Interprocedural Analysis (Almost) for Free

Markus Müller-Olm *†
FernUniversität Hagen
LG Praktische Informatik 5
58084 Hagen, Germany

Helmut Seidl ‡
TU München
Lehrstuhl für Informatik II
80333 München, Germany

Bernhard Steffen §
Universität Dortmund
FB 4, LS V
44221 Dortmund, Germany

Abstract

We consider a procedural language with assignments and C-like functions with call-by-value parameters and single return values and present the first algorithm for precise inter-procedural value-numbering. The algorithm is precise in that it infers for all program points all inter-procedurally valid Herbrand equalities. Moreover, it is asymptotically no more expensive than the best known algorithm for the same intra-procedural problem. Instead of un-interpreted operator symbols in assignments, we also consider affine assignments and the problem of inferring all inter-procedurally valid affine relationships between program variables. For this problem, we obtain a new polynomial algorithm whose complexity matches the complexity of the intra-procedural analysis in [21] and thus improves on the corresponding inter-procedural algorithm in [20] by a factor of k^5 where k is the number of variables.

1 Introduction

Classically, inter-procedural analyses have invested much effort into dealing with global variables precisely. For example, the seminal paper by Sharir and Pnueli [27] about the functional and the call-string approach to inter-procedural analysis deals (as they say, for expositional reasons) with parameterless procedures and global variables only. Cousot and Cousot's fundamental work [7] on inter-procedural analysis does not deal with global variables explicitly but considers multiple call-by-value parameters together with multiple result parameters which easily allow to simulate effects onto globals. Essentially the same class of procedures has been considered by Knoop and Steffen [15]. The key idea in these papers for obtaining maximally precise information about procedures is to describe the effect of a procedure call by a transformation of the local and global (abstract) state. In comparison, corresponding intra-procedural analyses avoid the use of second-order objects like functions and compute on data-flow facts directly. Thus, intra-procedural analyses are typically much more efficient.

In this paper, we suggest a light-weight approach to inter-procedural analysis where we consider C-like functions only. By this we mean functions with local variables, call-by-value parameters and *single* return values. Thus we abandon global variables. This is still an adequate setting for certain practical programming scenarios. For example, in system programming in C or in the development of multi-threaded applications, global variables often must be considered as volatile meaning that their values potentially are changed by the environment. Volatile variables cannot be handled easily within a transformer-based approach to inter-procedural analysis.

Here we present two instances of the light-weight approach. First, we present an inter-procedural inference algorithm which determines, for every program point, all valid *Herbrand equalities*. In fact, our algorithm is the first complete inter-procedural analysis of Herbrand equalities. Moreover, its running time asymptotically coincides with that of the best intra-procedural algorithms for the same problem [29, 11]. Secondly, we present an inter-procedural inference algorithm which determines for every program point in an *affine* program all valid affine relations. This algorithm also has essentially the same asymptotic running time as the fastest known intra-procedural inference algorithm [21] and thus beats the corresponding inter-procedural algorithm [20] by a factor k^5 where k is the number of variables.

Analyses for finding definite equalities between variables or variables and expressions in a program have been used in program optimization since long. Knowledge about definite equalities can be exploited for per-

*On leave from Universität Dortmund, FB 4, LS 5, 44221 Dortmund, Germany.
†Email: mmo@ls5.informatik.uni-dortmund.de
‡Email: seidl@in.tum.de
§Email: steffen@ls5.cs.uni-dortmund.de

forming and enhancing powerful optimizing program transformations. Examples include constant propagation, common subexpression elimination, and branch elimination [12, 4, 8], partial redundancy elimination and loop-invariant code motion [24, 29, 14], and strength reduction [30].

Clearly, it is undecidable whether two variables always have the same value at a program point even without interpreting conditionals [22]. Therefore, analyses are bound to detect only a subset, i.e., a safe approximation, of all equivalences. Analyses based on Herbrand interpretation of operators consider two values equal only if they are constructed by the same operator applications. Such analyses are said to detect Herbrand equalities. Another approach is to abstract programs with affine programs by interpreting affine assignments such as $\mathbf{x}_3 := 2\mathbf{x}_1 - \mathbf{x}_2 + 42$ precisely while approximating more complex assignments conservatively as assigning any value. These analyses typically compute valid affine relationships between program variables.

The key technical result onto which our new inter-procedural analyses are based is that the abstract effect of a function call $\mathbf{x}_1 := f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ can either be described by an assignment of an unknown value to $\mathbf{x}_1$ or by a *conditional* assignment, i.e., a pair $(\phi, \mathbf{x}_1 := t)$ consisting of a precondition ϕ together with an assignment $\mathbf{x}_1 := t$. In the Herbrand case, ϕ is a conjunction of Herbrand equalities whereas in the affine case, ϕ is given by a vector space of affine relations. If the precondition is satisfied, the function call behaves like the assignment $\mathbf{x}_1 := t$, otherwise, like an assignment of an unknown value. The interesting observation is that in the Herbrand as well as in the affine case, this is both sound and complete. Technically, the conditional assignments for functions are determined by effective weakest precondition computations for a particular postcondition. More specifically, this postcondition takes the form $\mathbf{y} \doteq \mathbf{x}_1$ where $\mathbf{y}$ is a fresh variable and $\mathbf{x}_1$ is the variable that receives the return value of the function.

Related Work. The earliest work on detecting Herbrand equalities dates back to Cocke and Schwartz [5]. While Cocke and Schwartz aimed at finding equalities inside basic blocks, the technique has been generalized to arbitrary control-flow graphs with branching and loops by Kildall [13]. For historical reasons, Kildall's technique is known as *global value numbering*. In contrast to a number of algorithms focusing more on efficiency than on precision [22, 1, 4, 25, 8, 10], Steffen [28] and Steffen et al. [29] concentrate on maximal precision. They employ a variant of Kildall's algorithm using a compact representation of Herbrand equivalences in terms of *structured partition DAGs (SPDAGs)*. However, the representations of equalities still can be of exponential size in terms of the argument program [11]. Recently, Gulwani and Necula modified this algorithm to obtain a polynomial time algorithm by exploiting the fact that SPDAGs can be pruned, if only equalities of bounded size are searched for [11].

For affine programs, a first intra-procedural algorithm for inferring all valid affine relations has been presented by Karr [12]. In earlier work, we have generalized Karr's ideas to affine programs with procedures [20]. This analysis can handle both local and global variables and has a running time which is linear in the program size and polynomial in the number of variables of degree 8. Extensions and implementations of this algorithm have been provided by Reps et al. [23, 3]. A probabilistic variant of Karr's algorithm has been suggested by Gulwani and Necula [9]. Their algorithm determines affine relations modulo some prime number which hold with a certain probability. Another variant of Karr's intra-procedural algorithm has been presented in [21] which computes all valid affine relations in time $\mathcal{O}(n \cdot k^3)$. Let us finally mention that all this work abstracts conditional branching by non-deterministic choice. In fact, if equality guards are taken into account then both in the Herbrand and in the affine case, determining whether a specific equality holds at a program point becomes undecidable [17, 21]. Disequality constraints, however, can be dealt with intra-procedurally [17, 18]. No inter-procedural extensions could be designed by now.

The current paper is organized as follows. In Section 2 we introduce un-interpreted programs with C-like functions as the abstract model of programs for which our Herbrand analysis is complete. In Section 3 we collect basic facts about conjunctions of Herbrand equalities. In Section 4 we present the weakest precondition computation to determine the effects of function calls. In Section 5 we use this description of effects to extend an inference algorithm for intra-procedurally inferring all valid Herbrand equalities to deal with C-like functions as well. In Section 6 we then turn to affine programs with C-like functions, i.e., now we analyze affine assignments precisely. We show in Section 7 that as in the Herbrand case, the effects of function calls can be described by a suitable weakest precondition. In Section 8 we show how this observation can be used to speed up the inter-procedural inference algorithm from [20]. Finally, in Section 9 we summarize and describe further directions of research.

2 Herbrand Programs

We model programs by systems of non-deterministic flow graphs that can recursively call each other as in

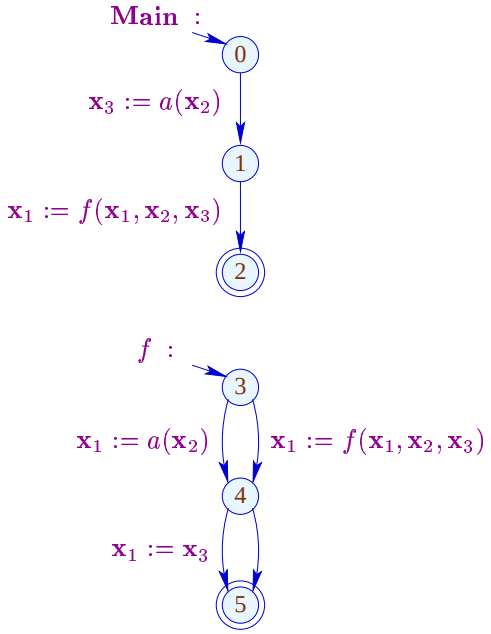

Figure 1: A small Herbrand program.

Figure 1. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of variables the program operates on. We assume that the basic statements in the program are either assignments of the form $\mathbf{x}_j := t$ for some expression t possibly involving variables from $\mathbf{X}$ or *nondeterministic* assignments $\mathbf{x}_j :=?$ and that branching in general is nondeterministic. Assignments $\mathbf{x}_j := \mathbf{x}_j$ have no effect onto the program state. They can be used as skip statements as, e.g., at the right edge from program point 4 to program point 5 in Figure 1 and also for the abstraction of guards. Non-deterministic assignments $\mathbf{x}_j :=?$ represent a safe abstraction of statements in a source program our analysis cannot handle precisely, for example input statements.

A *program* comprises a finite set Funct of *function names* that contains a distinguished function **Main**. Here, we consider C-like functions only with call-by-value parameters and return values. In this paper, we do not consider global variables. Without loss of generality, every call to a function f is of the form: $\mathbf{x}_1 := f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ — meaning that the values of all variables are passed to f as actual parameters, and that the variable $\mathbf{x}_1$ always receives the return value of f which is the final value $\mathbf{x}_1$ has received during the evaluation of f.[1] Note that more refined calling conventions of passing the values, e.g., by using designated argument variables or passing the values of arbitrary expressions into formal parameters can easily be reduced to our case. Due to our simple standard layout of calls, each call is uniquely represented by the name f of the called function. Let Stmt be the set of assignments and calls. Program execution starts with a call to **Main**. Each function name $f \in \mathsf{Funct}$ is associated with a *control flow graph* $G_f = (N_f, E_f, \mathsf{st}_f, \mathsf{ret}_f)$ that consists of:

- a set N_f of *program points*;
- a set of edges $E_f \subseteq N_f \times \mathsf{Stmt} \times N_f$;
- a special *entry (or start) point* $\mathsf{st}_f \in N_f$; and
- a special *return point* $\mathsf{ret}_f \in N_f$.

We assume that the program points of different functions are disjoint: $N_f \cap N_g = \emptyset$ for $f \neq g$. This can always be enforced by renaming program points. Moreover, we denote the set of edges labeled with assignments by Base and the set of edges calling function f by Call_f.

For describing program executions in presence of functions with local variables, it does not suffice to consider execution *paths*. Instead, we have to take the proper nesting of calls into account. Thus, we follow the approach taken, e.g., in [20] and represent executions as sequences r of (unranked) trees:

$$\begin{array}{lcll} \mathsf{rt} & ::= & \mathbf{x}_j := t \mid \mathsf{call}\langle r \rangle & \\ r & ::= & \mathsf{rt}_1; \ldots; \mathsf{rt}_n & (n \geq 0) \end{array}$$

Each individual tree rt in a sequence represents a base action or complete execution of a call. Let Runs denote the set of all runs. The set of runs *reaching program point* $u \in N$ can be characterized as the least solution of a system of subset constraints on run sets. For assignments we define: $\mathbf{S}(\mathbf{x}_j := t) = \{\mathbf{x}_j := t\}$. Similarly for nondeterministic assignments,

$$\mathbf{S}(\mathbf{x}_j :=?) = \{\mathbf{x}_j := c \mid c \in \mathcal{T}_\Omega\}\,.$$

The *same-level* runs of functions and program nodes are the smallest solution of the following constraint system $\mathbf{S}$:

$$\begin{array}{lllll} [\mathrm{S1}] & \mathbf{S}(f) & \supseteq \mathbf{S}(\mathsf{st}_f) & & \\ [\mathrm{S2}] & \mathbf{S}(\mathsf{ret}_f) & \supseteq \{\varepsilon\} & & \\ [\mathrm{S3}] & \mathbf{S}(u) & \supseteq \mathbf{S}(s)\,;\,\mathbf{S}(v) & & \text{if } (u,s,v) \in \mathsf{Base} \\ [\mathrm{S4}] & \mathbf{S}(u) & \supseteq \mathsf{call}\langle \mathbf{S}(f) \rangle\,;\,\mathbf{S}(v) & & \text{if } (u,f,v) \in \mathsf{Call}_f \end{array}$$

Note that, for convenience, the application of the constructor call to all sequences of a set S is denoted by $\mathsf{call}\langle S \rangle$. Also, we accumulate execution trees from the *rear*, i.e., starting from the return points. It is the constraint [S4] which deals with calls. If the ingoing edge (u, f, v) is a call to a function f, we concatenate the

[1] Alternatively, we could view the variable $\mathbf{x}_1$ as one global variable which serves as scratch pad for passing information from a called procedure back to its caller.

trees constructed from same-level runs of f by applying the constructor call with the same-level runs starting from v.

Using the sets of same-level runs of functions, we can characterize the runs *reaching* program points and functions as the smallest solution of the following constraint system $\mathbf{R}$:

$$\begin{array}{lll}
[\mathrm{R1}] & \mathbf{R}(\mathbf{Main}) \supseteq \{\varepsilon\} & \\
[\mathrm{R2}] & \mathbf{R}(f) \supseteq \mathbf{R}(u)\,, & \text{if } (u, f, _) \in \mathsf{Call} \\
[\mathrm{R3}] & \mathbf{R}(\mathsf{st}_f) \supseteq \mathbf{R}(f) & \\
[\mathrm{R4}] & \mathbf{R}(v) \supseteq \mathbf{R}(u)\,;\,\mathbf{S}(s)\,, & \text{if } (u, s, v) \in \mathsf{Base} \\
[\mathrm{R5}] & \mathbf{R}(v) \supseteq \mathbf{R}(u)\,;\,\mathsf{call}\langle\mathbf{S}(f)\rangle\,, & \text{if } (u, s, v) \in \mathsf{Call}
\end{array}$$

3 Conjunctions of Herbrand Equalities

We first consider the case where we maintain the structure of expressions but abstract from the concrete meaning of operators. This is also known as the *Herbrand* interpretation of terms. Let Ω denote a signature consisting of a set Ω_0 of constant symbols and sets $\Omega_r, r > 0$, of operator symbols of rank r which possibly may occur in right-hand sides or values. Let $\mathcal{T}_\Omega$ the set of all formal terms built up from Ω. For simplicity, we assume that the set Ω_0 is non-empty, and there is at least one operator. Note that under this assumption, the set $\mathcal{T}_\Omega$ is infinite. Let $\mathcal{T}_\Omega(\mathbf{X})$ denote the set of all terms with constants and operators from Ω which additionally may contain occurrences of variables from $\mathbf{X}$. Since we do not interpret constants and operators, a *state* assigning values to the variables is conveniently modeled by a mapping $\sigma : \mathbf{X} \to \mathcal{T}_\Omega$. Such mappings have also been called *ground substitutions*.

Every assignment statement $\mathbf{x}_j := t$ induces a transformation, $[\![\mathbf{x}_j := t]\!]_\mathcal{H}$, on a given program state σ given by:

$$[\![\mathbf{x}_j := t]\!]_\mathcal{H}\,\sigma = \sigma[\mathbf{x}_j \mapsto \sigma(t)]$$

Here $\sigma(t)$ is the term obtained from t by replacing each occurrence of a variable $\mathbf{x}_i$ by $\sigma(\mathbf{x}_i)$ and $\sigma[\mathbf{x}_j \mapsto t']$ is the ground substitution that maps $\mathbf{x}_j$ to $t' \in \mathcal{T}_\Omega$ and variables $\mathbf{x}_i \neq \mathbf{x}_j$ to $\sigma(\mathbf{x}_i)$.

The transformation $[\![\mathsf{call}\langle r\rangle]\!]_\mathcal{H}$ is more complicated. It must pass the values of the variables into the execution of the called function. Then it must update the value of the variable $\mathbf{x}_1$ of the state before the call. Thus, we define:

$$[\![\mathsf{call}\langle r\rangle]\!]_\mathcal{H}\,\sigma \;=\; \sigma[\mathbf{x}_1 \mapsto ([\![r]\!]_\mathcal{H}\;\sigma\;\mathbf{x}_1)]$$

Here we already use the straightforward inductive extension of $[\![\cdot]\!]_\mathcal{H}$ to runs which is defined by $[\![\varepsilon]\!]_\mathcal{H}\,\sigma = \sigma$ and $[\![r_1; r_2]\!]_\mathcal{H} = [\![r_2]\!]_\mathcal{H} \circ [\![r_1]\!]_\mathcal{H}$. The transformation $[\![r]\!]_\mathcal{H}$ can be described by a *substitution* $\tau_r : \mathbf{X} \to \mathcal{T}_\Omega(\mathbf{X})$ which maps the variables after the run to their values depending on the variables before the run:

$$\begin{array}{lcl}
\tau_\epsilon & = & \mathsf{Id} \\
\tau_{\mathbf{x}_j := t} & = & \{\mathbf{x}_j \mapsto t\} \\
\tau_{\mathsf{call}\langle r_1\rangle} & = & \{\mathbf{x}_1 \mapsto (\tau_{r_1}\,\mathbf{x}_1)\} \\
\tau_{r_1\,;\,r_2} & = & \tau_{r_2} \circ \tau_{r_1}
\end{array}$$

Here $\{\mathbf{x}_j \mapsto t\}$ is the substitution that maps $\mathbf{x}_j$ to t and, for $i \neq j$, $\mathbf{x}_i$ to itself; $\tau_2 \circ \tau_1$ is the substitution that maps $\mathbf{x}_i$ to $\tau_2(\tau_1(\mathbf{x}_i))$. We then have for every ground substitution σ:

$$[\![r]\!]_\mathcal{H}\,\sigma = \sigma \circ \tau_r$$

A substitution $\sigma : \mathbf{X} \to \mathcal{T}_\Omega(\mathbf{X})$ (possibly containing variables in the image terms) satisfies a conjunction of equations $\phi \;\equiv\; s_1 \doteq t_1 \wedge \ldots \wedge s_m \doteq t_m$ iff $\sigma(s_i) = \sigma(t_i)$ for $i = 1, \ldots, m$. We then also write $\sigma \models \phi$. In particular, ϕ is valid at a program point v iff it is valid for all states $[\![r]\!]_\mathcal{H}\;\sigma$ with $r \in \mathbf{R}[v]$, $\sigma : \mathbf{X} \to \mathcal{T}_\Omega$ a ground substitution.

Let us briefly recall some basic facts about conjunctions of equations. A conjunction ϕ is *satisfiable* iff $\sigma \models \phi$ for at least one σ. Otherwise, i.e., if ϕ is unsatisfiable, ϕ is logically equivalent to false. This value serves as the bottom value of our lattice. The greatest value is given by the *empty* conjunction which is always true and therefore also denoted by true. Whenever the conjunction ϕ is satisfiable, then there is a *most general* satisfying substitution σ, i.e., $\sigma \models \phi$ and for every other substitution τ satisfying ϕ, $\tau = \tau_1 \circ \sigma$ for some substitution τ_1. Such a substitution is often also called a *most general unifier* of ϕ. In particular, this means that the conjunction ϕ is equivalent to $\bigwedge_{\mathbf{x}_i \neq \sigma(\mathbf{x}_i)} \mathbf{x}_i \doteq \sigma(\mathbf{x}_i)$. Thus, every satisfiable conjunction of equations is equivalent to a (possibly empty) finite conjunction of equations $\mathbf{x}_{j_i} \doteq t_i$ where the left-hand sides $\mathbf{x}_{j_i}$ are distinct variables and none of the equations is of the form $\mathbf{x}_j \doteq \mathbf{x}_j$. Let us call such conjunctions *reduced*. The following fact is crucial for proving ternimation of our proposed fixpoint algorithms.

Proposition 1 *For every sequence $\phi_0 \Leftarrow \ldots \Leftarrow \phi_m$ of pairwise inequivalent conjunctions ϕ_j using k variables, $m \leq k + 1$.* □

Proposition 1 follows since for satisfiable reduced non-equivalent conjunctions ϕ_i, ϕ_{i+1}, $\phi_i \Leftarrow \phi_{i+1}$ implies that ϕ_i contains strictly more equations than ϕ_{i+1}.

In order to construct an abstract lattice of properties, we consider equivalence classes of conjunctions of equations which, however, will always be represented by one of their members. Let $\mathbb{E}(\mathbf{X}')$ denote the set of all (equivalence classes of) finite reduced conjunctions of

equations with variables from $\mathbf{X}'$. This set is partially ordered w.r.t. "$\Rightarrow$" (on the representatives) where the pairwise lower bound always exists and is given by conjunction "$\wedge$". Since by Proposition 1, all descending chains in this lattice are ultimately stable, not only finite but also infinite subsets $X \subseteq \mathbb{E}(\mathbf{X}')$ have a greatest lower bound. Hence, $\mathbb{E}(\mathbf{X}')$ is a *complete* lattice.

4 Weakest Preconditions

For reasoning about return values of functions, we introduce a fresh variable $\mathbf{y}$ and determine for every function f the weakest precondition $[\![f]\!]^{\mathrm{t}}_{\mathcal{H}}$ of the equation $\mathbf{y} \doteq \mathbf{x}_1$ w.r.t. f. We will work with the subset $\mathbb{E}_{\mathbf{y}}$ of $\mathbb{E}(\mathbf{X} \cup \{\mathbf{y}\})$ of (equivalence classes of) conjunctions of equalities with variables from $X \cup \{\mathbf{y}\}$ which are either equivalent to true or false or contain an equality $\mathbf{y} \doteq t$ for some $t \in \mathcal{T}_\Omega(\mathbf{X})$. Note that this subset is indeed closed under weakest preconditions for assignments and also under conjunction. In particular, it is a sublattice of $\mathbb{E}(X \cup \{\mathbf{y}\})$.

The weakest precondition $[\![\mathbf{x}_j := t]\!]^{\mathrm{t}}_{\mathcal{H}}\,\phi$ of a conjunction of equalities ϕ for an assignment $\mathbf{x}_j := t$ is given by the well-known rule:

$$[\![\mathbf{x}_j := t]\!]^{\mathrm{t}}_{\mathcal{H}}\phi = \phi[t/\mathbf{x}_j]$$

where $\phi[t/\mathbf{x}_j]$ denotes the formula obtained from ϕ by substituting t for $\mathbf{x}_j$.

In order to deal with calls, we introduce a binary operator $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}$. In the first argument, this operator takes a weakest precondition ϕ_1 of a function body for the equation $\mathbf{y} \doteq \mathbf{x}_1$. This formula should be in $\mathbb{E}_{\mathbf{y}}$ and thus is either equivalent to true, false or to a conjunction $\phi' \wedge (\mathbf{y} \doteq t)$ for some ϕ' and t not containing $\mathbf{y}$. The second argument of $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}$ is a postcondition ϕ_2 after the call. Since the weakest precondition of $\mathbf{y} \doteq \mathbf{x}_1$ is equivalent to true if and only if there is no executable program path, we set $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\mathsf{true}, \phi_2) = \mathsf{true}$. For ϕ_1 not equivalent to true we distinguish two cases. If ϕ_2 does not contain $\mathbf{x}_1$, we set $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_1, \phi_2) = \phi_2$. Finally, assume that ϕ_1 is not equivalent to true and ϕ_2 contains $\mathbf{x}_1$. If ϕ_1 is equivalent to false, then $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_1, \phi_2) = \mathsf{false}$. Otherwise, we can write ϕ_1 as $\phi' \wedge (\mathbf{y} \doteq t)$ where t and ϕ' do not contain $\mathbf{y}$ and define:

$$[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_1, \phi_2) \;=\; \phi' \wedge \phi_2[t/\mathbf{x}_1]$$

This definition is indeed independent of the chosen representation of ϕ_1. To see this, assume that ϕ_1 is also equivalent to $\phi'_1 \wedge (\mathbf{y} \doteq t_1)$ for some ϕ'_1, t_1 not containing $\mathbf{y}$. Then in particular, $\phi' \wedge (\mathbf{y} \doteq t)$ implies $\mathbf{y} \doteq t_1$ as well as ϕ'_1 from which we deduce that ϕ' also implies $t \doteq t_1$. Therefore, $\phi' \wedge \phi_2[t/\mathbf{x}_1]$ implies $\phi'_1 \wedge \phi_2[t_1/\mathbf{x}_1]$. By exchanging the roles of ϕ', t and ϕ'_1, t_1 we also find the reverse implication and the equivalence follows.

Next, we determine the weakest preconditions of runs. Sequential composition ";" is interpreted as composition of the individual transformers. Accordingly, we define:

$$[\![\varepsilon]\!]^{\mathrm{t}}_{\mathcal{H}}\phi = \phi \;\text{ and }\; [\![r_1; r_2]\!]^{\mathrm{t}}_{\mathcal{H}}\phi = [\![r_1]\!]^{\mathrm{t}}_{\mathcal{H}}\,([\![r_2]\!]^{\mathrm{t}}_{\mathcal{H}}\phi)$$

Applying the operator $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}$, we write:

$$[\![\mathsf{call}\langle r\rangle]\!]^{\mathrm{t}}_{\mathcal{H}}\phi \;=\; [\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}([\![r]\!]^{\mathrm{t}}_{\mathcal{H}}(\mathbf{y} \doteq \mathbf{x}_1), \phi)$$

The following proposition states that the transformation $[\![r]\!]^{\mathrm{t}}_{\mathcal{H}}$ indeed transforms a conjunction ϕ after the run r into the weakest precondition of ϕ before r.

Proposition 2 *If r is a run, then for every conjunction ϕ of equations (over $\mathbf{X} \cup \{\mathbf{y}\}$),*

1. *$\sigma \models [\![r]\!]^{\mathrm{t}}_{\mathcal{H}}\,\phi$ iff $[\![r]\!]_{\mathcal{H}}\,\sigma \models \phi$ for every substitution $\sigma : \mathbf{X} \cup \{\mathbf{y}\} \to \mathcal{T}_\Omega(\mathbf{X})$.*
2. *$[\![r]\!]^{\mathrm{t}}_{\mathcal{H}}\phi = \tau_r(\phi)$.*

Proof: The first assertion follows by induction over r. Since $\sigma \models \tau(\phi)$ iff $\sigma \circ \tau \models \phi$, the second assertion follows as $[\![r]\!]_{\mathcal{H}}$ is given by $[\![r]\!]_{\mathcal{H}}\,\sigma = \sigma \circ \tau_r$. □

In order to use weakest precondition transformations for precise program analyses, we establish the following distributivity properties:

Proposition 3 *1. For every assignment $\mathbf{x}_j := t$, $[\![\mathbf{x}_j := t]\!]^{\mathrm{t}}_{\mathcal{H}}$ preserves true and distributes over "$\wedge$".*

2. In each argument, the operation $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}$ preserves true and distributes over "$\wedge$".

Proof: The first assertion holds since substitutions preserve true and commute with "$\wedge$".

For the second assertion, the statement concerning the second argument of $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}$ is straightforwardly verified from the definition. The same is true for the preservation of true in the first argument. Hence, it remains to verify that

$$[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_1 \wedge \phi_2, \phi) = [\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_1, \phi) \wedge [\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_2, \phi)$$

If either ϕ_1 or ϕ_2 equal false, the assertion is obviously true. The same holds if either ϕ_1 or ϕ_2 equal true. Otherwise, we can assume w.l.o.g. that for $i = 1, 2$, ϕ_i is satisfiable, reduced and of the form: $\phi'_i \wedge (\mathbf{y} \doteq t_i)$ for some ϕ'_i not containing $\mathbf{y}$. If ϕ does not contain $\mathbf{x}_1$, the assertion is again trivially true. Therefore, we additionally may assume that ϕ contains at least one occurrence of $\mathbf{x}_1$. Then by definition, $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_i, \phi) = \phi'_i \wedge \phi[t_i/\mathbf{x}_1]$. Thus, we obtain:

$$\begin{aligned}
&[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_1, \phi) \wedge [\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{H}}(\phi_2, \phi)\\
&\quad= \;\phi'_1 \wedge \phi[t_1/\mathbf{x}_1] \wedge \phi'_2 \wedge \phi[t_2/\mathbf{x}_1]\\
&\quad= \;\phi'_1 \wedge \phi'_2 \wedge (t_1 \doteq t_2) \wedge \phi[t_1/\mathbf{x}_1]
\end{aligned}$$

since ϕ contains an occurrence of $\mathbf{x}_1$. On the other hand, we may also rewrite $\phi_1 \wedge \phi_2$ to: $\phi_1' \wedge \phi_2' \wedge (t_1 \doteq t_2) \wedge (\mathbf{y} \doteq t_1)$ where only the last equation contains $\mathbf{y}$. Therefore also:

$$[\![\mathsf{call}]\!]^{\mathsf{t}}_{\mathcal{H}}(\phi_1 \wedge \phi_2, \phi) = \phi_1' \wedge \phi_2' \wedge (t_1 \doteq t_2) \wedge \phi[t_1/\mathbf{x}_1]$$

which completes the proof. □

We introduce an abstraction function $\alpha^{\mathsf{t}}_{\mathcal{H}} : 2^{\mathsf{Runs}} \to \mathbb{E}_{\mathbf{y}}$ by:

$$\alpha^{\mathsf{t}}_{\mathcal{H}}(R) = \bigwedge_{r \in R} [\![r]\!]^{\mathsf{t}}_{\mathcal{H}}(\mathbf{y} \doteq \mathbf{x}_1)$$

By Propositions 2 and 3, the mapping $\alpha^{\mathsf{t}}_{\mathcal{H}}$ has the following properties:

1. $\alpha^{\mathsf{t}}_{\mathcal{H}}(\{\epsilon\}) = (\mathbf{y} \doteq \mathbf{x}_1)$;
2. $\alpha^{\mathsf{t}}_{\mathcal{H}}(\mathbf{x}_j := t\,;\, R) = [\![\mathbf{x}_j := t]\!]^{\mathsf{t}}_{\mathcal{H}}(\alpha^{\mathsf{t}}_{\mathcal{H}}(R))$;
3. $\alpha^{\mathsf{t}}_{\mathcal{H}}(\mathsf{call}\langle R_1\rangle\,;\, R_2) = [\![\mathsf{call}]\!]^{\mathsf{t}}_{\mathcal{H}}(\alpha^{\mathsf{t}}_{\mathcal{H}}(R_1), \alpha^{\mathsf{t}}_{\mathcal{H}}(R_2))$.

Based on these properties, we put up a constraint system for computing the weakest preconditions of functions by applying the abstraction function $\alpha^{\mathsf{t}}_{\mathcal{H}}$ to the corresponding system $\mathbf{S}$ for the same-level runs of functions and program nodes. In particular, we replace the set $\{\epsilon\}$ with $(\mathbf{y} \doteq \mathbf{x}_1)$; and precomposition of assignments or calls with the corresponding operators for weakest preconditions. Thus, we obtain the constraint system $\mathbf{WP}_{\mathcal{H}}$:

$$\begin{array}{lll}
[\mathbf{WP}_{\mathcal{H}}1] & \mathbf{WP}_{\mathcal{H}}(f) & \Rightarrow \mathbf{WP}_{\mathcal{H}}(\mathsf{st}_f) \\
[\mathbf{WP}_{\mathcal{H}}2] & \mathbf{WP}_{\mathcal{H}}(\mathsf{ret}_f) & \Rightarrow (\mathbf{y} \doteq \mathbf{x}_1) \\
[\mathbf{WP}_{\mathcal{H}}3] & \mathbf{WP}_{\mathcal{H}}(u) & \Rightarrow [\![s]\!]^{\mathsf{t}}_{\mathcal{H}}(\mathbf{WP}_{\mathcal{H}}(v))\,, \\
 & & \qquad \text{if } (u,s,v) \in \mathsf{Base} \\
[\mathbf{WP}_{\mathcal{H}}4] & \mathbf{WP}_{\mathcal{H}}(u) & \Rightarrow [\![\mathsf{call}]\!]^{\mathsf{t}}_{\mathcal{H}}(\mathbf{WP}_{\mathcal{H}}(f), \mathbf{WP}_{\mathcal{H}}(v))\,, \\
 & & \qquad \text{if } (u,f,v) \in \mathsf{Call}
\end{array}$$

where for $s \equiv \mathbf{x}_j :=?$, we have:

$$[\![s]\!]^{\mathsf{t}}_{\mathcal{H}}\phi = \bigwedge_{c \in \mathcal{T}_\Omega} \phi[c/\mathbf{x}_j]$$

By assumption, $\mathcal{T}_\Omega$ contains at least two elements $t_1 \neq t_2$. If ϕ contains $\mathbf{x}_j$, then $\phi[t_1/\mathbf{x}_j] \wedge \phi[t_2/\mathbf{x}_j]$ implies $t_1 \doteq t_2$ which is false by the choice of t_1, t_2. Therefore,

$$\begin{array}{rcl}
[\![s]\!]^{\mathsf{t}}_{\mathcal{H}}\phi & = & \begin{cases} \mathsf{false} & \text{if } \mathbf{x}_j \text{ occurs in } \phi \\ \phi & \text{otherwise} \end{cases} \\
 & = & \phi[t_1/\mathbf{x}_j] \wedge \phi[t_2/\mathbf{x}_j]
\end{array}$$

The last equality means that $\mathbf{x}_j :=?$ is semantically equivalent (w.r.t. weakest preconditions of Herbrand equalities) to the nondeterministic choice between the two assignments $\mathbf{x}_j := t_1$ and $\mathbf{x}_j := t_2$ and hence distributes over conjunctions as well.

By Knaster-Tarski fixpoint theorem, the constraint system $\mathbf{WP}_{\mathcal{H}}$ has a greatest solution which we again denote with $\mathbf{WP}_{\mathcal{H}}(f), \mathbf{WP}_{\mathcal{H}}(u)$, $f \in \mathsf{Funct}, u \in N$. From Propositions 2 and 3, we obtain by the Transfer Lemma from fixpoint theory (c.f., e.g., [2, 6]):

Theorem 1 *Assume p is a program of size n with k variables.*

1. *For every function f of p, $\mathbf{WP}_{\mathcal{H}}(f) = \bigwedge\{[\![r]\!]^{\mathsf{t}}_{\mathcal{H}}(\mathbf{y} \doteq \mathbf{x}_1) \mid r \in \mathbf{S}(f)\}$; and for every program point u of p, $\mathbf{WP}_{\mathcal{H}}(u) = \bigwedge\{[\![u]\!]^{\mathsf{t}}_{\mathcal{H}}(\mathbf{y} \doteq \mathbf{x}_1) \mid r \in \mathbf{S}(u)\}$.*
2. *The greatest solution of the constraint system $\mathbf{WP}_{\mathcal{H}}$ can be obtained with at most $(k+2) \cdot n$ evaluations of right-hand sides. The fixpoint iteration can be performed in time $\mathcal{O}(n \cdot k \cdot \Delta)$ where Δ is the maximal size of a dag representation of an occurring conjunction.* □

Note that each application of "$\wedge$" as well of any right-hand side in the constraint system $\mathbf{WP}_{\mathcal{H}}$ may at most double the sizes of dag representations of occurring conjunctions. Thus, the value Δ can be bounded by $2^{\mathcal{O}(n \cdot k)}$.

Example 1 *Consider the function f from Figure 1. First, f and every program point is initialized with the top element of the lattice $\mathbb{E}_{\mathbf{y}}$, i.e., with* true. *The first approximation of the weakest precondition at program point 4 for the postcondition $\mathbf{y} \doteq \mathbf{x}_1$ at 5, then is computed as:*

$$\begin{array}{rcl}
\mathbf{WP}_{\mathcal{H}}(4) & = & (\mathbf{y} \doteq \mathbf{x}_1) \wedge ([\![\mathbf{x}_1 := \mathbf{x}_3]\!]^{\mathsf{t}}_{\mathcal{H}}\,(\mathbf{y} \doteq \mathbf{x}_1) \\
 & = & (\mathbf{y} \doteq \mathbf{x}_1) \wedge (\mathbf{y} \doteq \mathbf{x}_3)
\end{array}$$

Accordingly, we obtain for the start point 3,

$$\begin{array}{rcl}
\mathbf{WP}_{\mathcal{H}}(3) & = & [\![\mathsf{call}]\!]^{\mathsf{t}}_{\mathcal{H}}(\mathsf{true}, \mathbf{WP}_{\mathcal{H}}(4)) \wedge \\
 & & \qquad ([\![\mathbf{x}_1 := a(\mathbf{x}_2)]\!]^{\mathsf{t}}_{\mathcal{H}}\,(\mathbf{WP}_{\mathcal{H}}(4))) \\
 & = & \mathsf{true} \wedge (\mathbf{y} \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3) \\
 & = & (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)
\end{array}$$

Thus, we obtain as a first approximation: $\mathbf{WP}_{\mathcal{H}}(f) = (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$. Since the fixpoint computation already stabilizes here, we have found that

$$[\![\mathbf{x}_1 := f(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)]\!]^{\mathsf{t}}_{\mathcal{H}}\,(\mathbf{y} \doteq \mathbf{x}_1) = (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$$

□

5 Inferring Herbrand Equalities

For computing weakest preconditions, we have relied on conjunctions of equations, (pre-) ordered by "$\Rightarrow$" where the greatest lower bound was implemented by the logical "$\wedge$". For *inferring* Herbrand equalities, we again may use conjunctions of equations, now over the set of variables $\mathbf{X}$ alone, i.e., we use $\mathbb{E} = \mathbb{E}(\mathbf{X})$ — but instead to greatest lower bounds, we now resort to least upper bounds. Conceptually, the least upper bound $\phi_1 \sqcup \phi_2$ of two elements in $\mathbb{E}$ corresponds to the best approximation of the disjunction $\phi_1 \vee \phi_2$. More precisely, it is the conjunction of all equations implied both by ϕ_1 and ϕ_2. Clearly, we can restrict ourselves to equations of the form $\mathbf{x}_i \doteq t$ ($\mathbf{x}_i \in \mathbf{X}, t \in \mathcal{T}_\Omega(\mathbf{X})$). Thus,

$$\begin{array}{rcl} \phi_1 \sqcup \phi_2 & = & \bigwedge\{\mathbf{x}_j \doteq t \mid (\phi_1 \vee \phi_2) \Rightarrow (\mathbf{x}_j \doteq t)\} \\ & = & \bigwedge\{\mathbf{x}_j \doteq t \mid (\phi_1 \Rightarrow (\mathbf{x}_j \doteq t)) \wedge (\phi_2 \Rightarrow (\mathbf{x}_j \doteq t))\} \end{array}$$

Consider, e.g.,

$$\begin{array}{rcl} \phi_1 & \equiv & (\mathbf{x}_1 \doteq g(a(\mathbf{x}_3))) \wedge (\mathbf{x}_2 \doteq a(\mathbf{x}_3)) \\ \phi_2 & \equiv & (\mathbf{x}_1 \doteq g(b)) \wedge (\mathbf{x}_2 \doteq b) \end{array}$$

Then $\phi_1 \sqcup \phi_2$ is equivalent to $\mathbf{x}_1 \doteq g(\mathbf{x}_2)$.

An efficient implementation of the operation "$\sqcup$" can, e.g., be obtained by using *partition dags* as in [29]. We recall from there that the least upper bound of two conjunctions with dag representations of sizes n_1, n_2 can be performed in time $\mathcal{O}(n_1 + n_2)$ resulting in (a dag representation of) a conjunction of size $\mathcal{O}(n_1 + n_2)$. We must provide abstract definitions for the abstract effects of assignments and calls. In particular, we define:

$$\begin{array}{rcl} [\![\mathbf{x}_j := t]\!]^\sharp_{\mathcal{H}}\phi & = & \bigwedge\{\psi \mid \phi \Rightarrow [\![\mathbf{x}_j := t]\!]^t_{\mathcal{H}}\psi\} \\ & = & \bigwedge\{\psi \mid \phi \Rightarrow \psi[t/\mathbf{x}_j]\} \\ [\![\mathbf{x}_j :=?]\!]^\sharp_{\mathcal{H}}\phi & = & \bigsqcup\{[\![\mathbf{x}_j := c]\!]^\sharp_{\mathcal{H}}\phi \mid c \in \mathcal{T}_\Omega\} \\ & = & \bigwedge\{\psi \text{ without } \mathbf{x}_j \mid \phi \Rightarrow \psi\} \end{array}$$

Thus, e.g.,

$$[\![\mathbf{x}_1 := \mathbf{x}_3]\!]^\sharp_{\mathcal{H}}\,(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \;=\; (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{x}_1 \doteq a(\mathbf{x}_2))$$

and:

$$[\![\mathbf{x}_3 :=?]\!]^\sharp_{\mathcal{H}}\,(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{x}_1 \doteq a(\mathbf{x}_2)) \;=\; (\mathbf{x}_1 \doteq a(\mathbf{x}_2))$$

Note that we have introduced here a specific operator $[\![\mathbf{x}_j :=?]\!]^\sharp_{\mathcal{H}}$ for nondeterministic assignments which is meant to abstract the effect of a nondeterministic choice between the assignments $\mathbf{x}_j := c, c \in \mathcal{T}_\Omega$. Also note that our definitions are effective. Efficient implementations again can be obtained by resorting to partition dags. From [29], we recall that the effect of an assignment $\mathbf{x}_j := t$ can be computed in time polynomial in the size n_1 of the argument and the size n_2 of (a dag representation of) t. Moreover, the dag representation of the result is again of size $\mathcal{O}(n_1 + n_2)$. A similar estimation also holds for nondeterministic assignments.

The crucial step in the derivation of our analysis is the construction of an abstract operator $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}$ for function calls. The first argument of this operator is meant to denote the weakest precondition ϕ_1 of $(\mathbf{y} \doteq \mathbf{x}_1)$ for a (possibly empty) set of runs through some function. The second argument ϕ_2 then is a conjunction of equations which are valid before the call.

If ϕ_1 is equivalent to true, the set of computations is empty. Therefore, we define: $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\mathsf{true}, \phi_2) = \mathsf{false}$ (everything is true after the call).

If ϕ_1 is equivalent to false, then nothing can be said about the value of $\mathbf{x}_1$ after the call. Therefore, we define: $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\mathsf{false}, \phi_2) = [\![\mathbf{x}_1 :=?]\!]^\sharp_{\mathcal{H}}\phi_2$. Otherwise, we assume that we can write ϕ_1 as $\phi' \wedge (\mathbf{y} \doteq t)$ where ϕ' and t do not contain $\mathbf{y}$. Then:

$$[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1, \phi_2) \;=\; \begin{cases} [\![\mathbf{x}_1 := t]\!]^\sharp_{\mathcal{H}}\phi_2 & \text{if} \quad \phi_2 \Rightarrow \phi' \\ [\![\mathbf{x}_1 :=?]\!]^\sharp_{\mathcal{H}}\phi_2 & \text{otherwise} \end{cases}$$

We can apply this definition to inductively define the abstract semantics of a single run r:

$$\begin{array}{rcl} [\![\mathsf{call}\langle r' \rangle]\!]^\sharp_{\mathcal{H}}\phi & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}([\![r']\!]^t_{\mathcal{H}}(\mathbf{y} \doteq \mathbf{x}_1), \phi) \\ [\![\epsilon]\!]^\sharp_{\mathcal{H}}\phi & = & \phi \\ [\![r_1\,;\,r_2]\!]^\sharp_{\mathcal{H}}\phi & = & [\![r_2]\!]^\sharp_{\mathcal{H}}([\![r_1]\!]^\sharp_{\mathcal{H}}\phi) \end{array}$$

We have:

Proposition 4 *For every run r and conjunction ϕ,*

1. $[\![r]\!]^\sharp_{\mathcal{H}}\phi = \bigwedge\{\psi \mid \phi \Rightarrow \tau_r(\psi)\}$;
2. $[\![r]\!]^\sharp_{\mathcal{H}}\phi \Rightarrow \psi$ *iff* $\phi \Rightarrow [\![r]\!]^t_{\mathcal{H}}\psi$ *for every conjunction* ψ. □

Whenever a substitution σ satisfies ϕ before the run r, then by Proposition 2, $[\![r]\!]_{\mathcal{H}}\,\sigma = \sigma \circ \tau_r$ will satisfy any equation ψ after the run whose weakest precondition is implied by ϕ. Thus by statement 2 of the proposition, σ satisfies also the conjunction $[\![r]\!]^\sharp_{\mathcal{H}}\,\phi$. Moreover, whenever an equation ψ is satisfied by all substitutions $[\![r]\!]_{\mathcal{H}}, \sigma = \sigma \circ \tau_r$, where $\sigma \models \phi$, then by Proposition 2, $\tau_r(\psi)$ is satisfied by all σ which also satisfy ϕ. Hence, $\phi \Rightarrow \tau_r(\psi)$ and therefore by statement 1, $[\![r]\!]^\sharp_{\mathcal{H}}\phi$ implies ψ. In conclusion, $[\![r]\!]^\sharp_{\mathcal{H}}\phi$ is precisely the conjunction of all equalities which are valid after the run r given that ϕ is valid before r.

For constructing a precise algorithm for inferring Herbrand equalities, we rely on the following key distributivity property:

Proposition 5 *1. $[\![\mathbf{x}_j := t]\!]^\sharp_{\mathcal{H}}$ and $[\![\mathbf{x}_j :=?]\!]^\sharp_{\mathcal{H}}$ preserve* false *and commute with "$\sqcup$".*

2. *In the first argument,* $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}$ *maps* true *to* false *and translates "$\wedge$"into "$\sqcup$", i.e.,*

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\mathsf{true},\phi) & = & \mathsf{false} \\ [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1 \wedge \phi_2,\phi) & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1,\phi) \sqcup [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_2,\phi) \end{array}$$

for $\phi_1, \phi_2 \in \mathbb{E}_{\mathbf{y}}$.

In the second argument, $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}$ *preserves* false *and commutes with "$\sqcup$", i.e.,*

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi,\mathsf{false}) & = & \mathsf{false} \\ [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi,\phi_1 \sqcup \phi_2) & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi,\phi_1) \sqcup [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi,\phi_2) \end{array}$$

Proof: Assertion 1 easily follows from the definitions. Therefore we only prove the statement 2 about the properties of $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}$. The assertion concerning the second argument easily follows from the assertions concerning assignments. The assertion about the transformation of true in the first argument follows from the definition. Therefore, it remains to consider a conjunction $\phi_1 \wedge \phi_2$ in the first argument of $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}$. We distinguish two cases.

Case 1: $\phi_1 \wedge \phi_2$ is not satisfiable, i.e., equivalent to false. Then $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1 \wedge \phi_2,\phi) = [\![\mathbf{x}_1 :=?]\!]^\sharp_{\mathcal{H}}\,\phi$. If any of the ϕ_i is also not satisfiable, then $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_i,\phi) = [\![\mathbf{x}_1 :=?]\!]^\sharp_{\mathcal{H}}\,\phi$ which subsumes the effect of any assignment $\mathbf{x}_1 := t$ onto ϕ, and the assertion follows. Therefore assume that both ϕ_1 and ϕ_2 are satisfiable. Each of them then can be written as $\phi_i' \wedge (\mathbf{y} \doteq t_i)$. If any of the ϕ_i' is not implied by ϕ, then again $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_i,\phi) = [\![\mathbf{x}_1 :=?]\!]^\sharp_{\mathcal{H}}\,\phi$ which subsumes the effect of the assignment $\mathbf{x}_1 := t_{2-i}$ onto ϕ. Therefore,

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1,\phi) \sqcup [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_2,\phi) & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_i,\phi) \\ & = & [\![\mathbf{x}_1 :=?]\!]_{\mathcal{H}}\,\phi \\ & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1 \wedge \phi_2,\phi) \end{array}$$

If on the other hand, both ϕ_i' are implied by ϕ, then $\phi_1' \wedge \phi_2'$ is satisfiable. Thus, $\sigma(t_1) \neq \sigma(t_2)$ for any $\sigma \models \phi_1' \wedge \phi_2'$. In particular, $t_1 \doteq t_2$ cannot be implied by ϕ. Since ϕ_i' is implied by ϕ, $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_i,\phi) = [\![\mathbf{x}_1 := t_i]\!]^\sharp_{\mathcal{H}}\,\phi$. On the other hand, for every ψ containing $\mathbf{x}_1$, it is impossible that both $\phi \Rightarrow \psi[t_1/\mathbf{x}_1]$ and $\phi \Rightarrow \psi[t_2/\mathbf{x}_1]$ holds. Therefore, the least upper bound of $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1,\phi)$ and $[\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_2,\phi)$ is given by the conjunction of all ψ implied by ϕ which do not contain $\mathbf{x}_1$. This conjunction precisely equals $[\![\mathbf{x}_1 :=?]\!]^\sharp_{\mathcal{H}}\;\phi = [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\mathsf{false},\phi)$, and the assertion follows.

Case 2: $\phi_1 \wedge \phi_2$ is satisfiable. Then also both of the ϕ_i are satisfiable and can be written as conjunctions $\phi_i' \wedge (\mathbf{y} \doteq t_i)$ for some ϕ_i' and t_i not containing $\mathbf{y}$. If ϕ does not imply $\phi_1' \wedge \phi_2'$, then both sides of the equation are equal to $[\![\mathbf{x}_1 :=?]\!]_{\mathcal{H}}\,\phi$ and nothing is to prove. Therefore, assume that $\phi \Rightarrow \phi_1' \wedge \phi_2'$. If ϕ also implies $t_1 \doteq t_2$, then for every ψ, $\phi \Rightarrow \psi[t_1/\mathbf{x}_1]$ iff $\phi \Rightarrow \psi[t_2/\mathbf{x}_1]$. Therefore in this case,

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_i,\phi) & = & [\![\mathbf{x}_1 := t_1]\!]^\sharp_{\mathcal{H}}\,\phi \\ & = & [\![\mathbf{x}_1 := t_2]\!]^\sharp_{\mathcal{H}}\,\phi \\ & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1 \wedge \phi_2,\phi) \end{array}$$

and the assertion follows. If ϕ does not imply $t_1 \doteq t_2$, the least upper bound of $[\![\mathbf{x}_1 := t_i]\!]^\sharp_{\mathcal{H}}\,\phi$ is the conjunction of all ψ not containing $\mathbf{x}_1$ which are implied by ϕ. This precisely equals:

$$\begin{array}{lcl} [\![\mathbf{x}_1 :=?]\!]^\sharp_{\mathcal{H}}\,\phi & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi' \wedge (\mathbf{y} \doteq t_1),\phi) \\ & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\phi_1 \wedge \phi_2,\phi) \end{array}$$

for $\phi' \equiv \phi_1' \wedge \phi_2' \wedge (t_1 \doteq t_2)$, and the assertion follows. □

Propositions 4 and 5 motivate the definition of an abstraction of (reaching) run sets by means of the function $\alpha_{\mathcal{H}} : 2^{\mathsf{Runs}} \to \mathbb{E}$ defined by:

$$\alpha_{\mathcal{H}}(R) = \bigsqcup\{[\![r]\!]^\sharp_{\mathcal{H}}\mathsf{true} \mid r \in R\}$$

Thus, $\alpha_{\mathcal{H}}(R)$ describes the maximal conjunction of equations guaranteed to hold after execution of runs in R — given that only the trivial assertion true holds at program start. In particular, if R is the set of runs reaching a program point v, then $\alpha_{\mathcal{H}}(R)$ precisely equals the conjunction of all equalities which are valid when reaching u. By construction, $\alpha_{\mathcal{H}}$ distributes over arbitrary unions and therefore is indeed an abstraction. Moreover by Proposition 4, we have:

1. $\alpha_{\mathcal{H}}(\{\epsilon\}) = \mathsf{true}$;
2. $\alpha_{\mathcal{H}}(R;\, \mathbf{x}_j := t) = [\![\mathbf{x}_j := t]\!]^\sharp_{\mathcal{H}}(\alpha_{\mathcal{H}}(R))$;
3. $\alpha_{\mathcal{H}}(R;\, \mathsf{call}\langle R'\rangle) = [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}([\![R']\!]^{\mathsf{t}}_{\mathcal{H}}(\mathbf{y} \doteq \mathbf{x}_1), \alpha_{\mathcal{H}}(R))$.

Applying the abstraction $\alpha_{\mathcal{H}}$ to the constraint system $\mathbf{R}$ of reaching runs, we obtain the constraint system $\mathbf{H}$:

$$\begin{array}{lll} [\mathbf{H}1] & \mathbf{H}(\mathbf{Main}) \Leftarrow \mathsf{true} & \\ [\mathbf{H}2] & \mathbf{H}(f) \Leftarrow \mathbf{H}(u)\,, & \text{if } (u,f,\_) \in \mathsf{Call} \\ [\mathbf{H}3] & \mathbf{H}(\mathsf{st}_f) \Leftarrow \mathbf{H}(f) & \\ [\mathbf{H}4] & \mathbf{H}(v) \Leftarrow [\![s]\!]^\sharp_{\mathcal{H}}(\mathbf{H}(u))\,, & \text{if } (u,s,v) \in \mathsf{Base} \\ [\mathbf{H}5] & \mathbf{H}(v) \Leftarrow [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\mathbf{WP}_{\mathcal{H}}(f),\mathbf{H}(u))\,, & \\ & & \text{if } (u,f,v) \in \mathsf{Call} \end{array}$$

Again by Knaster-Tarski fixpoint theorem, the constraint system $\mathbf{H}$ has a least solution which we again denote with $\mathbf{H}(f), \mathbf{H}(u)$, $f \in \mathsf{Funct}, u \in N$. We obtain the following theorem:

Theorem 2 *Assume p is a program of size n with k variables.*

1. *For every function* f, $\mathbf{H}[f] = \bigsqcup\{[\![r]\!]^\sharp_{\mathcal{H}}\mathsf{true} \mid r \in \mathbf{R}(f)\}$; *and for every program point* u, $\mathbf{H}[u] = \bigsqcup\{[\![u]\!]^\sharp_{\mathcal{H}}\mathsf{true} \mid r \in \mathbf{R}(u)\}$.

2. *Given the values* $\mathbf{WP}_{\mathcal{H}}(f)$, $f \in \mathsf{Funct}$, *the least solution of the constraint system* $\mathbf{H}$ *can be obtained with at most* $(k+1)\cdot n$ *evaluations of right-hand sides. The fixpoint iteration can be performed in time* $\mathcal{O}(n\cdot k\cdot\Delta)$ *where* Δ *is the maximal size of a dag representation of an occurring conjunction.*

By statement 1 of the theorem, we precisely compute for every program point u and for every function f, the conjunction of all equalities which are valid when reaching u and a call of f, respectively. Each application of "$\sqcup$" as well as of any right-hand side in the constraint system $\mathbf{H}$ may at most double the sizes of dag representations of occurring conjunctions. Together with the corresponding upper bound for the greatest solution of the constraint system $\mathbf{WP}_{\mathcal{H}}$, the value Δ therefore can be bounded by $2^{\mathcal{O}(n\cdot k)}$. Indeed, this upper bound is *tight* in that it matches the corresponding lower bound for the intra-procedural case [11].

Example 2 *We consider again the program from Figure 1. For the start point* 0 *of the main function* **Main**, *no non-trivial equation holds. Therefore,* $\mathbf{H}(0) = \mathsf{true}$. *For program point 1, we have:*

$$\begin{array}{rcl}\mathbf{H}(1) & \equiv & [\![\mathbf{x}_3 := a(\mathbf{x}_2)]\!]^\sharp_{\mathcal{H}}\mathsf{true}\\ & \equiv & \mathbf{x}_3 \doteq a(\mathbf{x}_2)\end{array}$$

In Section 4, we have computed the weakest precondition of $\mathbf{y} \doteq \mathbf{x}_1$ *for the function* f *as* $(\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{y} \doteq \mathbf{x}_3)$. *Since* $\mathbf{H}(1)$ *implies the equation* $\mathbf{x}_3 \doteq a(\mathbf{x}_2)$, *we obtain a representation of all equalities valid at program exit* 2 *by:*

$$\begin{array}{rcl}\mathbf{H}(2) & \equiv & [\![\mathsf{call}]\!]^\sharp_{\mathcal{H}}(\mathbf{WP}_{\mathcal{H}}(f), \mathbf{H}(1))\\ & \equiv & [\![\mathbf{x}_1 := \mathbf{x}_3]\!]^\sharp_{\mathcal{H}}(\mathbf{x}_3 \doteq a(\mathbf{x}_2))\\ & \equiv & (\mathbf{x}_3 \doteq a(\mathbf{x}_2)) \wedge (\mathbf{x}_1 \doteq a(\mathbf{x}_2))\end{array}$$

Thus at the return point of **Main** *both the equalities* $\mathbf{x}_3 \doteq a(\mathbf{x}_2)$ *and* $\mathbf{x}_1 \doteq a(\mathbf{x}_2)$ *hold.* □

6 Affine Relations

We now turn to *affine* programs. Thus, we now take meanings of operators into account — but restrict right-hand sides t of assignments to affine combinations. An example of such a program is shown in Figure 2. It is obtained from the example in Figure 1 by replacing all occurrences of the expression $a(\mathbf{x}_2)$ with $3\cdot\mathbf{x}_2 - 2$. Also for affine programs, we observe that the effect of a function can be described by means of the weakest precondition of the equation $\mathbf{y} \doteq \mathbf{x}_1$. Thus, we can use the

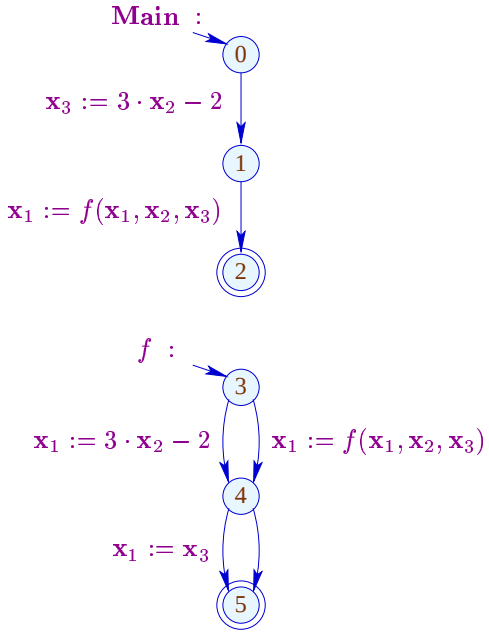

Figure 2: A small affine program.

same idea for analyzing affine programs as for Herbrand programs without global variables. In order to obtain a perfect match, we here only present in detail how Karr's original intra-procedural inference algorithm can be extended to affine programs with C-like functions. The corresponding extension of our improvement of Karr's algorithm in [21] which saves another factor k (k the number of variables) in the complexity is then explained through an example.

Here, we assume that the variables in the program take values in a field, namely the field $\mathbb{Q}$ of rational numbers. In an affine program, program states are given by mappings $\sigma : \mathbf{X} \to \mathbb{Q}$ which for convenience are denoted by vectors $x = (x_1, \ldots, x_k) \in \mathbb{Q}^k$. Each assignment statement $\mathbf{x}_j := t$, $t \equiv t_0 + t_1\mathbf{x}_1 + \ldots + t_k\mathbf{x}_k$, induces an *affine transformation* of the program state:

$$[\![\mathbf{x}_j := t]\!]_{\mathcal{A}}\, x = x[\mathbf{x}_j \mapsto (t_0 + t_1x_1 + \ldots + t_kx_k)]$$

where $x[\mathbf{x}_j \mapsto c]$ denotes the vector obtained from x by replacing the j-th component (corresponding to the value of $\mathbf{x}_j$) with c. The affine transformations of assignments can be extended to affine transformations $[\![r]\!]_{\mathcal{A}}$ for program runs r. In particular, the transformation $[\![\mathsf{call}\langle r\rangle]\!]_{\mathcal{A}}$ passes the values of the variables into the execution of r. Then it updates the value of the variable $\mathbf{x}_1$ of the state before execution of r with the result provided by r for $\mathbf{x}_1$. Thus, if $([\![r]\!]_{\mathcal{A}}\, x)_1$ denotes the first component of $[\![r]\!]_{\mathcal{A}}\, x$, then

$$[\![\mathsf{call}\langle r\rangle]\!]_{\mathcal{A}}\, x = x[\mathbf{x}_1 \mapsto ([\![r]\!]_{\mathcal{A}}\, x)_1]$$

7 Weakest Preconditions

We are interested in *affine relations*. An affine relation a is an equation $a_0 + a_1\mathbf{x}_1 + \ldots + a_k\mathbf{x}_k \doteq 0$ for $a_j \in \mathbb{Q}$. As usual, we say that a state $x = (x_1, \ldots, x_k) \in \mathbb{Q}^k$ satisfies a (denoted by $x \models a$) iff $a_0 + a_1x_1 + \ldots + a_kx_k = 0$ holds. Following [20, 21], we represent an affine relation by the vector $a = (a_0, \ldots, a_k)$ of its coefficients. As in Section 3, we want to determine for every function f the weakest precondition of the equation $\mathbf{y} \doteq \mathbf{x}_1$ for some fresh variable $\mathbf{y}$. In order to conveniently add the coefficient for $\mathbf{y}$ into vectors representing affine relations, we prefer to rename the variable $\mathbf{y}$ to $\mathbf{x}_{k+1}$. In particular, the relation $\mathbf{y} \doteq \mathbf{x}_1$ then is represented by the vector: $(0, 1, 0, \ldots, 0, -1)$.

We now introduce the required transformations for computing weakest preconditions.

For an assignment $\mathbf{x}_j := t$, $t \equiv t_0 + t_1\mathbf{x}_1 + \ldots t_k\mathbf{x}_k$, the weakest precondition of the affine relation a is given by: $[\![\mathbf{x}_j := t]\!]^{\mathrm{t}}_{\mathcal{A}} a = (a'_0, \ldots, a'_{k+1})$, where

$$a'_i = \begin{cases} a_i + a_j \cdot t_i & \text{if} \quad i \neq j \\ a_j \cdot t_j & \text{if} \quad i = j \\ a_{k+1} & \text{if} \quad i = k+1 \end{cases}$$

Note that the vector $a' = (a_0, \ldots, a_{k+1})$ represents the affine relation obtained by substituting t for $\mathbf{x}_j$ in the affine relation represented by a. As already observed in [20], this transformation is *linear*.

For calls, we introduce a binary operator $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{A}}$ defined by $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{A}}(b, a) = (a'_0, \ldots, a'_{k+1})$, where

$$a'_i = \begin{cases} b_i \cdot a_1 - b_{k+1} \cdot a_i & \text{if} \quad 1 \neq i \neq k+1 \\ b_1 \cdot a_1 & \text{if} \quad i = 1 \\ -b_{k+1} \cdot a_{k+1} & \text{if} \quad i = k+1 \end{cases}$$

Remark that this transformer is *multilinear*, i.e., linear in each of its arguments. To understand this definition, consider the special case where $b_{k+1} = -1$. Then the definition has the same effect as the assignment $\mathbf{x}_1 := b_0 + \sum_{i=1}^{k} b_i\mathbf{x}_i$. The general definition arises from this special case by appropriate scaling with the factor $-b_{k+1}$. As in the case of uninterpreted operators, we can combine these transformations to obtain, for every run r, a linear transformation $[\![r]\!]^{\mathrm{t}}_{\mathcal{A}}$. In the following, we consider the natural extension of the transformations $[\![r]\!]^{\mathrm{t}}_{\mathcal{A}}$ to subspaces of $\mathbb{Q}^{k+2}$. Recall that the set of subspaces of a finite-dimensional vector space V forms a complete lattice (w.r.t. the ordering set inclusion) where the least element is given by the 0-dimensional vector space consisting of the zero vector $\mathbf{0}$ only. The least upper bound of two spaces V_1, V_2 is given by:

$$\begin{aligned} V_1 \sqcup V_2 &= \mathbf{Span}(V_1 \cup V_2) \\ &= \{v_1 + v_2 \mid v_i \in V_i\}. \end{aligned}$$

Here, $\mathbf{Span}(X)$ denotes the sub-space generated by a subset $X \subseteq V$, i.e., the vector space of linear combinations of elements from X. Note that (w.r.t. the interpretation of vectors as affine relations), the least upper bound operation implements the logical *conjunction*, i.e., corresponds to the greatest lower bound operation for conjunctions. Accordingly, when we computed *greatest solutions* for determining weakest preconditions in Herbrand programs, we now compute *least solutions*. The height of the lattice, i.e., the maximal length of a strictly increasing chain, equals the dimension of V. Here, we will work with the subset $\mathbb{V}_{\mathbf{y}}$ of subspaces of $\mathbb{Q}^{k+2}$ which either are equal to the zero space $\{\mathbf{0}\}$ or contain at least one vector $(a_0, \ldots, a_{k+1})$ where $a_{k+1} \neq 0$. Note that this subset is indeed closed under "$\sqcup$" and therefore a complete sublattice of vector spaces.

Proposition 6 summarizes the properties analogous to those from the Propositions 2 and 3 for Herbrand programs. For convenience, we use the notation (x, x_{k+1}) to represent the vector obtained from x by appending the value x_{k+1} as additional component.

Proposition 6 *1. If r is a run, then for every state $x \in \mathbb{Q}^k$, value $x_{k+1} \in \mathbb{Q}$ and every affine relation $a \in \mathbb{Q}^{k+2}$, $(x, x_{k+1}) \models [\![r]\!]^{\mathrm{t}}_{\mathcal{A}} a$ iff $([\![r]\!]_{\mathcal{A}}\, x, x_{k+1}) \models a$.*

2. For every assignment $\mathbf{x}_j := t$, $[\![\mathbf{x}_j := t]\!]^{\mathrm{t}}_{\mathcal{A}}$ preserves the empty vector space $\{\mathbf{0}\}$ and commutes with binary "$\sqcup$".

3. In each argument, the operation $[\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{A}}$ preserves $\{\mathbf{0}\}$ and commutes with "$\sqcup$". □

Accordingly, we introduce an abstraction function $\alpha^{\mathrm{t}}_{\mathcal{A}} : 2^{\mathsf{Runs}} \to \mathbb{V}_{\mathbf{y}}$ by:

$$\alpha^{\mathrm{t}}_{\mathcal{A}}(R) = \mathbf{Span}(\{[\![r]\!]^{\mathrm{t}}_{\mathcal{A}}(0, 1, 0, \ldots, 0, -1) \mid r \in R\})$$

By Proposition 6, the mapping $\alpha^{\mathrm{t}}_{\mathcal{A}}$ has the following properties:

1. $\alpha^{\mathrm{t}}_{\mathcal{A}}(\{\epsilon\}) = \mathbf{Span}(\{(0, 1, 0, \ldots, 0, -1)\})$;
2. $\alpha^{\mathrm{t}}_{\mathcal{A}}(\mathbf{x}_j := t\,;\, R) = [\![\mathbf{x}_j := t]\!]^{\mathrm{t}}_{\mathcal{A}}(\alpha^{\mathrm{t}}_{\mathcal{A}}(R))$;
3. $\alpha^{\mathrm{t}}_{\mathcal{A}}(\mathsf{call}\langle R_1\rangle\,;\, R_2) = [\![\mathsf{call}]\!]^{\mathrm{t}}_{\mathcal{A}}(\alpha^{\mathrm{t}}_{\mathcal{A}}(R_1), \alpha^{\mathrm{t}}_{\mathcal{A}}(R_2))$.

Based on these properties, we put up a constraint system for computing the weakest preconditions for the functions in Funct by applying the abstraction function $\alpha^{\mathrm{t}}_{\mathcal{A}}$ to the constraint system $\mathbf{S}$ for the same-level runs of functions and program nodes. In particular, we replace the set $\{\epsilon\}$ with $\mathbf{Span}(\{(0, 1, 0, \ldots, 0, -1)\})$, and precomposition of assignments or calls with the corresponding operators for weakest preconditions. Thus, we

obtain the constraint system $\mathbf{WP}_\mathcal{A}$ over the complete lattice $\mathbb{V}_\mathbf{y}$:

$$\begin{array}{lll}
[\mathbf{WP}_\mathcal{A}1] & \mathbf{WP}_\mathcal{A}(f) & \supseteq \mathbf{WP}_\mathcal{A}(\mathsf{st}_f) \\
[\mathbf{WP}_\mathcal{A}2] & \mathbf{WP}_\mathcal{A}(\mathsf{ret}_f) & \supseteq \mathbf{Span}(\{(0,1,0,\ldots,0,-1)\}) \\
[\mathbf{WP}_\mathcal{A}3] & \mathbf{WP}_\mathcal{A}(u) & \supseteq [\![s]\!]^\mathsf{t}_\mathcal{A}(\mathbf{WP}_\mathcal{A}(v))\,, \\
 & & \qquad \text{if } (u,s,v) \in \mathsf{Base} \\
[\mathbf{WP}_\mathcal{A}4] & \mathbf{WP}_\mathcal{A}(u) & \supseteq [\![\mathsf{call}]\!]^\mathsf{t}_\mathcal{A}(\mathbf{WP}_\mathcal{A}(f), \mathbf{WP}_\mathcal{A}(v))\,, \\
 & & \qquad \text{if } (u,f,v) \in \mathsf{Call}
\end{array}$$

where for a nondeterministic assignment $s \equiv \mathbf{x}_j :=?$, we find as in [20] that it equivalently can be replaced with the nondeterministic choice between the assignments $\mathbf{x}_j := 0$ and $\mathbf{x}_j := 1$:

$$[\![\mathbf{x}_j :=?]\!]^\mathsf{t}_\mathcal{A} V \;=\; ([\![\mathbf{x}_j := 0]\!]^\mathsf{t}_\mathcal{A} V) \sqcup ([\![\mathbf{x}_j := 1]\!]^\mathsf{t}_\mathcal{A} V)$$

By Knaster-Tarski fixpoint theorem, the constraint system $\mathbf{WP}_\mathcal{A}$ has a least solution which we again denote with $\mathbf{WP}_\mathcal{A}(f), \mathbf{WP}_\mathcal{A}(u)$, $f \in \mathsf{Funct}, u \in N$. By the Transfer Lemma we obtain:

Theorem 3 *Assume p is a program of size n with k variables.*

1. *For every function f of p, $\mathbf{WP}_\mathcal{A}(f) = \bigsqcup\{[\![r]\!]^\mathsf{t}_\mathcal{A}(\mathbf{Span}(\{(0,1,0,\ldots,0,-1)\})) \mid r \in \mathbf{S}(f)\}$; and for every program point u of p, $\mathbf{WP}_\mathcal{A}(u) = \bigsqcup\{[\![u]\!]^\mathsf{t}_\mathcal{A}(\mathbf{Span}(\{(0,1,0,\ldots,0,-1)\})) \mid r \in \mathbf{S}(u)\}$.*
2. *The least solution of the constraint system $\mathbf{WP}_\mathcal{A}$ can be obtained with at most $(k+2)\cdot n$ evaluations of right-hand sides. Using semi-naive iteration, the least fixpoint can be computed in time $\mathcal{O}(n \cdot k^3)$.* □

In the complexity estimation, we here have assumed that arithmetic operations can be performed in constant time.

Example 3 *Let us consider the function f from the program of Figure 2. We first initialize all fixpoint variables with the zero vector space (representing the precondition* true*). Then the first approximation of $\mathbf{WP}_\mathcal{A}(4)$ for program point 4 is given by:*

$$\mathbf{WP}_\mathcal{A}(4) \;=\; \mathbf{Span}(\{(0,1,0,0,-1),(0,0,0,1,-1)\})$$

Accordingly, we obtain for program point 3:

$$\begin{array}{rcl}
\mathbf{WP}_\mathcal{A}(4) & = & [\![\mathsf{call}]\!]^\mathsf{t}_\mathcal{A}(\{\mathbf{0}\}, \mathbf{WP}_\mathcal{A}(4)) \;\sqcup \\
 & & \qquad [\![\mathbf{x}_1 := 3\cdot\mathbf{x}_2 - 2]\!]^\mathsf{t}_\mathcal{A}(\mathbf{WP}_\mathcal{A}(4)) \\
 & = & \mathbf{Span}(\{\mathbf{0}\} \cup \{(-2,0,3,0,-1),(0,0,0,1,-1)\}) \\
 & = & \mathbf{Span}(\{(-2,0,3,0,-1),(0,0,0,1,-1)\})
\end{array}$$

Since these are already the final values, we obtain for the function f:

$$[\![\mathbf{x}_1 := f(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3)]\!]^\mathsf{t}_\mathcal{A}(0,1,0,0,-1) \;= \\ \mathbf{Span}(\{(-2,0,3,-1,0),(0,0,0,1,-1)\})$$

□

8 Inferring Affine Relationships

In order to obtain an inter-procedural algorithm for inferring all affine relationships, we now extend Karr's intra-procedural analysis [12]. In Karr's approach, the set of affine relations valid at a program point is represented by a vector space $V \subseteq \mathbb{Q}^{k+1}$. Let $\mathbb{V}$ denote the complete lattice of all these subspaces. Since we are interested in definitely valid relations, the ordering in subspaces now is "$\supseteq$" where the greatest lower bound of two vector spaces $V_1, , V_2 \subseteq \mathbb{Q}^{k+1}$ is given by their intersection $V_1 \cap V_2$. Recall from linear algebra that, given the bases B_i of V_i, a basis for the intersection can be determined in time $\mathcal{O}(k^3)$.

Again, we must provide abstract definitions for the abstract effects of assignments and calls on vector spaces of affine relations. For an assignment $\mathbf{x}_j := t$, we define:

$$\begin{array}{rcl}
[\![\mathbf{x}_j := t]\!]^\sharp_\mathcal{A} V & = & \{a \in \mathbb{Q}^{k+1} \mid [\![\mathbf{x}_j := t]\!]^\mathsf{t}_\mathcal{A}\, a \in V\} \\
 & = & ([\![\mathbf{x}_j := t]\!]^\mathsf{t}_\mathcal{A})^{-1} V
\end{array}$$

Since $[\![\mathbf{x}_j := t]\!]^\mathsf{t}_\mathcal{A}$ is a linear transformation of vector spaces, also the inverse image of $[\![\mathbf{x}_j := t]\!]^\mathsf{t}_\mathcal{A}$ maps vector spaces of affine relations to vector spaces. Thus for example,

$$\begin{array}{l}
[\![\mathbf{x}_1 := \mathbf{x}_3]\!]^\sharp_\mathcal{A}(\mathbf{Span}(\{(-2,0,3,-1)\})) \\
\qquad = \;\; \mathbf{Span}(\{(-2,0,3,-1),(-2,-1,3,0)\})
\end{array}$$

For convenience, we also introduce the transformation of nondeterministic assignments:

$$\begin{array}{rcl}
[\![\mathbf{x}_j :=?]\!]^\sharp_\mathcal{A} V & = & \{(a_0,\ldots,a_k) \in V \mid a_j = 0\} \\
 & = & V \cap (\mathbb{Q}^j \times \{0\} \times \mathbb{Q}^{k-j})
\end{array}$$

Thus, the effect of $\mathbf{x}_j :=?$ amounts to restricting the set of argument relations to those not depending on the variable $\mathbf{x}_j$. For example,

$$\begin{array}{l}
[\![\mathbf{x}_3 :=?]\!]_\mathcal{A}(\mathbf{Span}(\{(-2,0,3,-1),(-2,-1,3,0)\})) \\
\qquad = \;\; \mathbf{Span}(\{(0,1,0,-1)\})
\end{array}$$

Karr shows in [12] that the effect of assignments can be computed in time $\mathcal{O}(k^2)$ given a basis of the argument vector space. The same holds true also for nondeterministic assignments. The crucial new point is the construction of an abstract operator $[\![\mathsf{call}]\!]^\sharp_\mathcal{A}$ for function calls. As its first argument, this operator takes a vector space representing the weakest precondition of the affine relation $(0,1,0,\ldots,0,-1)$ whereas the second argument is the vector space of relations which hold when reaching the program point before the call. For defining $[\![\mathsf{call}]\!]^\sharp_\mathcal{A}(V_1,V_2)$, we distinguish several cases. First, we set $[\![\mathsf{call}]\!]^\sharp_\mathcal{A}(\mathbf{0},V_2) = \mathbb{Q}^{k+1}$ for all V_2. Since the precondition $\mathbf{0}$ in the first argument corresponds to the empty

set of program executions, this definition means that everything is true at an unreachable program point. Now assume $V_1 \neq \{\mathbf{0}\}$. Under this assumption, V_1 can be written as $V_1 = \mathbf{Span}(G \cup \{(a'_0, \ldots, a'_k, -1)\})$ where $a_{k+1} = 0$ for all $(a_0, \ldots, a_{k+1}) \in G$. Then we define:

$$[\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1, V_2) = \begin{cases} [\![\mathbf{x}_1 := t]\!]^\sharp_{\mathcal{A}} V_2 & \text{if} \quad G \subseteq V_2 \\ [\![\mathbf{x}_j := ?]\!]^\sharp_{\mathcal{A}} V_2 & \text{otherwise} \end{cases}$$

where $t \equiv a'_0 + a'_1\mathbf{x}_1 + \ldots + a'_k\mathbf{x}_k$. We can use these rules to inductively define the abstract semantics of a single run r:

$$\begin{array}{lcl} [\![\mathsf{call}\langle r' \rangle]\!]^\sharp_{\mathcal{A}} V & = & \\ \qquad [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}([\![r']\!]^{\mathrm{t}}_{\mathcal{A}}(\mathbf{Span}(\{(0,1,0,\ldots,0,-1)\}), V) & & \\ [\![\epsilon]\!]^\sharp_{\mathcal{A}} V & = & V \\ [\![r_1\,;\, r_2]\!]^\sharp_{\mathcal{A}} V & = & [\![r_2]\!]^\sharp_{\mathcal{A}}([\![r_1]\!]^\sharp_{\mathcal{A}} V) \end{array}$$

We have:

Proposition 7 *For every run r and linear space $V \subseteq \mathbb{Q}^{k+1}$ of affine relations and $a \in \mathbb{Q}^{k+1}$, $a \in [\![r]\!]^\sharp_{\mathcal{A}} V$ iff $[\![r]\!]^{\mathrm{t}}_{\mathcal{A}}(a, 0) \in V \times \{0\}$.* □

Note that to the right of the equivalence stated in Proposition 7, we have tagged the affine relation a with an extra component 0 corresponding to the coefficient of the variable $\mathbf{x}_{k+1}$. This is necessary since in Section 7, we have defined formally only the weakest precondition of relations in $k+1$ variables, i.e., the k program variables plus the additional variable for the return value (which is no longer needed here). Analogous to Proposition 4, Proposition 7 allows us to conclude that $[\![r]\!]^\sharp_{\mathcal{A}} V$ precisely consists of all affine relations which are valid for states after the run iff the corresponding states before the run satisfied the relations in V. The following proposition is the analogue of Proposition 3 for Herbrand programs.

Proposition 8 *1. $[\![\mathbf{x}_j := t]\!]^\sharp_{\mathcal{A}}$ and $[\![\mathbf{x}_j := ?]\!]^\sharp_{\mathcal{A}}$ preserve $\{\mathbf{0}\}$ and commute with "$\sqcap$".*

2. In the first argument, $[\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}$ maps $\{\mathbf{0}\}$ to $\mathbb{Q}^{k+1}$ and translates "$\sqcup$" into "$\sqcap$", i.e.,

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(\{\mathbf{0}\}, V) & = & \mathbb{Q}^{k+1} \\ [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1 \sqcup V_2, V) & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1, V) \cap [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_2, V) \end{array}$$

for $V_1, V_2 \in \mathbb{V}_{\mathbf{y}}$.

In the second argument, $[\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}$ preserves $\{\mathbf{0}\}$ and commutes with "$\sqcap$", i.e.,

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(\{\mathbf{0}\}, V) & = & \{\mathbf{0}\} \\ [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V, V_1 \cap V_2) & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V, V_1) \cap [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V, V_2) \end{array}$$

Proof: We only prove the assertion about the distributivity in the first argument of $[\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}$ from the second statement. The assertion about the transformation of the zero vector space into $\mathbb{Q}^{k+1}$ follows from the definition. Therefore, consider a sum $V_1 \sqcup V_2$ of two vector spaces V_i from $\mathbb{V}_{\mathbf{y}}$. Clearly, the assertion is true if any of the V_i is the zero vector space. Therefore, assume that both $V_i \neq \{\mathbf{0}\}$. Then for each i, V_i contains a vector $a^{(i)}$ whose $(k+1)$-th component is nonzero and therefore (w.l.o.g.) equal to -1, and let G_i denote a (finite) set of vectors whose $(k+1)$-th components are all 0 such that $V_i = \mathbf{Span}(G_i \cup \{a^{(i)}\})$. Assume $a^{(i)} = (t^{(i)}_0, \ldots, t^{(i)}_k, -1)$. Then in particular, $V_1 \sqcup V_2 = \mathbf{Span}(G \cup \{a^{(1)}\})$ for $G = G_1 \cup G_2 \cup \{a^{(2)} - a^{(1)}\}$. We distinguish two cases.
Case 1: $G_1 \cup G_2 \not\subseteq V$. Then for some i, $G_i \not\subseteq V$. Since $[\![\mathbf{x}_i := ?]\!]^\sharp_{\mathcal{A}} V$ is included in $[\![\mathbf{x}_i := t]\!]^\sharp_{\mathcal{A}} V$ for any t, we have:

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_i, V) & = & [\![\mathbf{x}_i := ?]\!]_{\mathcal{A}}\, V \\ & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1, V) \cap [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_2, V) \end{array}$$

which equals $[\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1 \sqcup V_2, V)$ since G is not subsumed by V as well.
Case 2: $G_1 \cup G_2 \subseteq V$. Then for $i = 1, 2$,

$$[\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_i, V) = [\![\mathbf{x}_1 := t_i]\!]^\sharp_{\mathcal{A}} V$$

for $t_i \equiv t^{(i)}_0 + t^{(i)}_1\mathbf{x}_1 + \ldots + t^{(i)}_k\mathbf{x}_k$. If also $t^{(2)} - t^{(1)} \in V$, then

$$\begin{array}{lcl} [\![\mathbf{x}_1 := t_1]\!]_{\mathcal{A}}\, V & = & [\![\mathbf{x}_1 := t_2]\!]_{\mathcal{A}}\, V \\ & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1 \sqcup V_2, V) \end{array}$$

and the assertion follows. If on the other hand, $t^{(2)} - t^{(1)} \notin V$, then the intersection $[\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1, V) \cap [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_2, V)$ can only consist of vectors $(a_0, \ldots, a_k) \in V$ where $a_1 = 0$. Note that here we rely on the fact that $b_0 + b_1x_1 = 0$ and $b_0 + b_1x_2 = 0$ for $x_1 \neq x_2$ implies that $b_0 = 0$ and $b_1 = 0$.

Summarizing, we obtain:

$$\begin{array}{lcl} [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1, V) \cap [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_2, V) & = & V \cap (\mathbb{Q} \times \{0\} \times \mathbb{Q}^{k-1}) \\ & = & [\![\mathbf{x}_1 := ?]\!]^\sharp_{\mathcal{A}}\, V \\ & = & [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(V_1 \sqcup V_2, V) \end{array}$$

This completes the proof. □

Propositions 7 and 8 allow us to define an abstraction function $\alpha_{\mathcal{A}} : 2^{\mathsf{Runs}} \to \mathbb{V}$ by:

$$\alpha_{\mathcal{A}}(R) = \bigcap \{[\![r]\!]^\sharp_{\mathcal{A}}\{\mathbf{0}\} \mid r \in R\}$$

In particular, if R is the set of runs reaching a program point v, then $\alpha_{\mathcal{A}}(R)$ precisely equals the set of all affine relations valid at u. By construction, $\alpha_{\mathcal{A}}$ distributes over arbitrary unions and therefore is indeed an abstraction. By Proposition 7, we have:

1. $\alpha_{\mathcal{A}}(\{\epsilon\}) = \{\mathbf{0}\}$;
2. $\alpha_{\mathcal{A}}(R\,;\, \mathbf{x}_j := t) = [\![\mathbf{x}_j := t]\!]^\sharp_{\mathcal{A}}(\alpha_{\mathcal{A}}(R))$;
3. $\alpha_{\mathcal{A}}(R\,;\, \mathsf{call}\langle R'\rangle) \;=\; [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}([\![R']\!]^{\mathrm{t}}_{\mathcal{A}}(\mathbf{Span}(\{(0,1,0,\ldots,0,-1)\})), \alpha_{\mathcal{A}}(R))$.

Applying the abstraction $\alpha_{\mathcal{A}}$ to the constraint system $\mathbf{R}$ of reaching runs, we obtain the constraint system $\mathbf{A}$:

$$
\begin{array}{lll}
[\mathbf{A}1] & \mathbf{A}(\mathbf{Main}) \subseteq \{\mathbf{0}\} & \\
[\mathbf{A}2] & \mathbf{A}(f) \subseteq \mathbf{A}(u)\,, & \text{if } (u, f, \_) \in \mathsf{Call} \\
[\mathbf{A}3] & \mathbf{A}(\mathsf{st}_f) \subseteq \mathbf{A}(f) & \\
[\mathbf{A}4] & \mathbf{A}(v) \subseteq [\![s]\!]^\sharp_{\mathcal{A}}(\mathbf{A}(u))\,, & \text{if } (u, s, v) \in \mathsf{Base} \\
[\mathbf{A}5] & \mathbf{A}(v) \subseteq [\![\mathsf{call}]\!]^\sharp_{\mathcal{A}}(\mathbf{WP}_{\mathcal{A}}(f), \mathbf{A}(u))\,, & \\
 & & \text{if } (u, f, v) \in \mathsf{Call}
\end{array}
$$

Again by Knaster-Tarski fixpoint theorem, the constraint system $\mathbf{A}$ has a greatest solution which we denote by $\mathbf{A}(f), \mathbf{A}(u)$, $f \in \mathsf{Funct}, u \in N$. Summarizing, we obtain the following theorem:

Theorem 4 *Assume p is an affine program with C-link functions of size n with k variables.*

1. *For every function f of p, $\mathbf{A}(f) = \bigcap\{[\![r]\!]^\sharp_{\mathcal{A}}\{\mathbf{0}\} \mid r \in \mathbf{R}(f)\}$; and for every program point u of p, $\mathbf{A}(u) = \bigcap\{[\![r]\!]^\sharp_{\mathcal{A}}\{\mathbf{0}\} \mid r \in \mathbf{R}(u)\}$.*
2. *Given the values $\mathbf{WP}_{\mathcal{A}}(f)$, $f \in \mathsf{Funct}$, the greatest solution of the constraint system $\mathbf{A}$ can be obtained with at most $(k+1)\cdot n$ evaluations of right-hand sides in time $\mathcal{O}(n \cdot k^4)$.* □

Thus, we can determine for every program point u and function f, the vector space of all affine relations valid when reaching u and a call of f, respectively. For the complexity estimation of Theorem 4 we argue as follows: Firstly by Theorem 3, the values $\mathbf{WP}_{\mathcal{A}}(f), f \in \mathsf{Funct}$, can be computed in time $\mathcal{O}(n \cdot k^3)$; given these, the fixpoint iteration for computing the least solution of constraint system $\mathbf{A}$ amounts to $(k+1)\cdot n$ evaluations of right-hand sides and intersections — each of which can be done in time $\mathcal{O}(k^3)$.

Example 4 *Consider again the example program from Figure 2. At the start node of the function* **Main**, *no affine relation is valid. Therefore,* $\mathbf{A}(0) = \{\mathbf{0}\}$. *For program point 1, we obtain:*

$$
\begin{array}{rcl}
\mathbf{A}(1) & = & [\![\mathbf{x}_3 := 3\cdot\mathbf{x}_2 - 2]\!]^\sharp_{\mathcal{A}}\{\mathbf{0}\} \\
 & = & \mathbf{Span}(\{(-2,0,3,-1)\})
\end{array}
$$

Thus, using the weakest precondition of a call to function f as computed in Example 3, we obtain for program point 2:

$$
\begin{array}{rcl}
\mathbf{A}(2) & = & [\![\mathbf{x}_1 := \mathbf{x}_3]\!]^\sharp_{\mathcal{A}}(\mathbf{A}(1)) \\
 & = & \mathbf{Span}(\{(-2,0,3,-1),(-2,-1,3,0)\})
\end{array}
$$

where the first equality holds since the precondition $(-2,0,3,-1)$ is satisfied in $\mathbf{A}(1)$. Accordingly, we find that at program point 2, $-2 + 3\cdot\mathbf{x}_2 - \mathbf{x}_3 \doteq 0$ and $-2 - \mathbf{x}_1 + 3\cdot\mathbf{x}_2 \doteq 0$ are valid. □

The fixpoint algorithm which we have presented has the same running time as the intra-procedural inference algorithm of Karr [12]. We can improve on this by splitting the forward propagating second phase into two stages. In the first stage, we only determine the *affine hull* of the states reaching program points u or calls of functions f. In the second stage then the vector spaces of valid affine relations are obtained by computing the *duals* of these affine spaces. The latter can obviously be done in time $\mathcal{O}(n \cdot k^3)$ by solving appropriate systems of linear equations. Intra-procedurally, this approach has been suggested in [21] where it is also shown that intra-procedurally computing the affine hull of reachable states is amenable to a semi-naive fixpoint iteration strategy and therefore can be done in time $\mathcal{O}(n\cdot k^3)$ as well.

Example 5 *Consider again the example program from Figure 2. For computing the affine hulls of reachable state sets, we initialize the corresponding sets for all program points and functions with the empty set. Then we start the fixpoint computation by considering the start point of* **Main**, *here program point 0. At the start point every state is potentially reachable. A finite set of generators for this universal affine space is given by the state $(0,0,0)^{\mathrm{t}}$ together with the states $(1,0,0)^{\mathrm{t}}, (0,1,0)^{\mathrm{t}}$ and $(0,0,1)$. Applying the assignment $\mathbf{x}_3 := 3\cdot\mathbf{x}_2 - 2$ to this set of generators, we obtain for program point 1, a set of generators consisting of the states $(0,0,-2),(1,0,-2)$ and $(0,1,1)$. All three states satisfy the precondition $-2 + 3\cdot\mathbf{x}_2 - \mathbf{x}_3 \doteq 0$ for the function f. Therefore, f behaves like the assignment $\mathbf{x}_1 := \mathbf{x}_3$ — giving us the two states $(-2,0,-2)$ and $(1,1,1)$ as set of generators for the affine hull of the reachability set for program point 2. So, the set of all affine relations valid at program point 2 are obtained as the set of solution of the following homogeneous system of equations:*

$$
\begin{array}{rcl}
\mathbf{a}_0 + \mathbf{a}_1\cdot(-2) + \mathbf{a}_3\cdot(-2) & = & 0 \\
\mathbf{a}_0 + \mathbf{a}_1 + \mathbf{a}_2 + \mathbf{a}_3 & = & 0
\end{array}
$$

The vector space of solutions of this system is indeed spanned by the two vectors $(-2,0,3,-1)^{\mathrm{t}}$ and $(-2,-1,3,0)^{\mathrm{t}}$ — the same which we have found in our example calculation 4. □

Thus, by combining the weakest precondition approach to describing the effects of function calls with the semi-naive strategy for computing the affine hulls of reachable state set, we finally obtain:

Theorem 5 *Assume p is an affine program with C-like functions of size n with k variables. Then the sets of all affine relations valid at program points and calls of functions can be computed in time $\mathcal{O}(n \cdot k^3)$.* □

The inter-procedural analysis of integer arithmetic which we have presented here assumes infinite precision arithmetic. In particular, it relies on the embedding of integers into the field $\mathbb{Q}$ of rationals. This idealized assumption clearly does not always mirror the integer arithmetic provided by the programming language in question. In languages like C or Java, integers have a fixed bit length and arithmetic on them is modulo some power of 2. This implies that the arithmetic domain may have zero divisors and thus can no longer be embedded into a field. In a very recent paper we show how nonetheless the methods for integers can be adapted to work for modular arithmetic as well [19]. Indeed, this also holds for the algorithm of Theorem 5.

9 Conclusion

We have presented an inter-procedural algorithm for inferring all valid Herbrand equalities that is both precise and efficient. The key observation has been that the effects of a C-like function can be precisely determined by computing the weakest precondition of the equality $\mathbf{y} \doteq \mathbf{x}_1$. We have shown that the same ideas can also be used to analyze affine relations in programs with integer arithmetic. The running-time estimations of our new algorithms essentially match those of the corresponding intra-procedural algorithms. In particular when analyzing affine programs, this results in a speedup of a factor k^5 (k the number of program variables) over the corresponding inter-procedural analysis from [20]. The improvement in the complexity could be obtained since we restricted the programs to be analyzed to use functions with local variables only and single return values. In practice this may be combined with an approximative treatment of globals. Though this restriction is severe, there are application scenarios where it still is fully adequate. In [3], e.g., Balakrishnan and Reps use an inter-procedural analysis of affine relations to detect induction variables — which typically are local. In [26] we report on a framework for analyzing multi-threaded C programs. The idea for decoupling the analysis of multiple threads there is to track local information precisely while global data, which potentially can be accessed by other threads, is approximated through a single rough invariant. Obviously, in both contexts our new inter-procedural analysis is the method of choice for computing Herbrand equalities or affine relations. It remains as an interesting problem to find out which other inter-procedural analyses could take advantage of our restriction to functions without globals.

Other questions remain as well. We still do not know how, in presence of global variables, a precise and inter-procedural analysis of valid Herbrand equalities can be constructed. In [17] we have presented an analysis of Herbrand equalities which takes disequality guards into account. It is completely open in how far this intra-procedural analysis can be generalized to any inter-procedural setting.

References

[1] B. Alpern, M. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conf. Record of the 15th ACM POPL*, Jan. 1988.

[2] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.

[3] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction, 13th Int. Conf. (CC)*, pages 5–23. LNCS 2985. Springer-Verlag, 2004.

[4] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181 – 196, 1995.

[5] J. Cocke and J. T. Schwartz. Programming languages and their compilers. Courant Institute of Mathematical Sciences, NY, 1970.

[6] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL: `www.elsevier.nl/locate/entcs/volume6.html`.

[7] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.

[8] K. Gargi. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 45–56. ACM Press, 2002.

[9] S. Gulwani and G. Necula. Discovering affine equalities using random interpretation. In *30th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 74–84, 2003.

[10] S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–352. ACM Press, 2004.

[11] S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Proc. Int. Static Analysis Symposium (SAS'2004)*,. Springer Verlag, 2004. To appear.

[12] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[13] G. A. Kildall. A unified approach to global program optimization. In *Conf. Record of the first ACM POPL*, pages 194 – 206, Boston, MA, 1973.

[14] J. Knoop, O. Rüthing, and B. Steffen. Code motion and code placement: Just synonyms? In *Proc. 6th ESOP*, Lecture Notes in Computer Science 1381, pages 154 – 196, Lisbon, Portugal, 1998. Springer-Verlag.

[15] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Compiler Construction (CC)*, pages 125–140. LNCS 541, Springer-Verlag, 1992.

[16] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, Engelwood Cliffs, New Jersey, 1981.

[17] M. Müller-Olm, O. Rüthing, and H. Seidl. Testing herbrand equalities and beyond. Technical Report 788, Universität Dortmund, Fachbereich Informatik, 2004. Submitted for publication.

[18] M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Information Processing Letters*, 91(5):233–244, 2004.

[19] M. Müller-Olm and H. Seidl. Interprocedural analysis of modular arithmetic. Technical Report 789, Universität Dortmund, Fachbereich Informatik, 2004. Submitted for publication.

[20] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proceedings 31st POPL*, pages 330–341, 2004.

[21] M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *ICALP 2004*, to appear.

[22] J. H. Reif and R. Lewis. Symbolic evaluation and the gobal value graph. In *Conf. Record of the 4th ACM POPL*, pages 104 – 118, Los Angeles, CA, 1977.

[23] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Int. Static Analysis Symposium (SAS)*, pages 189–213. LNCS 2694, Springer-Verlag, 2003.

[24] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Record of the 15th ACM POPL*, pages 12 – 27, San Diego, CA, 1988.

[25] O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Proc. 6th Int. Static Analysis Symposium (SAS'99)*, Lecture Notes in Computer Science 1694, pages 232–247, Venice, Italy, 1999. Springer-Verlag.

[26] H. Seidl, V. Vene, and M. Müller-Olm. Global invariants for analysing multi-threaded applications. *Proc. of Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.

[27] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In [16], chapter 7, pages 189–233.

[28] B. Steffen. Optimal run time optimization—proved by a new look at abstract interpretations. In *Proc. 2nd International Joint Conference on Theory and Practice of Software Development (TAPSOFT'87)*, Lecture Notes in Computer Science 249, pages 52–68. Springer-Verlag, 1987.

[29] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *Proc. Third ESOP*, Lecture Notes in Computer Science 432, pages 389 – 405, Copenhagen, Denmark, 1990. Springer-Verlag.

[30] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *Proc. 4th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT)*, Lecture Notes in Computer Science 494, pages 394 – 415, Brighton, UK, 1991. Springer-Verlag.