analyses together creates a new issue when automating abstraction refinement: How do we decide which plugin to reconfigure? E.g. we may have the choice between adding a new predicate to the predicate abstraction plugin, or adding a new instrumentation predicate to the TVLA plugin. A similar issue is encountered in work on automatically proving termination [CPR05], where one has to choose whether to refine the choice of abstraction predicates, or refine the set of relations used to show termination.

Ownership: We pointed out earlier that ownership is related to shape properties of the heap. Therefore we are interested in using the same trick as in subsection 4.4 with ownership type systems. The one we have in mind is Universe types ([DM05]). In the Universe type system not every desired ownership property can be inferred statically, so the programmer sometimes has to override the type system by writing a type cast. Thus we plan to exploit a two-way exchange with shape analyses, using information from shape analysis to establish that the programmer's casts are legitimate, and using ownership information to enhance shape analysis.

Acknowledgements

Thanks to Dennis Dams for drawing our attention to the work in [CLCVH00], and Michael Huth for many productive discussions about the contents of this paper.

References

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BCO04] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16–18, 2004, Proceedings*, volume 3328 of *LNCS*, pages 97–109. Springer, December 2004.

[BG03] Doron Bustan and Orna Grumberg. Simulation-based minimization. *ACM Trans. Comput. Logic*, 4(2):181–206, 2003.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of SPIN 2000*, volume 1885 of *LNCS*, pages 113–130. Springer Verlag, 2000.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, volume 2057 of *LNCS*, pages 103–122. Springer Verlag, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report, Microsoft, 2002. MSR-TR-2002-09.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.

[CC04] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In *SAS*, pages 312–327, 2004.

[Cha06] Nathaniel Charlton. Verification of java programs with interacting analysis plugins. *Electronic Notes in Theoretical Computer Science*, 145, Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVoCS 2005):131–150, 2006.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, volume 3385 of *LNCS*, pages 147–163. Springer Verlag, 2005.

[CLCVH00] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, August 2000.

[CMB$^+$95] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria Garcia de la Banda, and Manuel Hermenegildo. Improving abstract interpretations by combining domains. *ACM Trans. Program. Lang. Syst.*, 17(1):28–44, 1995.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction-refinement for termination. In Chris Hankin and Igor Siveroni, editors, *Static analysis : 12th International Symposium, SAS 2005*, volume 3672 of *Lecture Notes in Computer Science*, page 15, London, UK, September 2005. Springer.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, volume 1633 of *LNCS*, pages 160–171, 1999.

[DM05] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[FM97] Pascal Fradet and Daniel Le Métayer. Shape types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997. ACM Press.

[GW99] E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *Proceedings of 14th Annual IEEE Symposium on Logic in Computer Science, Trento*, pages 45–54, 1999.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages pp. 58–70. ACM Press, 2002.

[HS96] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 662–681. Springer-Verlag, 1996.

[Hub03] Engelbert Hubbers. Integrating tools for automatic program verification. In *Ershov Memorial Conference*, pages 214–221, 2003.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[Imm99] Neil Immerman. *Descriptive Complexity.* Springer-Verlage, New York, NY, 1999.

[IRR$^+$04] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL'04*, volume 3210 of *LNCS*, pages 160–174. Springer Verlag, 2004.

[JMG$^+$02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference.* USENIX, June 2002.

[KLZR05] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Implications of a data structure consistency checking system. In *International Conference on Verifed Software: Tools, Techniques, Experiments*, 2005.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[KNR05] Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In *CADE*, pages 260–277, 2005.

[LAIR+05] Tal Lev-Ami, Neil Immerman, Tom Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE 2005*, volume 3632 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[LAMS04] Tal Lev-Ami, Roman Manevich, and Shmuel Sagiv. TVLA: A system for generating abstract interpreters. In *IFIP Congress Topical Sessions*, pages 367–376, 2004.

[Mic04] Static driver verifier: Finding bugs in device drivers at compile-time. Technical report, Microsoft, April 2004.

[MN05] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.

[MS01] Anders Moller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 221–231, New York, NY, USA, 2001. ACM Press.

[NEFE03] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 12(1):28–63, 2003.

[Nel80] Greg Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, 1980.

[Nel83] Greg Nelson. Verifying reachability invariants of linked structures. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–47, New York, NY, USA, 1983. ACM Press.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NR03] Karim Nour and Christophe Raffalli. Simple proof of the completeness theorem for second-order classical and intuitionistic logic by reduction to first-order mono-sorted logic. *Theoretical Computer Science*, 308(1-3):227–237, 2003.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[SYY03] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.

[Wre03] Alisdair Wren. Inferring ownership. Master's thesis, Imperial College, London, June 2003. MEng4 Thesis.

[YRS+06] Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2006)*, 2006. To appear.