A smooth combination of linear and Herbrand equalities for polynomial time must-alias analysis

Helmut Seidl[1], Vesal Vojdani[1]*, and Varmo Vene[2]*

[1] Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{seidl,vojdanig}@in.tum.de
[2] Deptartment of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
varmo@cs.ut.ee

Abstract. We present a new domain for analyzing *must*-equalities between address expressions. The domain is a smooth combination of Herbrand and affine equalities which enables us to describe field accesses and array indexing. While the full combination of uninterpreted functions with affine arithmetics results in intractable assertion checking algorithms, our restricted domain allows us to construct an analysis of address *must*-equalities that runs in polynomial time. We indicate how this analysis can be applied to infer access patterns in programs manipulating arrays and structs.

1 Introduction

Consistent correlations between memory locations used by a program lies at the heart of many safety properties. In order to verify absence of data races in multi-threaded programs, accesses to memory locations need to be correlated with locks that guard them. In a language with pointer variables, correlating address expressions requires knowing when two expressions *must* alias, i.e., evaluate to the same memory location. In general, techniques for verifying the correct use of interface methods (e.g., [1]) can be refined with must-alias information to check that calls in a syntactically correct sequence consistently refer to the right data elements: a sequence such as $\mathsf{open}(e_1)$; ...; $\mathsf{close}(e_2)$; e.g., should access the same file handle when referring to the address expressions e_1 and e_2.

More recently, program-specific correlations have been studied: the length of a list is, perhaps, maintained in a separate variable which is thus semantically correlated. Lu et al. [11] apply statistical techniques to detect plausible multi-variable correlations of this kind. Their methods, although successful in detecting real bugs, are flow-insensitive and essentially syntactic; hence not ideal for formal verification. As the precise control flow as well as equalities between variables in the program are ignored, syntactically similar expressions may not represent the same *semantic* correlation, while syntactically different expressions could very

* Partially supported by the Estonian Science Foundation under grant no. 6713.

well be correlated. In order to enable sound inference of semantic correlations between addresses, we propose a novel analysis of *must-equalities.*

Our analysis is able to interprocedurally relate address expressions which use array indexing and field selection in structs. An access to a nested struct consists in the base address of the data element followed by sequences of selectors, such as $A\,.\,person\,.\,name$. Two such expressions are definitely equivalent if they are *textually identical.* This corresponds to the *Herbrand* interpretation of the binary operator "." and the selector labels. In order to deal with arrays as well, we enhance this base domain by affine expressions for indexed accesses. Two index expressions are equivalent iff they are equivalent w.r.t. the *arithmetic interpretation.* We show that the resulting combination of theories allows to infer *all* valid address equalities in polynomial time.

This is in stark contrast to previous work on assertion checking over the domains of uninterpreted functions and linear arithmetic. Detecting affine equalities in programs was pioneered by Karr [9]. This algorithm was extended to the inter-procedural case by Müller-Olm and Seidl [13]. A long line of research has provided methods for intra-procedurally detecting Herbrand equalities precisely [3, 10, 12, 22] — while the inter-procedural case still remains unsolved. A precise analysis algorithm is known for *functions* without side effects [16] and for arbitrary procedures if only unary operator symbols are considered [6].

When it comes to combining affine and Herbrand equalities, the basic approach is inspired by methods of combining decision procedures [18]. However, Gulwani and Tiwari [4] have shown that assertion checking over the full combined domain is coNP-hard. Hence, they subsequently present a highly expressive domain that allows sound analysis of pointer arithmetic and recursive data-structures in the style of Deutsch [2], but their algorithm is no longer complete w.r.t. their chosen abstraction [5]. Our domain construction, based on a sufficiently restricted sub-class of Herbrand terms carefully enhanced with fragments of linear arithmetic, enables sound and complete analysis in polynomial time.

2 The Programming Model

One key abstraction on which our method relies is that we only track the values of **int** variables and pointers. Thus, we ignore the values stored in arrays or structs. To simplify our setting, we make the additional assumption that the tracked variables themselves are never accessed indirectly through pointers; a common coding practice when developing safety-critical code [8]. Programs to be analyzed are modeled by systems of flow graphs as in Figure 1.

Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ denote the set of **int**-variables and $\mathbf{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_m\}$ the set of pointer variables used by the program. For the moment, we assume all variables to be global, but we will present methods for local variables in Section 7. In addition, we assume that we are given a set of names $\mathcal{C}$ denoting the global static data-structures of the program. Each of these data-structures is built up by forming structs and arrays from a set of base types, such as **int**, **float** or **mutex**. In the presence of dynamic memory allocation, we infer must-equality

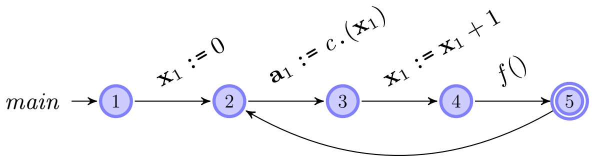
main
1
2
3
4
5
$\mathbf{x}_1 := 0$
$\mathbf{a}_1 := c.(\mathbf{x}_1)$
$\mathbf{x}_1 := \mathbf{x}_1 + 1$
$f()$

Fig. 1. Example flow-graph for a *main*-function.

relationships between pointer variables while also relying on may-alias pointer analysis, as further explained in Section 8; until then, we only deal with static data structures.

As we are only interested in assignments to integer and pointer variables, the set of statements Stmt at edges of programs in our model consists of:

- Affine assignments of the form $\mathbf{x}_j := t_0 + \sum_{i=1}^{k} t_i \mathbf{x}_i$ (with $t_i \in \mathbb{Z}$ and $\mathbf{x}_i \in \mathbf{X}$).
- Address assignments of the form $\mathbf{a}_j :=$ *adr* where *adr* is an address expression possibly involving variables from $\mathbf{X}$ and $\mathbf{A}$ in a way we will specify below.
- Non-deterministic assignments, $\mathbf{x}_j :=$? and $\mathbf{a}_j :=$?, which are used to abstract assignments that our analysis cannot handle.

With $\mathcal{C}$ denoting the set of global variable names, an address expressions *adr* is constructed from constants $B \in \mathcal{C}$ and address variables $\mathbf{a}_i$ according to the grammar:

$$adr \ ::= \ B \mid \mathbf{a}_i \mid adr\,.\,b \mid adr\,.\,(l)$$

where b is a field selector and l is an index expression of the form $l \equiv t_0 + t_1\mathbf{x}_1 + \ldots + t_1\mathbf{x}_k$. We assume that address expressions are *well-typed*. In particular, a selector b can only be applied to an address expression denoting a pointer to a struct with component b; likewise, only a pointer to an array can be indexed.

A *program* comprises a finite set Proc of *procedure names*. Execution starts with a call to the distinguished procedure $main \in$ Proc. Each procedure $q \in$ Proc is given through a *control flow graph* $G_q = (N_q, E_q, e_q, r_q)$ which consists of a set N_q of *program points*; a set of edges $E_q \subseteq N_q \times (\mathsf{Stmt} \cup \mathsf{Proc}) \times N_q$ annotated with assignments or procedure calls; a special *entry point* $e_q \in N_q$; and a special *return point* $r_q \in N_q$. We assume here that the program points of different procedures are disjoint.

Every address pointing somewhere into the global data-structures can be uniquely represented by an expression $B\,.\,s_1\,.\,\ldots\,.\,s_r$ where B is the base address of a global data-structure and each s_i is either a field selector or an array index in $\mathbb{Z}$. Since we consider addresses in fixed global data-structures only, the length r is bounded by some global constant d. Let $\mathcal{A}$ denote the set of all these addresses. Since we ignore the values stored in the global data-structures, a program state can be represented by a pair $\langle x, a\rangle$ where $x \in \mathbb{Z}^k$ and $a \in \mathcal{A}^m$ describe the values of the **int** variables and the address variables, respectively. We denote the set of all states by $\mathbb{S} = \mathbb{Z}^k \times \mathcal{A}^m$. Throughout this paper, we use k and m to denote the number of the (global) integer and address variables, and we use d to denote the maximal depth of data structures!

For an affine combination $t = t_0 + t_1\mathbf{x}_1 + \ldots + t_k\mathbf{x}_k$ and a state $\sigma = \langle x, a\rangle$, we write $[\![t]\!]\sigma$ for the value $t_0 + t_1x_1 + \ldots + t_kx_k \in \mathbb{Z}$. Likewise, for an address expression *adr* we write $[\![adr]\!]\sigma$ to denote the address obtained from *adr* by *substituting* the address variables in *adr* (if there are any) with their values in σ and by *evaluating* all affine index expressions w.r.t. the values of the **int**-variables in σ. Thus, the semantics of assignments for *sets* of states S is defined by:

$$[\![\mathbf{x}_j := t]\!]S = \{\langle(x_1, \ldots, x_{j-1}, [\![t]\!]\,\langle x,a\rangle\,, x_{j+1}, \ldots, x_k), a\rangle \mid \langle x,a\rangle \in S\} \quad (1)$$

$$[\![\mathbf{x}_j :=?]\!]S = \{\langle(x_1, \ldots, x_{j-1}, z, x_{j+1}, \ldots, x_k), a\rangle \mid \langle x,a\rangle \in S, z \in \mathbb{Z}\} \quad (2)$$

$$[\![\mathbf{a}_j := adr]\!]S = \{\langle x, (a_1, \ldots, a_{j-1}, [\![adr]\!]\,\langle x,a\rangle\,, a_{j+1}, \ldots, a_k)\rangle \mid \langle x,a\rangle \in S\} \quad (3)$$

$$\begin{aligned}[\![\mathbf{a}_j :=?]\!]S = \{\big\langle x, (a_1, \ldots, a_{j-1}, a'_j, a_{j+1} \ldots, a_k)\big\rangle \mid \langle x,a\rangle \in S,& \quad (4)\\ a'_j \in \mathcal{A} \text{ of appropriate type}\}&\end{aligned}$$

Every program execution π can be considered as a transformation $[\![\pi]\!] : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ of the set of states before the execution into the set of states after the execution. Here, we find it convenient to define the semantics as the transformation $\mathbf{R}[u] : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ that describes which program states can be attained at program point u when program execution starts in a given set of states. Given the transformation $\mathbf{R}[u]$, we can recover the *collecting semantics* of u, i.e., the set of all program states possibly attained during program execution when reaching u, as the set $\mathbf{R}[u](\mathbb{S})$.

In order to define the transformations $\mathbf{R}$, we additionally consider for every procedure q, the transformation of a set of program states before a call to q into the set of program states after the call. In order to determine this transformation, we introduce for every program point u of q, the auxiliary transformation $\mathbf{S}[u]$ which collects the transformation induced by the executions from u to the end point r_q of q at the same level, i.e., all recursive calls on its path towards the end of the procedure have returned. Then, the transformation of q is given by $\mathbf{S}[e_q]$ for the start point e_q of q, and we have:

$$\begin{array}{llll}
[\mathrm{S1}] & \mathbf{S}[r_q] & \supseteq \mathsf{Id} & \\
[\mathrm{S2}] & \mathbf{S}[u] & \supseteq \mathbf{S}[v] \circ [\![s]\!] & \text{if } (u,s,v) \text{ is an assignment edge}\\
[\mathrm{S3}] & \mathbf{S}[u] & \supseteq \mathbf{S}[v] \circ \mathbf{S}[e_q] & \text{if } (u,q,v) \text{ is a call edge}\\
 & & & \\
[\mathrm{R0}] & \mathbf{R}[e_{main}] & \supseteq \mathsf{Id} & \\
[\mathrm{R1}] & \mathbf{R}[e_q] & \supseteq \mathbf{R}[u] & \text{if } (u,q,\_) \text{ is a call edge}\\
[\mathrm{R2}] & \mathbf{R}[v] & \supseteq [\![s]\!] \circ \mathbf{R}[u] & \text{if } (u,s,v) \text{ is an assignment edge}\\
[\mathrm{R3}] & \mathbf{R}[v] & \supseteq \mathbf{S}[e_q] \circ \mathbf{R}[u] & \text{if } (u,q,v) \text{ is a call edge}
\end{array}$$

Here, the ordering "$\supseteq$" on transformers $f, g : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ is defined by $f \supseteq g$ iff for every set of states S, $f(S) \supseteq g(S)$.

3 Address Equalities

Our goal is to detect equalities between address expressions. In order to do so, we additionally need to track affine equalities between **int** variables. An affine equality is an assertion $t_0 + t_1\mathbf{x}_1 + \ldots + t_k\mathbf{x}_k \doteq 0$ for $t_0, \ldots, t_k \in \mathbb{Q}$. An address equality is an assertion of the form: $\mathit{adr} \doteq \mathit{adr}'$ of address expressions $\mathit{adr}, \mathit{adr}'$. Here, "$\doteq$" serves as a formal equality symbol. A program state σ *satisfies* the affine equality $t \doteq 0$ iff the left-hand side evaluates to zero: $[\![t]\!]\sigma = 0$. Likewise, the state σ satisfies the address equality $\mathit{adr} \doteq \mathit{adr}'$ iff $[\![\mathit{adr}]\!]\sigma = [\![\mathit{adr}']\!]\sigma$. This means that we consider the Herbrand interpretation for the operator "." as well as for base addresses and field selectors, but use an arithmetic interpretation for index expressions. The latter allows us to identify semantically equal index expressions, such as $\mathbf{x}_1 + 5 + 2\mathbf{x}_1$ and $5 + 3\mathbf{x}_1$.

The state σ satisfies a finite conjunction E of affine and address equalities iff σ satisfies every equality in E. In this case, we write $\sigma \models E$. Likewise for a set S of states, we write $S \models E$ iff $\sigma \models E$ for all $\sigma \in S$. The conjunction E is *valid* at a program point u, if E is satisfied by all states possible at u, i.e., $\mathbf{R}[u] \models E$.

Example 1. In the program from Figure 1, we are interested in the equalities which hold at program point 4. The set of states possible at this point is given by $\mathbf{R}[4] = \{\langle n, c\,.\,(n-1)\rangle \mid n > 0\}$, and thus the equality $\mathbf{a}_1 \doteq c\,.\,(-1 + \mathbf{x}_1)$ is valid at this program point. □

Given this notion of satisfiability, we say that a conjunction of equalities E implies another conjunction of equalities E', iff for all states $\sigma \in \mathbb{S}$, $\sigma \models E$ implies $\sigma \models E'$. Thus, the conjunctions of address and affine equalities can be ordered by implication "$\Rightarrow$". The greatest element $\top$ w.r.t. this ordering is the empty conjunction or true, as it is satisfied by all states. The bottom element $\bot$ in the ordering is false, denoting an unsatisfiable conjunction of equalities.

Consider a finite conjunction E with affine equalities $t_{i0} + t_{i1}\mathbf{x}_1 + \ldots + t_{ik}\mathbf{x}_k \doteq 0$, $i = 1, \ldots, h$. Assume that the conjunction E is satisfiable. Then, we say that it is in *canonical form* iff the following conditions are satisfied:

1. the affine equalities — more precisely, the corresponding coefficient matrix (t_{ij}) — is in row echelon form;[3]
2. the left-hand sides in the address equalities of E are pairwise distinct variables; and
3. no variable that is on the left-hand side of an address equality in E occurs in any of the right-hand sides.

By these restrictions, any conjunction in canonical form comprises at most k affine equalities as well as at most m address equalities.

Example 2. Take the conjunction $(\mathbf{a}_1\,.\,d \doteq c\,.\,(2\mathbf{x}_1)\,.\,d) \wedge (\mathbf{a}_1\,.\,m \doteq c\,.\,(\mathbf{x}_1)\,.\,m)$. An equivalent conjunction in canonical form is $(\mathbf{a}_1 \doteq c\,.\,(2\mathbf{x}_1)) \wedge (\mathbf{x}_1 \doteq 0)$. □

[3] A matrix is said to be in row echelon form if all zero rows are at the bottom, the leading entry of each non-zero row except the first occurs to the right of the leading entry of the previous row, and the leading entry of any non-zero row is 1.

Lemma 1. *For every finite conjunction of equalities E, a finite conjunction in canonical form which is equivalent to E can be constructed in polynomial time.*

Proof. Assume that the conjunction is of the form $E = E_a \wedge E_x$ where E_a is a conjunction of address equalities and E_x is a conjunction of affine equalities. We proceed in three steps. First, we replace every index expression t occurring in the conjunction E_a with the expression $\mathbf{x}_t$ for a fresh variable $\mathbf{x}_t$. Let E'_a denote the resulting conjunction of address equalities.

In the second step, we compute a most general unifier σ for E'_a w.r.t. the Herbrand interpretation. If unification succeeds, then due to the specific form of address expressions, the substitution σ will map each auxiliary variable $\mathbf{x}_t$ either to a field selector or to another auxiliary variable $\mathbf{x}_{t'}$. If there exists an $\mathbf{x}_t$, such that $\sigma(\mathbf{x}_t)$ is a field selector, then the conjunctions are inconsistent and the whole conjunction is equivalent to false.

Otherwise, let E'_x denote the conjunction of all equalities $t_1 - t_2 \doteq 0$ for which the corresponding auxiliaries $\mathbf{x}_{t_i}$ were unified, i.e., $\sigma(\mathbf{x}_{t_1}) = \sigma(\mathbf{x}_{t_2})$. Then E_a is equivalent to the conjunction of E'_x with $E''_a = \bigwedge_i (\mathbf{a}_i \doteq adr_i)$ where the address expressions adr_i are obtained from $\sigma(\mathbf{a}_i)$ by substituting back the affine index expressions t for the auxiliary variables $\mathbf{x}_t$.

Thus, a canonical form of the conjunction E is given by $E''_a \wedge E''_x$, where E''_x is the echelon form for the conjunction $E_x \wedge E'_x$. Using a linear unification algorithm [19] for computing σ, we conclude that the canonical form of E can be computed in time $\mathcal{O}((|E_x| + |E'_x|) \cdot k^2) = \mathcal{O}((s + r \cdot d) \cdot k^2)$ if E consists of s affine equalities and r address equalities. □

Note that we give the complexity estimates in this paper under the uniform cost measure, i.e., we assume a constant cost for arithmetic operations.

Lemma 2. *Assume E is a satisfiable conjunction of equalities in canonical form with k **int**-variables, and addresses of length at most d. Then the following holds:*

1. *For every affine combination t, $E \Rightarrow (t \doteq 0)$ can be decided in time $\mathcal{O}(k^2)$.*
2. *For every address expression adr, $E \Rightarrow (\mathbf{a}_i \doteq adr)$ can be decided in time $\mathcal{O}(d \cdot k^2)$.*

Proof. As the first statement is immediate from linear algebra, we only prove the second. Let us assume that $adr \equiv A\,.\,s_1\,.\,\ldots\,.\,s_h$, i.e., adr does not contain an address variable. Then the implication holds iff E contains an equality $\mathbf{a}_i \doteq A\,.\,s'_1\,.\,\ldots\,.\,s'_h$, and for each $\lambda = 1, \ldots, h$, the access expressions s_λ and s'_λ are equal under E: either both s_λ and s'_λ are field selectors and identical, or both s_λ and s'_λ are index expressions and $E \Rightarrow (s_\lambda - s'_\lambda \doteq 0)$.

Now assume that $adr \equiv \mathbf{a}_j\,.\,s_1\,.\,\ldots\,.\,s_h$ for some address variable $\mathbf{a}_j$. Unless $adr \equiv \mathbf{a}_i$, the implication can only hold if E also contains an equality for $\mathbf{a}_i$. Moreover, this equality is of the form $\mathbf{a}_i \doteq a\,.\,s'_1\,.\,\ldots\,.\,s'_{h+l}$ for some $l \geq 0$ where a is either an address constant A or an address variable $\mathbf{a}_r$. Then the implication holds iff E also contains an equality $\mathbf{a}_j \doteq a\,.\,s''_1\,.\,\ldots\,.\,s''_l$ where for $\lambda = 1, \ldots, l$, the accesses s'_λ and s''_λ are equal under E, and for $\lambda = l{+}1, \ldots, h$, the accesses s'_λ and

$s_{\lambda-l}$ are equal under E. Assuming that the address equality in E for particular address variables can be retrieved in constant time, at most d affine equalities must be checked for subsumption by E — giving us the stated complexity bound. □

Thus, both logical implication and equivalence between satisfiable conjunctions E, E' in canonical form can be decided in time $\mathcal{O}((m^2 \cdot d + k) \cdot k^2)$.

Let $\mathbb{E}$ denote the set of equivalence classes of finite conjunctions ordered by implication. The greatest lower bound of (the equivalence classes of) two conjunctions $E, E' \in \mathbb{E}$ is (the equivalence class containing) the conjunction of all the equalities in E and E'. The partial order $\mathbb{E}$ thus is a complete lattice — given that all descending chains are finite.

Corollary 1. *Every chain $E_0 \Rightarrow \ldots \Rightarrow E_p$ of pairwise inequivalent conjunctions E_j using k **int** variables and m address variables has length $p \leq m + k + 1$.*

This follows because any two inequivalent conjunctions E_i and E_j have counterparts in canonical form, E_i' and E_j', respectively. The implication $E_i' \Rightarrow E_j'$ can only hold, if E_i' contains strictly more equalities than E_j'. Therefore, all chains in the lattice will eventually stabilize after at most $m + k + 1$ steps.

In summary, we have proven that the set of equivalence classes of conjunctions of address equalities ordered with implication $(\mathbb{E}, \Rightarrow)$ is a complete lattice.

4 Weakest pre-conditions

Our approach to computing all valid equalities is based on an effective weakest pre-condition computation. For a conjunction of equalities E, the weakest pre-condition for an assignment and a non-deterministic assignment is given by substitution and universal quantification, respectively:

$$\begin{array}{ll} [\![\mathbf{x}_i := t]\!]^\top(E) = E[t/\mathbf{x}_i] & \qquad [\![\mathbf{a}_i := a]\!]^\top(E) = E[a/\mathbf{a}_i] \\ [\![\mathbf{x}_i := ?]\!]^\top(E) = \forall\, \mathbf{x}_i.\, E & \qquad [\![\mathbf{a}_i := ?]\!]^\top(E) = \forall\, \mathbf{a}_i.\, E \end{array}$$

While our domain is closed under substitution, it does not directly support universal quantification. We are rescued by the fact that in the sub-domain of linear arithmetic, determining the weakest pre-condition for a non-deterministic assignment to an **int** variable $\mathbf{x}_i$, it suffices to consider the conjunction of the weakest pre-conditions of the assignments $\mathbf{x}_i := 0$ and $\mathbf{x}_i := 1$ [13]. On the other hand, $\forall\, \mathbf{a}_i.\, E$ for a conjunction E in canonical form involving the address variable $\mathbf{a}_i$ is necessarily false, if $\mathbf{a}_i$ can range over at least two addresses [16]. For simplicity of presentation, let us assume there are no singleton types. Thus, the weakest pre-conditions for non-deterministic assignments can be simplified:

$$[\![\mathbf{x}_i := ?]\!]^\top(E) = E[0/\mathbf{x}_i] \wedge E[1/\mathbf{x}_i]$$

$$[\![\mathbf{a}_i := ?]\!]^\top(E) = \begin{cases} \mathsf{false} & \text{if } \mathbf{a}_i \text{ occurs in } E \\ E & \text{otherwise} \end{cases}$$

Note that these results do not hold for the general combination of linear arithmetic with uninterpreted functions.

We now set up a constraint system to characterize the weakest pre-condition *transformers* $\mathbf{R}^{\mathsf{T}}[v]$, which transform conjunctions of equalities at the program point v into the weakest pre-condition for their validity at *program start.* The constraint system uses auxiliary transformers $\mathbf{S}^{\mathsf{T}}[v]$, which transform the post-condition of a procedure q into the weakest pre-condition at the program point v of the same procedure q.

$$\begin{array}{llll}
[\mathrm{S1}^{\mathsf{T}}] & \mathbf{S}^{\mathsf{T}}[r_q] & \Rightarrow \mathsf{Id} & \\
[\mathrm{S2}^{\mathsf{T}}] & \mathbf{S}^{\mathsf{T}}[u] & \Rightarrow [\![s]\!]^{\mathsf{T}} \circ \mathbf{S}^{\mathsf{T}}[v] & (u, s, v) \text{ an assignment edge} \\
[\mathrm{S3}^{\mathsf{T}}] & \mathbf{S}^{\mathsf{T}}[u] & \Rightarrow \mathbf{S}^{\mathsf{T}}[e_q] \circ \mathbf{S}^{\mathsf{T}}[v] & (u, q, v) \text{ a call edge}
\end{array}$$

$$\begin{array}{llll}
[\mathrm{R0}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_{main}] & \Rightarrow \mathsf{Id} & \\
[\mathrm{R1}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_q] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] & (u, q, \_) \text{ a call edge} \\
[\mathrm{R2}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ [\![s]\!]^{\mathsf{T}} & (u, s, v) \text{ an assignment edge} \\
[\mathrm{R3}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ \mathbf{S}^{\mathsf{T}}[e_q] & (u, q, v) \text{ a call edge}
\end{array}$$

Here, the ordering "$\Rightarrow$" on transformers $f, g \colon \mathbb{E} \to \mathbb{E}$ is defined by $f \Rightarrow g$ iff for all conjunctions of equalities E, $f(E) \Rightarrow g(E)$. The greatest solution to the system will be the weakest pre-condition transformers. We state this as a theorem.

Theorem 1. *For every program point u, set of states $S \subseteq \mathbb{S}$, and conjunction of equalities $E \in \mathbb{E}$,*

$$\mathbf{S}[u](S) \models E \iff S \models \mathbf{S}^{\mathsf{T}}[u](E) \quad \textit{and} \quad \mathbf{R}[u](\mathbb{S}) \models E \iff \mathbf{R}^{\mathsf{T}}[u](E) = \mathsf{true}$$

Proof. The identity and weakest pre-condition transformers for individual edges are defined in a standard way. Relating the least fixed point of the system $\mathbf{S}$ with the greatest fixed point of the system $\mathbf{S}^{\mathsf{T}}$, we are only required to show that the following conditions are satisfied:

$$\begin{aligned}
f(S) \cup g(S) \models E &\iff S \models f^{\mathsf{T}}(E) \wedge g^{\mathsf{T}}(E) \\
(f \circ g)(S) \models E &\iff S \models (g^{\mathsf{T}} \circ f^{\mathsf{T}})(E).
\end{aligned}$$

These follow from the properties of weakest pre-condition transformers. The second equivalence follows from an analogous fixed-point induction and the fact that $\mathbb{S} \models E$ only if $\mathsf{true} \Rightarrow E$. □

Example 3. In our example program, the weakest predicate transformers for program points 2, 3 and 4 are given by the constraints:

$$\begin{array}{ll}
\mathbf{R}^{\mathsf{T}}[2] \Rightarrow [0/\mathbf{x}_1] & \mathbf{R}^{\mathsf{T}}[2] \Rightarrow \mathbf{R}^{\mathsf{T}}[4] \\
\mathbf{R}^{\mathsf{T}}[3] \Rightarrow \mathbf{R}^{\mathsf{T}}[2] \circ [c\,.\,(\mathbf{x}_1)/\mathbf{a}_1] & \mathbf{R}^{\mathsf{T}}[4] \Rightarrow \mathbf{R}^{\mathsf{T}}[3] \circ [\mathbf{x}_1 + 1/\mathbf{x}_1]
\end{array}$$

Using methods described below, we find that $\mathbf{R}^{\mathsf{T}}[2]$ maps the post-condition $\mathbf{a}_1 \doteq c\,.\,(-1 + \mathbf{x}_1)$ to the pre-condition $\mathbf{a}_1 \doteq c\,.\,(-1)$. □

Solving such constraint systems requires effective computation of function comparisons, greatest lower bounds and compositions. Thus, we need an finite and effective representation of these predicate transformers.

5 Finite representation

Inspired by order-theory, let us call single address equalities $\mathbf{a}_i \doteq \mathit{adr}$ and affine equalities $t \doteq 0$ *atomic*. Let $\mathbb{E}_A$ denote the set of atomic equalities. According to Lemma 1, every conjunction has a canonical form, which is a conjunction of atomic equalities. Hence, every transformer $f\colon \mathbb{E} \to \mathbb{E}$, which is completely distributive, i.e., preserves true and distributes over conjunctions, is uniquely determined by its restriction $f|_{\mathbb{E}_A}$ to atomic equalities.

This observation, though, does not provide yet a finite representation of weakest pre-condition transformers, since the number of single equalities is still infinite. The second idea, therefore, is not to track weakest pre-conditions for each equality separately, but to consider *generic equalities*. Every generic equality serves as a template which covers a range of equalities of similar form simultaneously.

In order to infer weakest pre-conditions for all affine equalities, we consider the generic post-condition $p \equiv \mathbf{p}_0 + \mathbf{p}_1\mathbf{x}_1 + \cdots + \mathbf{p}_k\mathbf{x}_k \doteq 0$, where $\mathbf{p}_0, \ldots, \mathbf{p}_k$ are fresh variables not occurring in the program. The weakest pre-conditions for p can be represented as conjunctions of equalities

$$\sum_{i=0}^{k} c_{i0}\mathbf{p}_i + \sum_{i=0}^{k}\sum_{j=1}^{k} c_{ij}\mathbf{p}_i\mathbf{x}_j \;\doteq\; 0 \tag{5}$$

for constants $c_{ij} \in \mathbb{Q}$.

Example 4. Since our running example has just one **int** variable, the generic affine post-condition is is $e_{\mathit{aff}} \equiv \mathbf{p}_0 + \mathbf{p}_1\mathbf{x}_1 \doteq 0$. The parametric pre-condition for e_{aff} w.r.t. the assignment $\mathbf{x}_1 \mathrel{:=} \mathbf{x}_1 + 1$ is then $\mathbf{p}_0 + \mathbf{p}_1 + \mathbf{p}_1\mathbf{x}_1 \doteq 0$. □

A generic address post-condition is of the form $\mathbf{a}_i \doteq a\,.\,s_1\,.\,\ldots\,.\,s_r$ (for some $r \leq d$) where a is either an address constant in $\mathcal{C}$ or another address variable in $\mathbf{A}$, and each s_l is either a field name or an indexing pattern $\mathbf{p}_{l0} + \mathbf{p}_{l1}\mathbf{x}_1 + \cdots + \mathbf{p}_{lk}\mathbf{x}_k$. Weakest pre-conditions for such a generic address post-condition will be conjunctions of parametric affine equalities and parametric address equalities. The generic coefficients to be considered in the parametric affine equalities now are elements from the set $\mathbf{P}_r = \{\mathbf{p}_{li} \mid l \in [1,r], i \in [0,k]\}$. Thus, the affine equalities are of the form:

$$c_{000} + \sum_{j=1}^{k} c_{00j}\mathbf{x}_j + \sum_{l=1}^{r}\sum_{i=0}^{k} c_{li0}\mathbf{p}_{li} + \sum_{l=1}^{r}\sum_{i=0}^{k}\sum_{j=1}^{k} c_{lij}\mathbf{p}_{li}\mathbf{x}_j \;\doteq\; 0 \tag{6}$$

for constants $c_{lij} \in \mathbb{Q}$. Also, the parametric address equalities will be address equalities where index expressions are of the same form as left-hand sides in (6).

Example 5. For the address variable $\mathbf{a}_1$, a generic post-condition is of the form $e_{adr} \equiv \mathbf{a}_1 \doteq c\,.\,(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1)$. The parametric pre-condition for e_{adr} w.r.t. the assignment $\mathbf{a}_1 := c\,.\,(\mathbf{x}_1)$ is given by $c\,.\,(\mathbf{x}_1) \doteq c\,.\,(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1)$, whose canonical form is $-\mathbf{x}_1 + \mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1 \doteq 0$. □

The conjunction of parametric equalities forms a lattice $\mathbb{E}_d$, which has the same structure as the lattice $\mathbb{E}$ – except that the set of **int** variables is now extended with the set of parameters $\mathbf{p}_{l0}$ and products $\mathbf{p}_{li}\mathbf{x}_j$ of parameters and **int** variables. The height of the complete lattice $\mathbb{E}_d$ therefore is bounded by $\mathcal{O}(d \cdot k^2 + m)$.

In our application, generic post-conditions suffice to arrive at a finite specification of weakest pre-condition transformers. Let T denote the set of well-typed generic address equalities. Then the set T is finite and of cardinality $\mathcal{O}(m^2 \cdot t \cdot d)$, where t is the maximal size, i.e., number of fields, of a global data structure's type. This set T is *complete* in the sense that for any concrete atomic equality $e \in \mathbb{E}_A$, there exists a substitution $\sigma\colon \mathbf{P}_d \to \mathbb{Q}$ and a generic post-condition $e' \in T$ such that $e = e'\sigma$. Any function $f\colon T \to \mathbb{E}_d$ can be extended to a completely distributive function $\mathsf{ext}\,(f)\colon \mathbb{E} \to \mathbb{E}$ defined by $(\mathsf{ext}\,(f))(e) = (f(e'))\sigma$ for all atomic equalities $e = e'\sigma$ (e' is a generic equality, σ a substitution).

We now show that the weakest predicate transformers that occur in our constraint system can indeed be obtained as extensions of functions from $T \to \mathbb{E}_d$. In order to do so, we set up a new constraint system $\mathbf{R}^\sharp$ over functions from $T \to \mathbb{E}_d$. This is obtained from the constraint system $\mathbf{R}^\top$ by replacing all operations by their parametric counterparts. Thus, implication "$\Rightarrow^\sharp$" and greatest lower bounds $\wedge^\sharp$ are now defined according to the domain $\mathbb{E}_d$. Also, the transfer functions for assignments are lifted to parametric equalities. It remains to define composition $\circ^\sharp$ for functions $f^\sharp, g^\sharp\colon T \to \mathbb{E}_d$.

First, we observe that every parametric equality can be obtained from one of the generic post-conditions by a transformation σ of the parameters. Therefore assume that e' is a generic post-condition and $g^\sharp(e') = e_1 \wedge \cdots \wedge e_r$ where $e_l = e'_l\sigma_l$ for generic post-conditions e'_l and linear transformations σ_l. Then we define

$$(f^\sharp \circ^\sharp g^\sharp)(e') = (f^\sharp(e'_1))\sigma_1 \wedge \cdots \wedge (f^\sharp(e'_r))\sigma_r\,.$$

If e' is the generic affine equality, this amounts to computing the canonical form of a conjunction of $\mathcal{O}(k^4)$ parametric equalities. If e' is a generic address equality, the canonical form must be computed for a conjunction of $\mathcal{O}(m^2)$ parametric address equalities and $\mathcal{O}(d^2k^4)$ parametric affine equalities whose normalization may at worst consume time $\mathcal{O}(m^2 \cdot d^4 \cdot k^8)$.

Example 6. Let $f = [\![\mathbf{x}_1 := ?]\!]^\top$ and $g = [\![\mathbf{a}_1 := c\,.\,(\mathbf{x}_1)]\!]^\top$. We then compute the composition $(f \circ g)(e_{adr})$ as follows:

$$\begin{aligned}(f \circ g)(e_{adr}) &= f(-\mathbf{x}_1 + \mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1 \doteq 0)\\ &= (f(e_{\mathit{aff}}))\,\sigma \quad \text{for } \sigma = [\mathbf{p}_{10}/\mathbf{p}_0, (-1 + \mathbf{p}_{11})/\mathbf{p}_1]\\ &= ((\mathbf{p}_0 \doteq 0) \wedge (\mathbf{p}_0 + \mathbf{p}_1 \doteq 0))\sigma\\ &= ((\mathbf{p}_0 \doteq 0) \wedge (\mathbf{p}_1 \doteq 0))\,\sigma = (\mathbf{p}_{10} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0)\end{aligned}$$

This computation occurs during the analysis of our running example, because the while-loop has the same effect as the non-deterministic assignment of f. □

Theorem 2. *For any program point u,* $\mathbf{R}^{\mathsf{T}}[u] = \mathsf{ext}\,(\mathbf{R}^{\sharp}[u])$.

Proof. We proceed by fixpoint induction. A crucial step is to show that not only "$\wedge$", but also composition commutes with ext, i.e., that

$$\mathsf{ext}\,(f) \circ \mathsf{ext}\,(g) = \mathsf{ext}\,(f \circ^{\sharp} g)$$

To see that, we calculate:

$$\begin{array}{rll} (\mathsf{ext}\,(f) \circ \mathsf{ext}\,(g))(e'\sigma) & = \mathsf{ext}\,(f)(\mathsf{ext}\,(g)(e'\sigma) & = \mathsf{ext}\,(f)(g(e'))\sigma) \\ & = \mathsf{ext}\,(f)((\bigwedge e'_i\sigma_i)\sigma) & = \mathsf{ext}\,(f)(\bigwedge e'_i(\sigma_i\sigma)) \\ & = \bigwedge(f(e'_i))(\sigma_i\sigma) & = (\bigwedge(f(e'_i))\sigma_i)\sigma \\ & = ((f \circ^{\sharp} g)(e'))\sigma & = (\mathsf{ext}\,(f \circ^{\sharp} g))(e'\sigma) \end{array}$$

where $g(e') = \bigwedge e'_i\sigma_i$ as above. □

Example 7. We can now compute the solution to the constraint system by fixpoint iteration starting from true. The computation stabilizes after three iterations, giving the following pre-conditions for the address post-condition:

$$\begin{aligned} \mathbf{R}^{\mathsf{T}}[2](e_{adr}) &= (\mathbf{a}_1 \doteq c\,.(\mathbf{p}_{10})) \wedge (\mathbf{p}_{10} + \mathbf{p}_{11} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0) \\ \mathbf{R}^{\mathsf{T}}[3](e_{adr}) &= (\mathbf{p}_{10} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0) \\ \mathbf{R}^{\mathsf{T}}[4](e_{adr}) &= (\mathbf{p}_{10} + \mathbf{p}_{11} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0) \end{aligned}$$

For the affine post-condition e_{aff}, the pre-condition $(\mathbf{p}_0 \doteq 0) \wedge (\mathbf{p}_1 \doteq 0)$ is obtained, meaning no non-trivial affine equalities hold at these points. □

6 Computing all valid equalities

Given the weakest pre-condition transformer $\mathbf{R}^{\mathsf{T}}[v]$ for program point v, computing all equalities which are valid at v then boils down to solving a suitable inhomogeneous system of equations. We have:

Theorem 3. *The equalities that hold at each program point can be computed in polynomial time.*

Proof. Let e' denote a generic post-condition and $e = e'\sigma$ an atomic equality for some substitution σ of the parameters occurring in e'. By Theorem 1, e holds at program point u iff $\mathbf{R}^{\mathsf{T}}[u](e) = \mathsf{true}$, which, by Theorem 2, means that $(\mathbf{R}^{\sharp}[u](e'))\sigma = \mathsf{true}$. The latter means that $\mathbf{R}^{\sharp}[u](e')$ does not contain nontrivial address equalities, but is a conjunction of at most $\mathcal{O}(d\cdot k^2)$ affine equalities $t \doteq 0$ where $t\sigma \doteq 0$ is valid for all values $x \in \mathbb{Z}^k$.

Assume that $\mathbf{p}'_1, \ldots, \mathbf{p}'_r$ are the parameters occurring in t, the affine combination t is of the form: $t \equiv c_{00} + \sum_{i=1}^{k} (c_{0i} + \sum_{l=1}^{r} c_{li}\mathbf{p}'_l)\,\mathbf{x}_i$ for suitable $c_{li} \in \mathbb{Q}$. Then $t\sigma \doteq 0$ is valid for all values $x \in \mathbb{Z}^k$ iff $c_{00} = 0$ and σ is a solution of each of the equations $c_{0i} + \sum_{l=1}^{r} c_{li}\mathbf{p}'_l \doteq 0$ $(i = 1, \ldots, k)$. We conclude that finding all substitutions σ such that $e'\sigma$ is valid at program point u can be reduced to solving a system of $\mathcal{O}(k \cdot d \cdot k^2) = \mathcal{O}(d \cdot k^3)$ inhomogeneous equations over $\mathbb{Q}$ where the number of unknowns is bounded by $d \cdot (k+1)$. The latter task can be done with a polynomial number of arithmetic operations. By repeating this procedure for every possible generic post-condition, we obtain a finite representation of all equalities which are valid at program point u. □

Example 8. As we saw in Example 7, at all points in the loop the parametric pre-condition for e_{aff} has $\mathbf{p}_0 = \mathbf{p}_1 = 0$ as its solution. The parametric pre-condition for the generic post-condition e_{adr}, on the other hand, is given by:

$$\mathbf{R}^{\mathsf{T}}[4](\mathbf{a}_1 \doteq c\,.(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1)) \;=\; (\mathbf{p}_{10} + \mathbf{p}_{11} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0)$$

As no **int**-variables $\mathbf{x}_i$ are involved here, this pre-condition is true iff $\mathbf{p}_{11} = 1$ and $\mathbf{p}_{10} = -1$. Therefore, the only non-trivial equality which holds at program point 4 is $\mathbf{a}_1 \doteq c\,.(-1 + \mathbf{x}_1)$. □

To summarize, the set of all equalities, which hold at a given program point, can be compactly represented by a polynomially sized set of triples $\langle e, \sigma, V\rangle$ — each consisting of a generic post-condition e together with one particular solution for the conjunction of parametric affine pre-conditions of e and a basis V of the vector space of solutions of the corresponding homogeneous system. Assuming that the basis V is in (column) echelon form, we can determine if a given equality holds at a certain program point in time $\mathcal{O}(d^2 \cdot k^2)$.

7 Local variables

All program variables have so far been considered *global.* Along the lines of [15], we now extend the analysis to possibly recursive programs with local variables as well. From the k integer variables, we consider the first $k' \leq k$ variables $\mathbf{x}_1, \ldots, \mathbf{x}_{k'}$ as local and the remaining ones as global. Similarly for pointers, the first $m' \leq m$ variables $\mathbf{a}_1, \ldots, \mathbf{a}_{m'}$ denote local variables while the remaining ones denote global pointer variables.

For passing of parameters, we adopt w.l.o.g. the convention that *all* locals of the caller are passed by value into the locals of the callee. This enables us to reason about equalities involving local variables of the caller.

We extend the concrete semantics with an extra operator H which transforms the effect of a procedure body into the effect of a procedure call:

$$\begin{aligned}\mathsf{H}(f)(S) = \{\langle (x_1, \ldots, x_{k'}, x'_{k'+1}, \ldots, x'_k), (a_1, \ldots, a_{m'}, a'_{m'+1}, \ldots, a'_m)\rangle \mid \\ \langle x, a\rangle \in S, \langle x', a'\rangle \in f(\{\langle x, a\rangle\})\}\end{aligned}$$

The constraint system for computing weakest pre-conditions of procedure calls is modified accordingly by introducing the operator H^{T}:

[S1$^{\mathsf{T}}$]	$\mathbf{S}^{\mathsf{T}}[r_q] \Rightarrow \mathsf{Id}$	
[S2$^{\mathsf{T}}$]	$\mathbf{S}^{\mathsf{T}}[u] \;\Rightarrow [\![s]\!]^{\mathsf{T}} \circ \mathbf{S}^{\mathsf{T}}[v]$	(u, s, v) an assignment edge
[S3$^{\mathsf{T}}$]	$\mathbf{S}^{\mathsf{T}}[u] \;\Rightarrow \mathsf{H}^{\mathsf{T}}(\mathbf{S}^{\mathsf{T}}[e_q]) \circ \mathbf{S}^{\mathsf{T}}[v]$	(u, q, v) a call edge

Here, the operator H^T must be defined such that statement 1 of Theorem 1 holds for the new constraint system. Given the concrete transformer f of a procedure and the corresponding weakest pre-condition transformer f^T, the following condition must hold for all sets of states S and conjunctions of equalities E:

$$\mathsf{H}(f)(S) \models E \iff S \models \mathsf{H}^\mathsf{T}(f^\mathsf{T})(E)$$

Consider an arbitrary post-condition E for a procedure call to f. This post-condition may not only speak about globals, but also about locals of the caller as well as any local variable further down in the call-stack. All these locals, however, are inaccessible during the execution of the procedure f and thus can temporarily be considered as *constants*. In order to deal with these temporary constants, we introduce place holders $\bullet_\tau$ for every possible type of local pointer variables a_j or constant addresses.

Accordingly, we consider the following set of parametric post-conditions E':

$$\begin{array}{llll} (1) & \mathbf{a}_i \doteq \mathbf{a}_j \,.\, s & (2) & \mathbf{a}_i \doteq \bullet_\tau \,.\, s \\ (3) & \bullet_\tau \doteq \mathbf{a}_i \,.\, s & (4) & \bullet_{\tau_1} \doteq \bullet_{\tau_2} \,.\, s \end{array}$$

for global pointer variables a_i, a_j and type-compatible parametric sequences of selectors s, where each parametric index is of the form $\mathbf{p}_{l0} + \mathbf{p}_{l(k'+1)}\mathbf{x}_{k'+1} + \cdots + \mathbf{p}_{lk}\mathbf{x}_k$. Furthermore, we consider the parametric affine post-condition:

$$(5) \qquad \mathbf{p}_0 + \mathbf{p}_{k'+1}\mathbf{x}_{k'+1} + \cdots + \mathbf{p}_k\mathbf{x}_k \doteq 0$$

for global variables $\mathbf{x}_{k'+1}, \ldots, \mathbf{x}_k$. Assume now that we are given the weakest pre-conditions $f^\mathsf{T}(E')$ of the called procedure for all these post-conditions E' speaking about global variables (and perhaps $\bullet_\tau$).

We now define the weakest pre-condition $\mathsf{H}^\mathsf{T}(f^\mathsf{T})(E)$. In each case, we decompose $E = E'\sigma$ for a generic post-condition E' of one of the types (1) through (5) and a suitable substitution σ. Then, we define

$$\mathsf{H}^\mathsf{T}(f^\mathsf{T})(E) = (f^\mathsf{T}(E'))\sigma\,.$$

It only remains to explain the decomposition of E. We first consider a post-condition E of the form $\mathbf{a}_i \doteq \mathbf{a}_j \,.\, s$ for global variables $\mathbf{a}_i, \mathbf{a}_j$. Then E' is of the parametric post-condition of format (1). For every index expression $s_l = t_0 + t_1\mathbf{x}_1 + \cdots + t_k\mathbf{x}_k$ in s, σ maps p_{l0} to the affine combination consisting of t_0 together with all occurring multiples of locals, i.e., to $t_0 + t_1\mathbf{x}_1 + \cdots + t_{k'}\mathbf{x}_{k'}$.

If E is of the form $\mathbf{a}_i \doteq X \,.\, s$ where $\mathbf{a}_i$ is a global variable and X either is a local of the caller, a constant address or a place holder $\bullet_\tau$ all of the type τ, we choose E' of the parametric format (2) and σ is constructed as before, but moreover maps the place holder $\bullet_\tau$ in E' to X.

The case where E is of the form $X \doteq \mathbf{a}_i \,.\, s$ is treated analogously. In case where E is of the form $X_1 \doteq X_2 \,.\, s$ and each X_i is a local of the caller, constant address or place holder, then we choose the appropriate generic post-condition E' now of type (4). The substitution σ treats index expressions as before, but now maps $\bullet_{\tau_1}$ to X_1 and $\bullet_{\tau_2}$ to X_2.

Finally, if E is an affine equality $t_0 + t_1\mathbf{x}_1 + \cdots + t_k\mathbf{x}_k \doteq 0$, then we choose E' to be of format (5) where the substitution σ maps p_0 to $t_0 + t_1\mathbf{x}_1 + \cdots + t_{k'}\mathbf{x}_{k'}$, and p_i to t_i for $i > k'$.

Example 9. Consider the post-condition $\mathbf{a}_1 \doteq \mathbf{a}_2 \,.\, (\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1 + \mathbf{p}_{12}\mathbf{x}_2)$, where $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{x}_1$ are local, but $\mathbf{x}_2$ is global and may be changed during the procedure call. Assume that the callee only performs the statement $\mathbf{x}_2 \mathrel{:=} ?$. Since the post-condition is of the type (4), we compute the pre-condition as follows:

$$\begin{array}{l} \llbracket \mathbf{x}_2 \doteq ? \rrbracket^{\mathsf{T}} (\bullet_{\tau_1} \doteq \bullet_2 \,.\, (\mathbf{p}_{10} + \mathbf{p}_{12}\mathbf{x}_2)) = \\ (\bullet_{\tau_1} \doteq \bullet_2 \,.\, (\mathbf{p}_{10})) \wedge (\bullet_{\tau_1} \doteq \bullet_2 \,.\, (\mathbf{p}_{10} + \mathbf{p}_{21})) = (\bullet_{\tau_1} \doteq \bullet_2 \,.\, (\mathbf{p}_{10})) \wedge (\mathbf{p}_{21} \doteq 0) \end{array}$$

To obtain the weakest pre-condition of $\mathbf{a}_1 \doteq \mathbf{a}_2 \,.\, (\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1 + \mathbf{p}_{12}\mathbf{x}_2)$, we apply the substitution σ, which maps $\mathbf{p}_{10}$ to $\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1$ and replaces the place-holders with the local address variables: $\mathbf{a}_1 \doteq \mathbf{a}_2 \,.\, (\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1) \wedge \mathbf{p}_{21} \doteq 0$. □

The second part of our analysis applies the weakest pre-condition transformers of procedures, as defined through the first part of the constraint system, to construct a constraint system for the weakest pre-condition transformers for post-conditions at program points v:

$$\begin{array}{llll} [\mathrm{R0}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_{main}] \Rightarrow \mathsf{Id} & & \\ [\mathrm{R1}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_q] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] & (u, q, \_) \text{ a call edge} \\ [\mathrm{R2}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ \llbracket s \rrbracket^{\mathsf{T}} & (u, s, v) \text{ an assignment edge} \\ [\mathrm{R3}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ \mathsf{H}^{\mathsf{T}}(\mathbf{S}^{\mathsf{T}}[e_q]) & (u, q, v) \text{ a call edge} \end{array}$$

This time, however, the post-conditions for the weakest pre-condition transformer $\mathbf{R}^{\mathsf{T}}[v]$ for a program point of a procedure f need not use $\bullet$-variables to refer to variables deeper down in the call-stack. Instead, they may refer to the *locals* of f. Accordingly, occurring transformers are described by their weakest pre-conditions for the generic affine post-condition together with the generic address post-conditions $\mathbf{a}_i \doteq \mathbf{a}_j \,.\, s$ for local or global address variables a_i, a_j and suitable selector sequences s.

8 Example Application: Race detection

One common approach to data race analysis is to ensure the following condition for every pair of accesses in the program: if the two access expressions *may* alias, then the acquired lock expressions *must* alias [17]. We ensure this condition by inferring access correlations using the must-equality analysis and associating these correlations with may-alias equivalence classes, as we will illustrate through the following example.

Example 10. Assume the address variables $\mathbf{a}_{acc}$ and $\mathbf{a}_{lock}$ represent an access expression and a lock expression that need to be correlated, and our must-alias analysis provides the following information:

$$(\mathbf{a}_{acc} \doteq \mathbf{a}_1 \,.\, data \,.\, (\mathbf{x}_1)) \wedge (\mathbf{a}_{lock} \doteq \mathbf{a}_1 \,.\, mutex \,.\, (\mathbf{x}_1))$$

These equalities imply that the access to the data array of the structure pointed to by $\mathbf{a}_1$ is protected by a corresponding element in the mutex array. □

The access pattern we can infer in the above example depends on the information we have about $\mathbf{a}_1$. If the analysis can infer that $\mathbf{a}_1$ is definitely equal to some statically allocated structure c, a pattern for access to the elements of c is obtained. Otherwise, may-alias analysis [7] is called upon to divide the set of all pointer variables into equivalence classes. The simplest such approach, which suffices for some applications [11], equates all pointers of the same type. Then our method allows to infer access patterns for data structures of a given type. A more refined analysis distinguishes heap objects depending also on their allocation sites, in which case our analysis derives more refined patterns.

Note that must-equality information complements may-aliasing by ensuring that $\mathbf{a}_{acc}$ and $\mathbf{a}_{lock}$ are referring to the *same object* within the equivalence class of $\mathbf{a}_1$. This is crucial in order to verify per-element locking schemes, where each element in, e.g., a linked list has its own lock. Pratikakis et al. [20] describe a technique based on existentially typed label-flow to address this issue with the aid of programmer annotations; must-equality information allows one to infer per-element correlations automatically.

9 Conclusion

We have presented a must-alias analysis which infers all equalities between address expressions and can be proven to be valid w.r.t. the chosen abstraction. In this abstraction, conditional branching is replaced with non-deterministic branching and pointers stored in the shared data-structures are not tracked. We indicated how these equalities can be used to infer correlations between locks and accesses. Our analysis infers relevant must-equality information also for dynamically allocated data which combined with may-alias information allows one to infer access patterns. A variant of this approach has been implemented in the Goblint data race analyzer [23] and also extended by an accompanying may-alias analysis [21].

For simplicity, we have assumed that index expressions are evaluated over the integral domain $\mathbb{Z}$. Instead, we could have chosen $\mathbb{Z}_{2^w}$, i.e., integers modulo a suitable power of 2, by replacing the linear algebra methods for vector spaces of affine equalities with the corresponding methods for modules over the principal ideal ring $\mathbb{Z}_{2^w}$ [14]. However, if the programs to be analyzed only employ simple forms of index expressions, it might be sufficient to replace tracking of affine equalities with tracking of variable equalities alone [15].

References

1. A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *CAV'02*, LNCS, vol. 2404, pages 654–663. Springer, 2002.

2. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI'94*, pages 230–241. ACM Press, 1994.
3. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *SAS'04*, LNCS, vol. 3148, pages 212–227. Springer, 2004.
4. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *ESOP'06*, LNCS, vol. 3924, pages 279–293. Springer, 2006.
5. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV'07*, LNCS, vol. 4590, pages 379–392. Springer, 2007.
6. S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP'07*, LNCS, vol. 4421, pages 253–267. Springer, 2007.
7. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Prog. Lang. Syst.*, 21(4):848–894, 1999.
8. G. J. Holzmann. The power of ten: Rules for developing safety critical code. *IEEE Computer*, 39(6):95–97, 2006.
9. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.
10. G. A. Kildall. A unified approach to global program optimization. In *POPL'73*, pages 194–206. ACM Press, 1973.
11. S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP'07*, pages 103–116. ACM Press, 2007.
12. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand equalities and beyond. In *VMCAI'05*, LNCS, vol. 3385, pages 79–96. Springer, 2005.
13. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL'04*, pages 330–341. ACM Press, 2004.
14. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Trans. Prog. Lang. Syst.*, 29(5), 2007.
15. M. Müller-Olm and H. Seidl. Upper adjoints for fast inter-procedural variable equalities. In *ESOP'08*, LNCS, vol. 4690, pages 178–192. Springer, 2008.
16. M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Herbrand equalities. In *ESOP'05*, LNCS, vol. 3444, pages 31–45. Springer, 2005.
17. M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07*, pages 327–338. ACM Press, 2007.
18. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Syst.*, 1(2):245–257, 1979.
19. M. Paterson and M. N. Wegman. Linear unification. In *STOC'76*, pages 181–186. ACM Press, 1976.
20. P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *SAS'06*, LNCS, vol. 4134, pages 88–106. Springer, 2006.
21. H. Seidl and V. Vojdani. Region analysis for race detection. In *SAS'09*, LNCS, vol. 5673, pages 171–187. Springer, 2009.
22. B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *ESOP'90*, LNCS, vol. 1694, pages 232–247. Springer, 1990.
23. V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.*, 30:141–155, 2009.