Sharing Ghost Variables in a Collection of Abstract Domains

Marc Chevalier, Jérôme Feret

To cite this version:

Marc Chevalier, Jérôme Feret. Sharing Ghost Variables in a Collection of Abstract Domains. VMCAI 2020 - 21st International Conference on Verification, Model Checking, and Abstract Interpretation, Jan 2020, New Orleans, LA, United States. hal-02378809

HAL Id: hal-02378809

https://hal.inria.fr/hal-02378809

Submitted on 25 Nov 2019

Sharing Ghost Variables in a Collection of Abstract Domains*

Marc CHEVALIER[1,2] and Jérôme FERET[1,2]

[1] Inria
[2] Département d'informatique de l'ENS, ENS, CNRS, PSL University, Paris, France
{marc.chevalier,feret}@ens.fr

Abstract. We propose a framework in which we share ghost variables across a collection of abstract domains allowing precise proofs of complex properties.

In abstract interpretation, it is often necessary to be able to express complex properties while doing a precise analysis. A way to achieve that is to combine a collection of domains, each handling some kind of properties, using a reduced product. Separating domains allows an easier and more modular implementation, and eases soundness and termination proofs. This way, we can add a domain for any kind of property that is interesting. The reduced product, or an approximation of it, is in charge of refining abstract states, making the analysis precise.

In program verification, ghost variables can be used to ease proofs of properties by storing intermediate values that do not appear directly in the execution.

We propose a reduced product of abstract domains that allows domains to use ghost variables to ease the representation of their internal state. Domains must be totally agnostic with respect to other existing domains. In particular the handling of ghost variables must be entirely decentralized while still ensuring soundness and termination of the analysis.

1 Introduction

Ghost variables can help to proof complex properties on programs: they store intermediate values that do not appear in the program but help to express complex values or allow better expressivity, like in [16]. We would like to reason on these variables as well as we do on real variables. We propose here a flexible way to make abstract interpretation and ghost variables to work together.

Abstract interpretation [7] is a framework of semantics approximation that allow to prove semantic properties on programs such as the absence of runtime errors. Analyses using abstract interpretation are usually made to be sound and terminating and consequently are not complete. False alarms are unavoidable in theory, but we can strive to make them as rare as possible.

Running an analysis using several domains without communication is no better than running separate analyses. Each domain handles some kind of properties

* This work was partially supported by ANR project AnaStaSec ANR-14-CE28-0014.

but independently, they are usually not precise enough to prove the properties we are interested in. Yet there are good sides to use separate domains. From a mathematical point of view, it makes the proofs of soundness and termination easier and compositional. From a software engineering point of view, it makes the implementation more modular. Usually, we use a reduced product [8] that preserves soundness and termination while improving internal states of the domains: each domain communicates what it knows to all other domains so that they can use this information to improve their own internal state and thus avoiding some false alarms. In this framework, we benefit from mutual induction (where each domain can refine its state thanks to any other domain's information) which is better than cascading analyses, where domains can only get information from previous domain, with respect to a linear ordering. We want to keep the power of mutual induction. A classical method in program verification to express complex properties is to refer to ghost variables. They are variables that are useful for the proof but do not appear in the program. A standard (but not the only) usage of ghost variables is to remember former values of real variables.

Though, ghost variables are difficult to use in a reduced product. The constraints are plenty, since each domain shall not make any assumption on other existing domains. Whatever the chosen combination of domains, the result should always be correct. This has several implications. Firstly, a ghost variable can need another ghost variable to represent its state, thus the set of ghost variables must be handled dynamically. The management of ghost variables (creation, deletion and all operations) is totally distributed. Consequently, we must be sure that these concurrent operations still guarantee a sound abstraction, with very little knowledge on evaluation order. Indeed, deciding an evaluation order would require to know all involved domains. Moreover, the analysis should always terminate even for non-terminating programs.

An example that illustrates the need for ghost variables is to abstract precisely Global Descriptor Table (GDT) entries. In the x86 architecture, GDT is an array in which each entry describes a memory segment or some kind of enriched function pointer (e.g. call gate). Along with various flags controlling permissions and miscellaneous settings, these entries need to store an address: the base address of the memory segment or the bare function pointer. But these addresses are not stored contiguously in memory: they are chopped into several pieces with bitwise operations. Though, remembering precise information is necessary to be able to get a precise pointer when reconstructing it (e.g. to call the function pointed by the call gate). A quite realistic case is the following:

```
/* Given int limit, *base; char access, flags; short t[4]; */
t[0]=limit&0xffff
t[1]=base&0xffff;
t[2]=((base>>16)&0xff)|((access&0xff)<<8);
t[3]=((base>>24)<<8)|((limit>>16)&0xf)|((flags&0xf)<<4);
```

We assume that we have some hypotheses on the architecture and on the compiler that go beyond the C standard but are usual in the context of embedded systems. We want to know that `t[1]|((t[2]&0xff)<<16)|((t[3]&0xff00)<<16)`

has the same value as base. It may be a variable or an arbitrary expression, and evaluates to an integer or a pointer. Typically, it may be fn_ptr_array[index] or &some_function. More generally, low level programming uses extensively bitwise operations, e.g. for pointer alignment or to change the endianness when communicating on a network.

It might be tempting to simply stack domains: each domain has one underlying domain and acts on it as it decides. However, this has several limitations. Firstly, properties of the overlaying domain can use the ones of the underlying domain, but not in the other way or with recursive nesting. For instance, we might want to represent simultaneously slices of linear combinations of pointers, and linear combinations of slices of pointers (this is not a fantasy, it happens in real world low-level source code). Moreover, from an implementation point of view, adding a domain is very costly as it requires to implement all the primitives to translate and forward instructions to the underlying level and to compute inductive invariant from increasing iterations.

2 Related work

Our work stands in the ancient tradition of abstract interpretation which goes back to Cousot & Cousot [7]. Analysis precision can be improved through several strategies, like disjunction of states depending on their contents [4], or the context [2]. Non-standard semantics can also enhance analyses by adding information that does not directly appear in the semantics, for instance in [11, 19, 18] where objects are enriched with information about their history. In [5] ghost variables are used to represent expressions with external symbols. Here, ghost variables are statically allocated. [15, 12, 1] are examples of works where ghost variables have a more dynamic semantics, but they are local to the domain and are not shared with outside domains.

Another way to improve precision is to make several abstract domains work together, typically, with a reduced product [8, 6]. Our work is motivated by pragmatic constraints: allowing easy proofs and implementation by delegation to the domains. In particular our work is an extension of [9]. Our product was introduced to add some new specific domains, that will be discussed in the following, that get their power from shared ghost variables. Nevertheless, some existing abstract domains were adjusted to this new framework such as Miné's pointer abstraction [14]. There are other works on domain cooperation with dynamic support, like cofibered domains [17]. Those have some limitations we try to overcome. The product we propose does not enforce a hierarchy between domains and the current support is known by every domain to improve precision.

[13] also uses transformation of statements but based on expression rewriting. In this approach, several abstraction levels use rewriting rules to gradually simplify expressions. This strategy is well-fitted for function resolution in a context where there are function overloading and dynamic typing, which is crucial to analyze Python source code. Given our application, we have other priorities. This is why we choose to promote a more flexible framework where statement

transformation may depend on the history (see subsection 5.2 for an example). It allows domains to declare new ghost variables without knowing *a priori* when they will be used; their value will be updated during the following computation steps, and used when useful without knowing precisely when they have been declared. Another requirement is to allow arbitrary predicate nesting (cf. section 1): domains are free to use ghost variables to represent any available property, even if it means that some kinds of transformations are not so straightforward.

We implemented all the following in a development version of Astrée. Astrée is a static analyzer designed to analyze critical C source code coming from automotive application, avionics, astronautics, etc. [3]. This development version goes beyond safety and aims to prove security properties that are crucial in critical contexts. It has been successfully used to analyze the source code of a proprietary host platform.

3 The Setup

Let us define the framework in which to work. We remain very general at this point and simply define the concrete semantics as the composition of primitives.

We denote by $\mathbb{V}$ the set of variables, and by $\mathbb{E}$ the set of expressions. We consider any usual operation except dereferencing. Nevertheless, we do not forbid the use of "address of" (`&` in C), so variables may store pointers. Variables are assumed to not alias. For instance, they may represent memory blocks in C and each C statement is transformed to multiple statements to simulate the effect of the C statements on overlapping memory blocks. This is because in Astrée, pointer resolution is performed before (and thanks to) pointer abstraction.

To stay very general, we describe the (non-deterministic) semantics with variable allocation and killing, comparison, assignment and union. Since we have variable allocation and killing, the support (set of living variables) is dynamic. Formally, let $S = \mathbb{V} \rightharpoonup \mathbb{I}$ be the set of states: they are partial maps from variables to values where $\mathbb{I}$ is an arbitrary set. The support of such a state s is denoted $\operatorname{supp}(s)$.

We assume we are given a primitive to assign a variable, one to guard by an arbitrary comparison and one to allocate and kill a variable. The two former shall not change the support, while the two latter respectively add and remove a variable to the support. So, even though the semantics is non-deterministic, each step leads to a set of states with the same support. We denote the poset $D = \left(\left\{\mathcal{S} \in \mathcal{P}(S) \,\middle|\, \forall (s,s') \in \mathcal{S}^2, \operatorname{supp}(s) = \operatorname{supp}(s')\right\}, \subseteq\right)$. The unique support of all states in a $d \in D$ is also denoted $\operatorname{supp}(d)$. We denote $\mathbb{O} = \{<, >, =, \neq, \leqslant, \geqslant\}$, $\mathbb{C} = \mathbb{E} \times \mathbb{O} \times \mathbb{E}$ the type of comparisons and $\mathbb{A} = \mathbb{V} \times \mathbb{E}$ the type of assignments. The directives aforementioned have types $\mathit{Assign} : \mathbb{A} \times S \to D$, $\mathit{Comp} : \mathbb{C} \times S \to D$, $\mathit{Alloc} : \mathbb{V} \times S \to D$ and $\mathit{Kill} : \mathbb{V} \times S \to D$, with the assumptions:

- $\forall (a,s) \in \mathbb{A} \times S, \operatorname{supp}(\mathit{Assign}(a,s)) = \operatorname{supp}(s)$
- $\forall (c,s) \in \mathbb{C} \times S, \operatorname{supp}(\mathit{Comp}(c,s)) = \operatorname{supp}(s)$
- $\forall s \in S, \forall v \notin \operatorname{supp}(s), \mathit{Alloc}(v,s) = \operatorname{supp}(s) \cup \{v\}$
- $\forall s \in S, \forall v \in \operatorname{supp}(s), \mathit{Kill}(v,s) = \operatorname{supp}(s) \setminus \{v\}$

We identify these functions with their lift to D, ie., they respectively are $Assign : \mathbb{A} \times D \to D$, $Comp : \mathbb{C} \times D \to D$, $Alloc : \mathbb{V} \times D \to D$ and $Kill : \mathbb{V} \times D \to D$.

Let us add some requirements inspired by the C language. First, even if the support is dynamic (due to local variables, scopes, ...), we assume that non existing variables do not appear in expressions, and that binary flow operations (such as union) are performed with the same support in both operands. We also assume there is no allocating an existing variable or killing of a dead variable. Allocations of variables are done by *Alloc* and killings by *Kill*; all other primitives keep the same support. We add a few assumptions:

- In assignments, the left-hand side variable does not appear in the right-hand side expression.
- $\forall a \in D, \forall (c, d) \in \mathbb{C}^2, \forall (e, f, g) \in \mathbb{E}^3, \forall (v, w) \in \mathbb{V}^2, \forall o \in \mathbb{O}, v \neq w$
  - $\{v, w\} \cap (\mathrm{Var}\,(e) \cup \mathrm{Var}\,(f)) = \emptyset \Rightarrow Assign((v, e), Assign((w, f), a))$ $= Assign((w, f), Assign((v, e), a))$ where $\mathrm{Var}\,(e)$ is the set of variables in e (independent assignments commute)
  - $Comp(c, Comp(d, a)) = Comp(d, Comp(c, a))$ (comparisons commute)
  - $v \notin \mathrm{Var}\,(e) \cup \mathrm{Var}\,(f) \Rightarrow$ $Comp((e, o, f), Assign((v, g), a)) = Assign((v, g), Comp((e, o, f), a))$ (independent comparison and assignment commute)
  - $Comp(c, a) \subseteq a$

These assumptions are quite reasonable and allow to commute statements as long as they are independent enough. It will come in handy to run a set of statements in a distributed way without guarantees on the execution order.

We write $[\![c]\!]$ the function $D \to D$ that runs the statement c. We naturally extend this notation to any sequence of instructions and control flow graphs (using least fix-point operator to stabilize loops).

4 An Abstract Domain with Dynamic Support

4.1 The Difficulties

Using a single abstract domain to analyze programs may result in poor precision. To overcome this issue, we use the reduced product [8]. This is a compound abstract domain that runs a collection of domains and make them cooperate to improve their internal states. Formally, the internal state of each domain is the best abstraction of the intersection of the concretization of all internal states. This structure allows easier (because separate) implementation and correctness proofs. The reduced product ensures the best possible precision given the set of available domains, however, this is not realistic. But we can use an over-approximation of the reduced product: internal states may be improved, but they are not supposed to be the best possible.

We want a product that allows each domain to add new ghost variables to the current state, reduce them and kill them. All domains must store ghost variables of all domains: they must communicate their policy for ghost variables management, i.e. what to do on ghost variables while we operate on real variables. Recursively, reduction of ghost variables may trigger new reductions.

There are several problems that the proposed framework is solving: – one cannot order meaningfully constraints coming from different domains. So concurrent constraints must be safely reorderable; – we need to recursively collect all constraints on ghost variables while ensuring termination; – when performing binary operations, we have assumed that the support of real variables is the same, but there is no such guarantee about ghost variables. So, before performing a binary operation, supports shall be unified.

Let us take a look at a simpler code that sublimates the difficulties of the real-world case and that will be used all along the paper to illustrate the framework:

```
int* x = &t[idx]; /* given int t[]; and int idx; */
int low = x & 0xffff;
int high = x >> 16;
int* y = low | (high << 16);
```

In this example, mask and bit shift are used to split a pointer and recombine it.

4.2 Ghost Variables and Constraints

Let us define ghost variables, their structure and their relation to abstract domains. Ghost variables are built recursively from normal or ghost variables using unary constructors. These constructors specify the semantics of the ghost variable, for this reason, these constructors are called ”roles”. Each role must be associated to the domain in charge of managing this kind of ghost variables. To define all the component of the signatures of abstract domains, we need to reference them, although we still are not able to define them entirely.

Notation 1 (Domain name). *Let us denote $\mathcal{N}$ a set of domain names.*

To analyze properly the running example we will use 3 domains: bitwise properties, pointers as a memory block and an offset [14] and equality. These domains are respectively named $\mathfrak{Slice}$, $\mathfrak{Offset}$ and $\mathfrak{Equality}$. In the examples, we can choose $\mathcal{N}$ to contain only these three names.

Each role is associated to the domain that decides what to do with ghost variables that have this role as outermost one.

Definition 1 (Roles). *Let $\mathcal{R}$ be a set of unary constructors. These constructors are called ”roles”. Let $\mathfrak{o} : \mathcal{R} \to \mathcal{N}$. When $\mathfrak{o}(\mathscr{R}) = d$, we say that d is the owner of a role $\mathscr{R}$, or that $\mathscr{R}$ belongs to d, and we denote $\mathscr{R} \Subset d$.*

For instance, after the second line of the simplified example, we would like to remember that the 16 upper bits of `low` are 0 and the 16 lower bits are the 16 lower bits of a ghost variable named $\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low})$. We can write this property $\texttt{low} = [_0\, \mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low})[0,15]\,_{15}|_{16}\, 0\,_{31}]$, where $v[a,b]$ designates the bits a to b of the variable v, and $\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low})$ stores the value of `x`. This way, even if `x` falls out of scope or is modified, the ghost variable is safe.

From now on, everything written in this Round Hand style (like $\mathscr{R}$) is about ghost variables.

Notation 2 (All variables). *We denote by $\mathscr{V}$ the inductively defined set of all variables, starting with real variables (in $\mathbb{V}$).*

Formally $\mathscr{V} := \bigcup_{i \in \mathbb{N}} \mathscr{V}_i$ *where* $\mathscr{V}_i := \begin{cases} \mathbb{V} & \textit{if } i = 0 \\ \{\mathscr{R}(v) \mid \mathscr{R} \in \mathcal{R}, v \in \mathscr{V}_{i-1}\} & \textit{otherwise} \end{cases}$
$\mathscr{E}$ *denotes the set of expressions whose variables are* $\mathscr{V}$*.*

It is worth noticing that $\mathbb{V} \subseteq \mathscr{V}$ and $\mathbb{E} \subseteq \mathscr{E}$. We assume we can naturally extend concrete primitives to $\mathscr{V}$ and $\mathscr{E}$.

Though the handling of ghost variables is decentralized, there are restrictions on which variables a domain can create or delete. For instance, only the domain $\mathfrak{Slice}$ should be able to decide what to do with $\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low})$. So, we need a belonging relation between ghost variables and domain names.

Definition 2. *We extend the* $\Subset$ *to* $\mathscr{V} \times \mathcal{N}$ *by* $\forall v \in \mathscr{V}, \forall \mathscr{R} \in \mathcal{R}, \mathscr{R}(v) \Subset \mathfrak{o}(\mathscr{R})$

The ownership is purely syntactic: the topmost role is enough to decide the owner of a ghost variable. Real variables (in $\mathbb{V}$) are not owned.

A role defines the semantics of the ghost variable. This is usually a relation between the value of the ghost variable and the variable immediately above. For instance, $\mathscr{O}\text{ffset}(p)$ should be the offset of p.

Notation 3. *Let* $(\!|\_|\!) : \mathscr{V} \to \mathcal{P}(S)$ *be the semantics of ghost variables.*

For instance, $(\!|\mathscr{O}\text{ffset}(p)|\!)$ is the set of states where $\mathscr{O}\text{ffset}(p)$ is indeed the offset part of the pointer p. The codomain of $(\!|\_|\!)$ is not D since legal states do not have necessarily the same support.

Definition 3 (Ghostly ordering). *Let* $\lhd$ *be the smallest transitive relation on* $\mathscr{V}^2$ *such that* $\forall v \in \mathscr{V}, \forall \mathscr{R} \in \mathcal{R}, \mathscr{R}(v) \lhd v$*. Let* $\trianglelefteqslant$ *be the reflexive closure of* $\lhd$*. If* $x \lhd y$*, we say that* x *is less real (or ghostlier) than* y*. We extend this relation to* $\mathscr{E}^2$ *by* $\forall (e, e') \in \mathscr{E}^2, (\forall v' \in \text{Var}(e'), \exists v \in \text{Var}(e) : v' \trianglelefteqslant v) \Leftrightarrow e' \trianglelefteqslant e$ *where* $\text{Var}(e)$ *is the set of variables of the expression* e*. That is every variable in* e' *is ghostlier than a variable in* e*.*

$\trianglelefteqslant$ is clearly an order on $\mathscr{V}$ and a preorder on $\mathscr{E}$. Now that we have all variables, we want to operate on them. In addition to variable creation/deletion that are handled separately, domains can exchange information about ghost variables in the form of comparisons and directed reduction.

Definition 4 (Ghost constraints). *We let* $\mathscr{O} := \{\leqslant, <, \geqslant, >, =, \neq\}$*,* $\mathscr{C} := \mathscr{E} \times \mathscr{O} \times \mathscr{E}$ *(comparison),* $\mathscr{A} := \mathscr{V} \times \mathscr{E}$ *(directed constraint) and* $\mathscr{U} := \mathscr{C} \uplus \mathscr{A}$

These are the types of constraints about ghost variables. $\mathscr{C}$ is the set of comparisons of two arbitrary expressions. $\mathscr{A}$ are equality between a variable and an expression, but with a restriction: they are used to point out the variable is unknown (it is $\top$) and that reduction may only occur in one direction. This reduction can be implemented very efficiently as an assignment. These constraints are written in the form $v \leftarrow e$. This may look like a mere implementation concern, but it relies on fundamental assumptions.

First constraints are generated from the executed statement (in $\mathbb{A}$ or $\mathbb{C}$), then each constraint may trigger the generation of several other constraints. Let us see on a simple example why atomic constraints are not expressive enough. We consider two variables `a` and `b` that have been built with bitwise instructions, like in the example. Let us say they are both made of two parts of 16 bits:

$$\begin{aligned}\mathtt{a} &= [_0 \quad \mathscr{S}lice_{[0,15]\to[0,15]}(a)[0,15] \quad {}_{15}|_{16} \quad \mathscr{S}lice_{[16,32]\to[16,32]}(a)[16,31] \quad {}_{31}]\\ \mathtt{b} &= [_0 \quad \mathscr{S}lice_{[0,15]\to[0,15]}(b)[0,15] \quad {}_{15}|_{16} \quad \mathscr{S}lice_{[16,32]\to[16,32]}(b)[16,31] \quad {}_{31}]\end{aligned}$$

A way to run the comparison $a = b$ in a state s is to compute $f \circ g(s)$ where

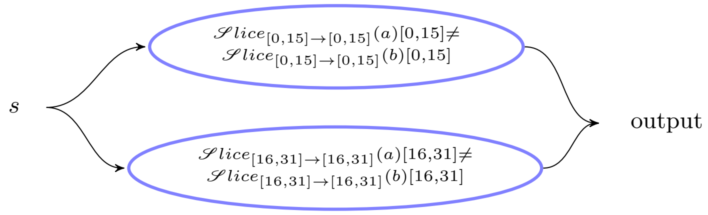
- $f = [\![\mathscr{S}lice_{[0,15]\to[0,15]}(a)[0,15] = \mathscr{S}lice_{[0,15]\to[0,15]}(b)[0,15]]\!]$
- $g = [\![\mathscr{S}lice_{[16,31]\to[16,31]}(a)[16,31] = \mathscr{S}lice_{[16,31]\to[16,31]}(b)[16,31]]\!]$

So we need to be able to sequence constraints. If we want the result of $a \neq b$, we need to compute $f(s) \cup g(s)$ where

- $f = [\![\mathscr{S}lice_{[0,15]\to[0,15]}(a)[0,15] \neq \mathscr{S}lice_{[0,15]\to[0,15]}(b)[0,15]]\!]$
- $g = [\![\mathscr{S}lice_{[16,31]\to[16,31]}(a)[16,31] \neq \mathscr{S}lice_{[16,31]\to[16,31]}(b)[16,31]]\!]$

To combine both branching and sequencing, and still have obvious termination, the compound constraints communicated across domains are DAGs. More precisely the internal language to communicate constraints is made of DAGs with one source and one sink and whose edges wear sequence of constraints in $\mathscr{U}$. Edges converging to a node mean that a join must be performed before guarding by the constraints on the node. It allows conditional branching and avoids loops and thus non-termination. For instance, the graph corresponding to the test `a` $\neq$ `b` is given in Fig. 1.

This kind of compound constraints is quite powerful. The graph in Fig. 2 computes $\kappa(\theta(\zeta(i) \sqcup \eta(i)) \sqcup \iota(\eta(i)))$ where $i \in D^\sharp$ is the input. We notice that $\eta(i)$ appears several times in the one-line expression, but the graph form allows easy sharing of intermediate computations.



s
$\mathscr{S}lice_{[0,15]\to[0,15]}(a)[0,15] \neq$ $\mathscr{S}lice_{[0,15]\to[0,15]}(b)[0,15]$
$\mathscr{S}lice_{[16,31]\to[16,31]}(a)[16,31] \neq$ $\mathscr{S}lice_{[16,31]\to[16,31]}(b)[16,31]$
output

Fig. 1. An example of a constraint DAG

Notation 4 (Constraint graph). *$\mathscr{G}$ is the set of 3-tuples (V, E, n) where (V, E) is a DAG with exactly one source and one sink and $n : V \to \mathscr{U}^*$ maps nodes to finite sequences of elements of $\mathscr{U}$. Given $g \in \mathscr{G}$ and $u \in \mathscr{U}$, we write $u \in g$ when $\exists v \in V : u \in n(v)$.*

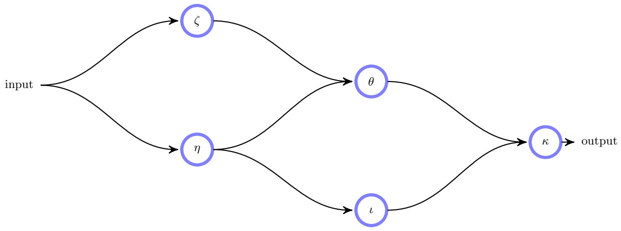


Fig. 2. A more complicated constraint DAG

In fact, the exact form of the intermediate language does not matter. If needed, it could include other terminating constructs like constant-bounded for-style loops. We choose this language made of DAGs because it is both simple and expressive enough.

Given a support V, the semantics of such graph is defined inductively on the nodes and is an element of D denoted $[\![G]\!]_V$. The semantics of the source is $\{s \in S \mid \operatorname{supp}(s) = V\}$ and the semantics of each node is the union of the semantics of the parent nodes guarded by the constraints of the node. The semantics of the graph is the semantics of the sink. This is well-founded thanks to acyclicity.

A graph expresses constraints about ghost variables, so to be a sound constraint, its semantics shall be bigger than the intersection of the semantics of ghost variables. In other words, it can do no more than enforcing the semantics of ghost variables.

Definition 5 (Constraint-DAG soundness). *A graph G is said to be sound in the support $V \subseteq \mathscr{V}$ if* $\left\{ s \in \bigcap_{v \in V \setminus \mathbb{V}} (\!|v|\!) \;\middle|\; \operatorname{supp}(s) = V \right\} \subseteq [\![G]\!]_V$.

Though any sound graph can be used, there are several ways to build them systematically. Here is a non-exhaustive list that covers most common cases.

- If we test equality between variables x and y in a context where both $\mathscr{R}(x)$ and $\mathscr{R}(y)$ exist, for a given role $\mathscr{R}$, and the implication $x = y \Rightarrow \mathscr{R}(x) = \mathscr{R}(y)$ is true in all states, we can generate the one-node graph containing the unique constraint $\mathscr{R}(x) = \mathscr{R}(y)$. If several such implications hold, we may simply sequence all graphs in an arbitrary order (since these conditions do not interfere). This is the general case of the example `a` $=$ `b`.
- To test difference between variables x and y in a context where $(\mathscr{R}_i(x))_{i \in [\![1,n]\!]}$ and $(\mathscr{R}_i(y))_{i \in [\![1,n]\!]}$ exist, for a given family of roles $(\mathscr{R}_i)_{i \in [\![1,n]\!]}$ and the implication $x \neq y \Rightarrow \bigvee_{i \in [\![1,n]\!]} \mathscr{R}_i(x) \neq \mathscr{R}_i(y)$ holds in any state, we can generate the graph that joins the results of these n conditions taken separately (like Fig. 1). This is like the comparison `a` $\neq$ `b` in the last example.
- Some constraints come from the language semantics but can be reinterpreted with ghost variables. E.g. since the ghost variable offset must coincide with the C standard offset, the latter can be substituted by the former.

Directed constraints $(v \leftarrow e)$ enforce that the left-hand side is $\top$, to be able to run it efficiently as an assignment. The way to ensure this property stands in two arguments: – a directed constraint about a variable can only be issued by the domain that owns the variable; – after a variable assignment, all consequent assignments can only modify ghostlier variables.

Of course, the condition cannot be totally local. Here, the hidden global hypothesis is that the roles of different domains are disjoint. To help the proof that the restriction on directed constraints holds, we introduce the following relation. If any generated constraint is smaller than the previous with respect to this relation, then only variables set to $\top$ will appear in the left-hand side of a directed constraint, and at most once.

Definition 6. *Let $n \in \mathcal{N}$. Let $\succ_n$ the smallest relation on $(\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U}) \times \mathscr{U}$ satisfying:*

- $\forall v \in \mathscr{V}, \forall \mathscr{R} \in \mathcal{R}, \forall (e, e') \in \mathscr{E}^2, \mathscr{R} \Subset n \wedge e' \trianglelefteqslant e \Rightarrow (v, e) \succ_n (\mathscr{R}(v), e')$
- $\forall (a, b, c, d) \in \mathscr{E}^4, \forall (o, p) \in \mathscr{O}^2, a \trianglelefteqslant c \wedge b \trianglelefteqslant d \Rightarrow (c, o, d) \succ_n (a, p, b)$
- $\forall (a, b, c) \in \mathscr{E}^3, \forall v \in \mathscr{V}, \forall o \in \mathscr{O}, b \trianglelefteqslant a \wedge c \trianglelefteqslant a \Rightarrow (v, a) \succ_n (b, o, c)$

The type of the left-hand side is $\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U}$, this is because a ghost constraint can be generated from a real assignment ($\mathbb{A}$), a real comparison ($\mathbb{C}$), or another ghost constraint ($\mathscr{U}$). At the first step, the left-hand side is in $\mathbb{A} \uplus \mathbb{C}$; after, only elements of $\mathscr{U}$ appear. Let us reword the three cases: – a directed reduction (resp. assignment) can trigger a directed reduction about a variable immediately ghostlier with a ghostlier right-hand side; – a (real or ghost) comparison can trigger a comparison that involves ghostlier expression; – a directed reduction (resp. assignment) can trigger a comparison whose expressions are ghostlier than the right-hand side of the directed reduction (resp. assignment).

4.3 Generic Abstract Domain

Before describing the structure of an abstract domain, let us see how the running example behaves. We assume we use the 3 domains we introduced before: one to express bitwise properties (named $\mathfrak{Slice}$), one that represents pointers as a block and an offset [14] ($\mathfrak{Offset}$) and one about equality ($\mathfrak{Equality}$). The $\mathfrak{Offset}$ domain uses a single role $\mathscr{O}$ffset that means the ghost variable is the offset of the base variable as a pointer. The equality domain does not have any role.

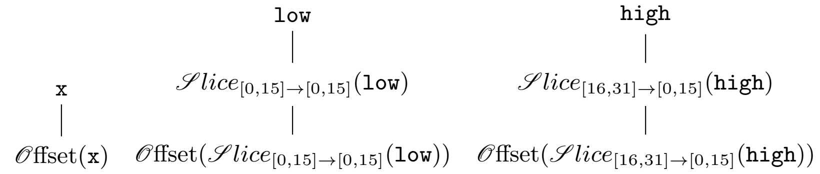
At the end of the third line, we have:

$$\left.\begin{aligned} \mathtt{low} &= [_0\ \mathscr{S}lice_{[0,15]\to[0,15]}(\mathtt{low})[0,15]\ {}_{15}|_{16}\ 0\ {}_{31}] \\ \mathtt{high} &= [_0\ \mathscr{S}lice_{[16,31]\to[0,15]}(\mathtt{high})[16,31]\ {}_{15}|_{16}\ 0\ {}_{31}] \end{aligned}\right\} \mathfrak{Slices}$$

$$\left.\begin{aligned} \mathtt{x} &= \mathtt{t} + \mathscr{O}\mathrm{ffset}(\mathtt{x}) \\ \mathscr{S}lice_{[0,15]\to[0,15]}(\mathtt{low}) &= \mathtt{t} + \mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\mathtt{low})) \\ \mathscr{S}lice_{[16,31]\to[0,15]}(\mathtt{high}) &= \mathtt{t} + \mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[16,31]\to[0,15]}(\mathtt{high})) \end{aligned}\right\} \mathfrak{Offset}$$

$$\left.\begin{aligned} \mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\mathtt{low})) &= \mathscr{O}\mathrm{ffset}(\mathtt{x}) \\ \mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[16,31]\to[0,15]}(\mathtt{high})) &= \mathscr{O}\mathrm{ffset}(\mathtt{x}) \end{aligned}\right\} \mathfrak{Equality}$$

The support consists in the following trees of variables:



low
high
x
$\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low})$
$\mathscr{S}lice_{[16,31]\to[0,15]}(\texttt{high})$
$\mathscr{O}\text{ffset}(\texttt{x})$
$\mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low}))$
$\mathscr{O}\text{ffset}(\mathscr{S}lice_{[16,31]\to[0,15]}(\texttt{high}))$

Now, let us run the last statement. $\mathfrak{Offset}$ and $\mathfrak{Equality}$ domains get nothing interesting from that. However, the $\mathfrak{Slice}$ domain is more clever. It can deduce

$$\texttt{y} = [_0 \quad \mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y})[0,15] \quad {}_{15}|_{16} \quad \mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y})[0,15] \quad {}_{31}]$$

but requires the creation of two ghost variables (initialized to $\top$) and yields two directed reductions: $\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y}) \leftarrow \mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low})$ and $\mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y}) \leftarrow \mathscr{S}lice_{[16,31]\to[0,15]}(\texttt{high})$.

One can remark that the restriction on directed constraints is satisfied: both left-hand sides are $\top$ since they are freshly allocated. After this reduction, we know that:

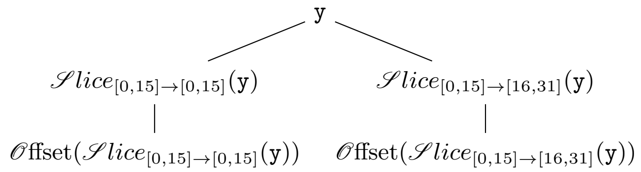
$$\left.\begin{aligned}\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y}) &= \texttt{t} + \mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y}))\\ \mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y}) &= \texttt{t} + \mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y}))\end{aligned}\right\}\ \mathfrak{Offset}$$

$$\begin{aligned}\mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y})) &\leftarrow \mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low}))\\ \mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y})) &\leftarrow \mathscr{O}\text{ffset}(\mathscr{S}lice_{[16,31]\to[0,15]}(\texttt{high}))\end{aligned}$$

Again, left hand-sides of constraints are unknown before reduction. Finally, we get:

$$\left.\begin{aligned}\mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y})) &= \mathscr{O}\text{ffset}(\texttt{x})\\ \mathscr{O}\text{ffset}(\mathscr{S}lice_{[16,31]\to[16,31]}(\texttt{y})) &= \mathscr{O}\text{ffset}(\texttt{x})\end{aligned}\right\}\quad \mathfrak{Equality}$$

and we use this additional tree of ghost variables:



y
$\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y})$
$\mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y})$
$\mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y}))$
$\mathscr{O}\text{ffset}(\mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y}))$

This illustrates why we need distributed handling of ghost variables and that they should be known by other domains. In fact, when we use a ghost variable, it is likely that another domain can do better about it, otherwise we wouldn't need a ghost variable. For instance, parts of `low`, `high` and `y` are pointers: it is more suitable to ask the pointer domain to represent them. Likewise, the offset is a numeric value so, while the pointer domain associates it to the pointer of which it is the offset, it is worth to let a numeric domain represent the offset value. But, if we know for offset, in general we don't *a priori* know the kind of value the ghost variable store, for instance, in the case of bitwise slices that can be pointers, numeric values or anything else. Furthermore, we do not know in advance the list of available domains, so we cannot decide which domain is the best to represent a ghost variable. Thus, every ghost variable must be known by all domains. This prevents the simple use of stacked domains, each being controlled by the overlaying one. Indeed, a domain won't be able to make one of its ghost variable represented by a domain higher in the hierarchy.

Let us discuss the problems that we have highlighted. The termination is satisfied: directed reductions of offsets do not trigger new constraints. We can observe that each constraint is more elementary than the previous, and there is nothing simpler than copying a numeric variable. The decreasing complexity is a good approach to ensures termination. The other problem was the execution order. Here, we can see that the assignments are disjoint: we assign variables under `y` from variables under `low` and `high`. So, an assigned variable is never in the right-hand side of an assignment and since we always go deeper in ghost variables, each variable is assigned only once, making directed reductions legal. Both these points will be detailed and generalized later.

This result is correct but not interesting in this form. With domain cooperation, we can infer $\mathscr{S}lice_{[0,15]\to[0,15]}(\mathtt{y}) = \mathtt{x}$ and $\mathscr{S}lice_{[0,15]\to[16,31]}(\mathtt{y}) = \mathtt{x}$, hence $\mathtt{y} = \mathtt{x}$ which is the expected result.

We can be very general on the form of the abstract domains. Classically, a domain needs a set of abstract states and their meaning in the concrete world, fixpoint approximation and abstract counterparts of concrete primitives. In addition, we add two other kinds of function.

First, there are two maps to decide what to do with ghost variables: one to react to the execution of a unary statement or a ghost constraint, and one to unify supports before performing a binary operation (typically, the join).

Moreover, domains include primitives to communicate information about their abstract state *à la* [9]. This allows domains to refine themselves (as a reduced product is meant to) but also to communicate their policy on ghost variables management: since all domains must care about the ghost variables of the other domains, they need to communicate what to do.

We are given a lattice $IO^\sharp$ with a concretization $\gamma_{IO} : IO^\sharp \to D$. This is a lattice common to all domains that will be the middleman for all communication.

Definition 7 (Generic abstract domain). *A generic abstract domain with dynamic support is a tuple*
$(n, D^\sharp, \gamma, \sqcup, \mathrm{lfp}^\sharp, \textsc{Assign}, \textsc{Comp}, \textsc{Alloc}, \textsc{Kill}, \textsc{Extract}, \textsc{Refine}, \mathcal{U}, \mathcal{B})$ *where:*

- $n \in \mathcal{N}$
- $D^\sharp$ *is a set of abstract properties,*
- $\gamma : D^\sharp \to D$ *is the concretization, for any* $a \in D^\sharp$ *we denote* $\mathrm{supp}\,(a) := \mathrm{supp}\,(\gamma(a))$*,*
- $\sqcup : D^\sharp \to D^\sharp \to D^\sharp$ *such that* $\forall (a, b) \in {D^\sharp}^2, \mathrm{supp}\,(a) = \mathrm{supp}\,(b) \Rightarrow \gamma(a) \cup \gamma(b) \subseteq \gamma(a \sqcup b)$
- $\mathrm{lfp}^\sharp : (D^\sharp \to D^\sharp) \to D^\sharp \to D^\sharp$ *such that* $\forall f : D \to D, \forall f^\sharp : D^\sharp \to D^\sharp, f \circ \gamma \subseteq \gamma \circ f^\sharp \Rightarrow \mathrm{lfp}(f) \subseteq \gamma \circ \mathrm{lfp}^\sharp\left(f^\sharp\right)$
- $\textsc{Assign} : (\mathscr{V} \times \mathscr{E}) \times D^\sharp \to D^\sharp$ *such that*

  - $\forall (v, e, a) \in \mathscr{V} \times \mathscr{E} \times D^\sharp, \mathit{Assign}((v, e), \gamma(a)) \subseteq \gamma(\textsc{Assign}((v, e), a))$
  - $\forall ((v, e), a) \in \mathscr{A} \times D^\sharp,$ ***let*** $d = \gamma(\textsc{Assign}(v, e, a))$ ***in*** $\forall (w, s, x) \in \{v\}^{\triangleleft} \times d \times \mathbb{I}, s[w \mapsto x] \in d$ *where* $S^{\mathcal{R}} := \{x \mid \exists y \in S : x\mathcal{R}y\}$.
- $\textsc{Comp} : \mathscr{C} \times D^\sharp \to D^\sharp$ *such that* $\forall (c, a) \in \mathscr{C} \times D^\sharp, \mathit{Compare}(c, \gamma(a)) \subseteq \gamma(\textsc{Comp}(c, a))$
- $\textsc{Alloc} : \mathscr{V} \times D^\sharp \to D^\sharp$ *such that* $\forall (v, a) \in \mathscr{V} \times D^\sharp, \mathit{Allocate}(v, \gamma(a)) \subseteq \gamma(\textsc{Alloc}(v, a))$
- $\textsc{Kill} : \mathscr{V} \times D^\sharp \to D^\sharp$ *such that* $\forall (v, a) \in \mathscr{V} \times D^\sharp, \mathit{Kill}(v, \gamma(a)) \subseteq \gamma(\textsc{Kill}(v, a))$
- $\textsc{Extract} : D^\sharp \times \mathit{IO}^\sharp \to \mathit{IO}^\sharp$ *such that* $\forall a \in D^\sharp, \forall \mathit{io} \in \mathit{IO}^\sharp, \gamma(a) \cap \gamma_{IO}(\mathit{io}) \subseteq \gamma_{IO}(\textsc{Extract}(a, \mathit{io}))$
- $\textsc{Refine} : D^\sharp \times \mathit{IO}^\sharp \to D^\sharp$ *such that* $\forall a \in D^\sharp, \forall \mathit{io} \in \mathit{IO}^\sharp, \gamma(a) \cap \gamma_{IO}(\mathit{io}) \subseteq \gamma(\textsc{Refine}(a, \mathit{io}))$
- $\mathcal{U} : D^\sharp \to (\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U}) \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G}$ *such that for all* $(a, u) \in D^\sharp \times (\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U})$, *letting* $(\mathit{new}, \mathit{old}, G) := \mathcal{U}(a)(u)$
  - $\mathit{new} \cap \operatorname{supp}(a) = \emptyset$ *and* $\mathit{old} \subseteq \operatorname{supp}(a)$
  - *variables that occur in* G *belong to* $\mathit{new} \cup \operatorname{supp}(a)$.
  - G *is sound in support* $(\operatorname{supp}(a) \cup \mathit{new}) \setminus \mathit{old}$
  - $\forall (u', u'') \in G^2, u \succ_n u' \wedge ((u', u'') \in \mathscr{A}^2 \wedge \pi_1(u') = \pi_1(u'') \Rightarrow u' = u'')$ *where* π_i *is the* i^{th} *projection.*
- $\mathcal{B} : D^\sharp \times D^\sharp \to \mathbb{N} \to (\mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G})^2$ *such that* $\forall (a, b) \in {D^\sharp}^2, \forall i \in \mathbb{N}$
  - $\gamma(a) \subseteq \gamma(a') \wedge \gamma(b) \subseteq \gamma(b')$
  - $((\forall j \in [\![0, i-1]\!], \operatorname{supp}(a) \cap \mathscr{V}_j = \operatorname{supp}(b) \cap \mathscr{V}_j) \Rightarrow (\forall j \in [\![0, i]\!], \operatorname{supp}(a') \cap \mathscr{V}_j = \operatorname{supp}(b') \cap \mathscr{V}_j))$

  where $a' = [\![(\mathit{new}_1, \mathit{old}_1, g_1)]\!]^\sharp(a)$, $b' = [\![(\mathit{new}_2, \mathit{old}_2, g_2)]\!]^\sharp(b)$, $((\mathit{new}_1, \mathit{old}_1, g_1), (\mathit{new}_2, \mathit{old}_2, g_2)) = \mathcal{B}(a, b)(i)$ *and* $[\![(\mathit{new}, \mathit{old}, g)]\!]^\sharp$ *denote allocating variables in new guarding by* g, *and killing variables in old.*

The Assign directive makes all ghost variables under the assigned variable unknown, since their previous value is not *a priori* valid anymore.

Let us take a look at the types of ghost variables management maps. Given a state $a \in D^\sharp$ and an unary statement or a constraint $u \in (\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U})$, $\mathcal{U}(a)(u)$ yields a 3-tuple: the sets of ghost variables to add, the set of ghost variables to kill and a constraint-DAG. This result must be shared across all domains, and we must proceed recursively for each constraint in the DAG.

Of course there are more hypotheses for the domain to be always terminating during recursive exploration of constraints. They will be detailed in the following since it isn't an intrinsic property of domains.

The $\mathcal{B}$ map is slightly more tricky. It is used to unify supports of ghost variables layer by layer: the first call care about ghost variables that have only one role around a real variable, the second call is for variable with two nested roles, and so on. Similarly to $\mathcal{U}$ it returns the actions to perform on each branch.

4.4 Running Ghost Constraints

To get more precision, functions $\mathcal{U}$ of all domains must be called recursively to get as many constraints about ghost variables as possible. In the state a,

$\mathcal{U}(a) : (\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U}) \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G}$ is the function that returns subconstraints that are implied by the constraint in argument. We denote this map by $\mathcal{U}_a$. Every domain shall receive the partial application $\mathcal{U}_a$ of all domains, making them able to know what other domains know. Transmitting these partial applications can be done using EXTRACT and REDUCE via the channel $IO^\sharp$ which must be performed before each real statement.

We combine them using the function

$$C = (f_n)_{n \in N} \mapsto \left(u \mapsto \bigcup_{n \in N} \pi_1(f_n(u)), \bigcup_{n \in N} \pi_2(f_n(u)), \bigcup_{n \in N} \{\pi_3(f_n(u))\} \right)$$

Overall, in the state a we get from all domains a single function $\mathcal{U}_a = C((\mathcal{U}_n(a_n))_{n \in N})$ where all returned sets are finite. Especially, the third component has the same cardinal as N.

When executing a constraint, all these functions are called, and they return constraint-DAGs to run. The domain executes these constraints (using the abstract primitives) by taking care of recursively asking other domains for ghostlier constraints. Eventually, each constraint generates a tree of constraint-DAGs. The recursive exploration terminates when the graph returned is empty.

Definition 8. *Given* $\mathcal{U}_a : (\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U}) \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{G})$ *we define*

$$\begin{aligned}
&\mathcal{U}_a^* : (\mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U}) \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{G}) \\
&\qquad\qquad \textbf{\textit{let}}\ \mathit{new}, \mathit{old}, g = \mathcal{U}_a(u)\ \textbf{\textit{in}} \\
&\quad u \mapsto \quad \operatorname{lfp} \begin{pmatrix} (N, O, G) \mapsto\ \ \textbf{\textit{let}}\ R = \{\mathcal{U}_a(u') \mid g' \in G, u' \in g'\}\ \textbf{\textit{in}} \\ \mathit{new} \cup \{\pi_1(r) \mid r \in R\}, \mathit{old} \cup \{\pi_2(r) \mid r \in R\}, g \cup \{\pi_3(r) \mid r \in R\} \end{pmatrix}
\end{aligned}$$

$\mathcal{U}_a^*$ *is the analog of a transitive closure, hence the notation.*

Communications are performed between a recursive ghost reduction and the next real statement (in $\mathbb{A}$ or $\mathbb{C}$) to improve precision and exchange new $\mathcal{U}_n(a_n)$.

4.5 Non-interference

As previously mentioned, directed constraints ensure that their left-hand side is $\top$ so as to be able to implement them as assignments. Clearly, such assignments cannot interfere with each other or with other comparisons, since they are semantically equivalent to a comparison (and we assumed comparisons commute). We just have to ensure that the hypothesis holds: the left-hand side of directed constraints must be $\top$. In the example, we see that it was achieved by increasing the ghostliness of the involved variables. We would like to guarantee non-interference with a local condition, i.e. a hypothesis that must be verified in each domain independently and that makes any combination of domains correct.

Theorem 1 (Non-interference). *Let* $n \in \mathbb{N}$ *and* $(\mathfrak{D}_i)_{i \in [\![1,n]\!]}$ *a collection of domains with distinct names. Let* $(a_i \in D_i^\sharp)_{i \in [\![1,n]\!]}$. *Let* $\mathcal{U}_a := C\left((\mathcal{U}_i(a_i))_{n \in [\![1,n]\!]}\right)$. *Let* $u \in \mathbb{A} \uplus \mathbb{C} \uplus \mathscr{U}$ *and* $G := \pi_3\left(\mathcal{U}_a^*(u)\right)$.

- *If* $u \in \mathbb{A} \uplus \mathscr{A}$*,* $\forall (u', u'') \in G^2, \big((u', u'') \in \mathscr{A}^2 \wedge \pi_1(u') = \pi_1(u'')\big) \Rightarrow (u' = u'' \wedge \pi_1(u') \in \pi_1(u)^{\triangleleft})$
- *If* $u \in \mathbb{C} \uplus \mathscr{C}$*,* $\forall u' \in G, u' \in \mathscr{C}$

That is, there is at most one directed reduction for each variable under the left-hand side of an assignment and none for other variables.

Rewording the hypothesis hidden in the definition of abstract domains, there is a family of $\mathcal{U}_a$ functions such that: – they only allocate and kill variables belonging to the domain they are part of; – constraint triggering process satisfies the relation $\succ_n$ (increasing ghostliness); – a constraint-DAG does not involve twice the same variable in a directed constraint.

Let us draw a proof sketch. We made the assumption that real assignments never use their left-hand side in their right-hand side. Thus, the tree of variables that are ghostlier than the left-hand side and the forest of variables ghostlier than variables in the right hand-side are disjoint. This has a crucial consequence: all left-hand sides of assignments may never appear in the right-hand side of a directed constraint or in a comparison. Indeed, in the example, the last directed constraint assign variables under `y` using variables under `low` and `high`.

We now have to check that an assigned variable can only be written once. Let $v \in \mathscr{V}$. We distinguish two cases:

- $v \in \mathbb{V}$. A directed constraint can only assign a ghost variable (thank to $\succ_n$). So v is only assigned by the real assignment. So, only once.
- $v \notin \mathbb{V}$. There is a variable v', a role $\mathscr{R}$ and an integer i such that $v = \mathscr{R}(v')$ and $v \in \mathscr{V}_i$. Since each directed constraint triggered by another directed constraint writes in a variable exactly one level more ghostly, an assignment to a variable in $\mathscr{V}_i$ can only be generated at the i^{th} step of recursion. Moreover, it can only come from the domain that owns the role $\mathscr{R}$. Consequently, there is only one constraint-DAG that may contain an assignment to v. As we assumed that constraint-DAGs may only assign each variable once, there is only one assignment to v.

So, variables under the left-hand side of an assignment appear at most once and only as the left-hand side of a directed constraint, and these variables were previously set to $\top$ by Assign. Hence the hypothesis on directed constraints holds.

4.6 Termination

Generally, $\mathcal{U}_a^*(u)$ is an infinite set. But it can be made finite, which is necessary to actually execute ghost constraints, under some conditions. The example shows that ghostlier and ghostlier variables are assigned from simpler and simpler expressions. The idea behind is to use a well-founded order: if ghost constraints keep being simpler, within the meaning of a well-founded notion of "simplicity", recursive exploration of constraints eventually terminates.

Let us take a look to the tree of ghost constraints. From `y = low | (high << 16)`, at the first step of recursion we got

- $\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y}) \leftarrow \mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low})$

- $\mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y}) \leftarrow \mathscr{S}lice_{[16,31]\to[0,15]}(\texttt{high})$

and at the second step

- $\mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{y})) \leftarrow \mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[0,15]\to[0,15]}(\texttt{low}))$
- $\mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[0,15]\to[16,31]}(\texttt{y})) \leftarrow \mathscr{O}\mathrm{ffset}(\mathscr{S}lice_{[16,31]\to[0,15]}(\texttt{high}))$

It's clear that the left-hand side of assignment decreases (like prescribed by the non-interference condition). Sadly, it is not enough to ensure termination: since we can add ghost variables, we may end up into adding an infinitely deep tree of ghost variables. It is also worth noticing that all new ghost variables are under the left-hand side of an assignment, which is pretty natural.

The relation we are going to define has no fundamental importance, unlike non-interference condition. Here, any well-founded relation is acceptable and the relation might be fine-tuned depending on the domains. Here, the hypothesis is global to all involved domains. Indeed, the union of well-founded relations is not necessarily well-founded, so if each domain has its own relation, it does not guarantee termination. When implementing, there are two philosophies. One can choose a reasonable relation and dictate itself to stick with it. Or, conversely, one can try to adapt the relation according to the domains. In practice, a hybrid approach should be favored since the first method can lack flexibility and the second may result in trying to use a non-terminating set of domains.

The following relation is a real-world one that covers most of the cases while still being simple. As said before, the main idea is to increase the ghostliness of assigned variables, but, to avoid allocation of an infinitely deep tree of ghost variables, we consume "complexity" of expressions (right-hand side of assignments or operand in comparisons) on allocation.

First let us define the complexity order on expressions.

Notation 5. *Let us denote $\preccurlyeq$ the order relation on $\mathscr{E}$ defined by $a \preccurlyeq b :\Leftrightarrow \mathrm{VarM}(a) \lhd_{\mathcal{M}} \mathrm{VarM}(b) \vee (\mathrm{VarM}(a) \trianglelefteq_{\mathcal{M}} \mathrm{VarM}(b) \wedge a \subseteq b)$ where $\subseteq$ is the structural inclusion, $\mathrm{VarM}(e)$ is the multiset of variables in e and $\trianglelefteq_{\mathcal{M}}$ is the multiset order induced by $\trianglelefteq$ [10]. $\prec$ denotes the corresponding strict order.*

Lemma 1. *Let V be a finite subset of $\mathscr{V}$ and $E := \{e \in \mathscr{E} \mid \mathrm{Var}(e) \subseteq V\}$. The restriction to E of $\trianglelefteq_{\mathcal{M}}$ is well-founded.*

It is a well-known result [10], since $\trianglelefteq$ is well-founded on such a set.

Lemma 2. *Let V be a finite subset of $\mathscr{V}$ and $E := \{e \in \mathscr{E} \mid \mathrm{Var}(e) \subseteq V\}$. The restriction to E of $\preccurlyeq$ is well-founded.*

It is a lexicographic order induced by two well-founded orders.

Notation 6. *Let us denote $\rightarrowtriangle$ the smallest relation on $\mathscr{U}$ that satisfies*

- $\forall (v,e) \in \mathbb{A} \uplus \mathscr{A}, \forall (l,op,r) \in \mathscr{C}, \mathrm{Var}(l) \cup \mathrm{Var}(r) \subseteq \mathrm{Var}(e)^{\trianglelefteq} \Rightarrow (v,e) \rightarrowtriangle (l,op,r)$
- $\forall ((l,op,r),(l',op',r')) \in (\mathbb{C} \uplus \mathscr{C}) \times \mathscr{C}, l' \prec l \wedge r' \prec r \Rightarrow (l,op,r) \rightarrowtriangle (l',op',r')$
- $\forall ((v,e),(v',e')) \in (\mathbb{A} \uplus \mathscr{A}) \times \mathscr{A}, v' \lhd v \wedge \left\{ \begin{matrix} e' \prec e \mathit{if}\ v\ \mathit{is\ freshly\ allocated} \\ e' \preccurlyeq e \mathit{otherwise} \end{matrix} \right\} \Rightarrow (v,e) \rightarrowtriangle (v',e')$

Theorem 2. *Let V be a finite subset of $\mathscr{V}$, $E := \{e \in \mathscr{E} \mid \mathrm{Var}\,(e) \subseteq V\}$ and $U := (E \times \mathscr{O} \times E) \uplus \{(v, e) \mid v \in \mathscr{V}, e \in E\}$. $\rightarrowtriangle$ is well-founded on U.*

This is not enough to ensures termination in all cases. We add a last constraint that has been already observed in the example: new variables are under the left-hand side of an assignment:

- $\forall c \in \mathbb{C} \uplus \mathscr{C}, \pi_1(\mathcal{U}_a(c)) = \emptyset$ (comparisons don't allocate ghost variables)
- $\forall (v, e) \in \mathbb{A} \uplus \mathscr{A}, \forall w \in \pi_1(\mathcal{U}_a((v, e))), w \lhd v$ (assignments allocate only under their left-hand side)

Finally, we need to recall an intermediate result of non-interference: the tree of ghost variables under the left-hand side of an assignment has no intersection with the trees under other involved variables. Consequently, any new variable cannot appear in a comparison or the right-hand side of an assignment. Thus, the hypotheses of theorem 2 are indeed satisfied, ensuring termination.

4.7 Support Unification

A last big part is the problem of support unification prior to binary operation. Unifying the support implies adding, killing and assigning variables so as to ensure consistency of both states. But, when a variable is allocated and reduced, it can lead to the allocation of new ghostlier variables that may be not unified. Thus, the way to proceed is to unify the support layer by layer. First, there are real variables: they are already unified since it is guaranteed by the language. The first step is to unify the ghost variables in $\mathscr{V}_1$. This may add new ghost variables in deeper layers. Then, variables in $\mathscr{V}_2$ can be unified. At this round, domains can allocate and reduce variables only in $\mathscr{V}_2$ and lower layers, but, they cannot constrain (or kill) any variable of $\mathscr{V}_1$ and higher. Thus, at the end of this step, both $\mathscr{V}_1$ and $\mathscr{V}_2$ are unified. We continue this way until all variables are unified. We simply iterate calls to $\mathcal{B}$ to unify layer $\mathscr{V}_i$ with i increasing.

Ensuring termination of this process is quite tricky. Since each round consists in applying unary operations on the two abstract states, they will all clearly terminate. But, we should still have a finite number of rounds.

Unlike regular assignments that can lead to a finite but arbitrarily high number of new ghost variables depending on the right-hand side expression, unification assignments are meant to make both states similar. The form of the forest of ghost variables can change, but its depth must stay the same. It is a reasonable constraint since adding a variable where there weren't any means guessing some information from nothing, which seems dubious.

Thus, to ensure termination, we dictate that the depth of the ghost variable tree should not increase. The depth is the maximum number of roles nested around a living variable. In other words, the depth for an abstract state a is $\mathrm{depth}(a) = \max\{i \in \mathbb{N} \mid \mathrm{supp}\,(a) \in \mathscr{V}_i \neq \emptyset\}$. At the end, the depth should not be bigger than $\max(\mathrm{depth}(a), \mathrm{depth}(b)) + 1$. It is not a natural property but a political one. It is ensured by assigning $\top$ in all variables that exceed the maximum depth. We need each domain to be able to represent the $\top$ value for a variable without using ghost variables. Actually, with reasonable domains, especially domains given in example, unification never allocate variables beyond the limit and this forced-termination protocol is never triggered.

Some unification examples will be detailed in the following.

5 Some Abstract Domains

Here are some abstract domains that benefit from this framework, or simply, that tolerate it well.

5.1 Pointers as Base + Offset

The domain described in [14] represents each pointer as a set of blocks (typically, structures or arrays) it can point to and a numeric offset inside these blocks. This domain has been reimplemented in this framework.

The old implementation use an underlying numeric domain to handle offsets. In the new implementation, the offset is a ghost variable. More precisely, the domain defines a single role $\mathscr{O}$ffset such that the offset of the variable v is stored in the ghost variable $\mathscr{O}\text{ffset}(v)$. Pointer arithmetic is translated on arithmetic computation on the offset.

```
int t[3];
int u[3];
int* p;
if (?)
  p = &t;
else
  p = &u;
if (?)
  p++;
```

For instance, at the end of the program beside, we have $\texttt{p} = \{\texttt{t}, \texttt{u}\} + \mathscr{O}\text{ffset}(\texttt{p})$ and $\mathscr{O}\text{ffset}(\texttt{p}) = \{0, 4\}$ (in a possible numeric domain).

If we look closer at the `p++;` statement. In expanded form, it is `p = p + 1;`. This violates the hypothesis that the left-hand side is not part of the right-hand side. So, internally, this statement is rewritten as `q = p + 1; p = q;` where `q` is a fresh variable. The second generated statement is a mere copy, so not very interesting. Let examine the first one.

Before this assignment we have $\texttt{p} = \{\texttt{t}, \texttt{u}\} + \mathscr{O}\text{ffset}(\texttt{p})$. The pointer domain can check that this computation is correct and has two effects: setting the base of `q` to $\{\texttt{t}, \texttt{u}\}$ and modifying the offset of `q` as $\mathscr{O}\text{ffset}(\texttt{q}) \leftarrow \mathscr{O}\text{ffset}(\texttt{p}) + \texttt{sizeof}(\texttt{int}) \times 1$.

5.2 Slices

In low-level system management, bitwise operations on pointers are sometimes mandatory. A natural example is the initialization of Global Descriptor Table (GDT) in x86 architecture. It is a structure that describes memory spaces with their base address, their size, and some miscellaneous flags. The base address is not stored contiguously in memory: each part is computed using bitwise operations (typically, shifts and bitwise and). The GDT can be used to describe the main memory, but also special structures like call gates: these are mechanisms by which a non privileged application can perform system calls. In this case, the base address is a pointer to the function to call. And so, while accessing a call gate, we must be able to reconstruct the pointer to check the call is valid and continue the analysis. This need a domain that smartly handle bitwise computations. Il should be able to keep a precise representation of variables that are cut and rebuilt with bitwise operators.

The main idea is to remember that bit slices of different variables are equal. A slice may also be 0, 1 or ⊤ (unknown). Remembering the 0 and 1 parts is necessary to handle nicely bitmasks.

For instance, with the instruction `z = x & 0xff | (y & 0xff << 16)` using the notation defined in section 4.2, the slices domain will remember that

$$\mathtt{z} = [_0\ \mathscr{S}lice_{[0,7]\to[0,7]}(\mathtt{z})[0,7]\ {}_7|_8\ 0\ {}_{15}|_{16}\ \mathscr{S}lice_{[0,7]\to[16,23]}(\mathtt{z})[0,7]\ {}_{23}|_{24}\ 0\ {}_{31}]$$

It requires two ghost variables to store the same value. With more expressive roles, we could use only one ghost to mean "this variable represents slices $[0,7]$ and $[16,23]$ of `y`". This solution may be appealing but was rejected due to the complexity it adds into algorithms while being very rarely useful. Indeed, it is uncommon to select two non-consecutive slices of the same variable (here `x`).

```
int x, y;
if (?)
  y=x&0xffff
else
  y=x&0xff
```

Let us also look at support unification. The code above a non-trivial join. At the end of the "if" branch, the 16 lower bits of `y` are those of `x`, while at the end of the "else", only the 8 lower bits are those of `x`. To compute the state after the last line, we need to join these states. But the supports are different. We have to unify the subdivisions. The first branch becomes

$$\mathtt{y} = [_0\quad \mathscr{S}lice_{[0,7]\to[0,7]}(\mathtt{y})[0,7]\quad {}_7|_8\quad \mathscr{S}lice_{[8,16]\to[8,15]}(\mathtt{y})[8,15]\quad {}_{15}|_{16}\quad 0\quad {}_{31}]$$

while in the "else" branch, we add a useless $\mathscr{S}lice_{[8,15]\to[8,15]}(\mathtt{y})$ variable equal to `x`. We can now proceed to the join. The first slice is the same, we keep it. The second slice join "0" and $\mathscr{S}lice_{[8,15]\to[8,15]}(\mathtt{y})[8,15]$ which becomes ⊤. The third slice is just "0" so we keep it. Overall

$$\begin{aligned}\mathtt{y} &= [_0\quad \mathscr{S}lice_{[0,7]\to[0,7]}(\mathtt{y})[0,7]\quad {}_7|_8\quad \top\quad {}_{15}|_{16}\quad 0\quad {}_{31}]\\ \mathtt{x} &= \mathscr{S}lice_{[0,7]\to[0,7]}(\mathtt{y}) = \mathscr{S}lice_{[8,15]\to[8,15]}(\mathtt{y})(\text{unused})\end{aligned}$$

Some ghost variables haven't been deleted but are not used anymore. They may be garbage collected after reduction.

5.3 Numeric Domains: a Singular Case

We can adapt vanilla numeric domains in this framework as a domain without role. All the ghost variables management functions are consequently trivial. This allows straightforward integration of existing numeric abstract domains to this framework.

The converse is not true: there are numerical domains that may take advantage of ghost variables. For instance, it is a way to implement signedness-agnostic domains that need to remember the unsigned value that have the same bit-representation of a signed variable, and conversely. This domain was already part of Astrée but used an ad hoc implementation trick that cannot be generalized. This domain was naturally adapted to the new framework.

5.4 Linear Combinations

In assembly, there are several kinds of jumps. Among near jumps (the simple family of jumps), there are two ways of specifying the destination: either by giving the explicit address of the target instruction, or the offset relative to the address of the current instruction. During system initialization, it might be necessary (for technical reasons) to write dynamically in the code to set such an offset computed as the difference between the destination function pointer and the address of the jump instruction (pinned with a label). Both these addresses are unknown, thus the difference must be remembered symbolically. Later, when the jump is executed and the destination computed, the current address is added to the previously computed difference. We can symbolically simplify this result, and we get the expected function pointer.

The ghost variables are the terms of the linear combination. Just like the slices domain, this domain uses ghost variables to remember values of expressions that may be rvalues, may change or whose variables might fall out of scope.

6 Conclusion

We have proposed a new product of abstract domains that handles ghost variables. It allows decentralized allocation and deletion of ghost variables, while still being shared by all domains. Moreover, it supports communication of arbitrary constraints to allow reduction of internal states, thus to improve precision. This framework has been implemented in the Astrée static analyzer along with the base-offset domain [14], slices domain and an adapter from old framework to new one to reuse all the numerical domains. This development version of Astrée has been successfully used to analyze real-world critical source code where the old framework is not expressive enough.

Some domains can be added to the current implementation, like linear combinations domain. Beyond that, though we designed this product with dynamic support to ease pointer abstraction, there is no *a priori* limitation on the abstraction level at which it can work. For instance, one can adapt this domain to make reduced product of shape abstraction domains. A product with dynamic support can indeed be a nice and modular way to implement cofibered domains. Thereby, a list-abstraction domain can seamlessly use integers as the content of the list, or any other available domain, such as another kind of data-structure.

References

1. Alur, R., Cerný, P., and Weinstein, S.: Algorithmic Analysis of Array-Accessing Programs. In: Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings, pp. 86–101 (2009). DOI: 10.1007/978-3-642-04027-6_9
2. Amato, G., Scozzari, F., Seidl, H., Apinis, K., and Vojdani, V.: Efficiently intertwining widening and narrowing. Sci. Comput. Program. 120, 1–24 (2016). DOI: 10.1016/j.scico.2015.12.005

3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), pp. 196–207. ACM Press, San Diego, California, USA (2003)
4. Bourdoncle, F.: Abstract Interpretation by Dynamic Partitioning. J. Funct. Program. 2(4), 407–423 (1992). DOI: 10.1017/S0956796800000496
5. Chang, B.E., and Leino, K.R.M.: Abstract Interpretation with Alien Expressions and Heap Structures. In: Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings, pp. 147–163 (2005). DOI: 10.1007/978-3-540-30579-8_11
6. Cortesi, A., Costantini, G., and Ferrara, P.: A Survey on Product Operators in Abstract Interpretation. In: Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013. Pp. 325–336 (2013). DOI: 10.4204/EPTCS.129.19
7. Cousot, P., and Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, New York, NY, Los Angeles, California (1977)
8. Cousot, P., and Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979, pp. 269–282 (1979). DOI: 10.1145/567752.567778
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Annual Asian Computing Science Conference, pp. 272–300. Springer, Tokyo, Japan (2006). DOI: https://doi.org/10.1007/978-3-540-77505-8. (Visited on 09/13/2016)
10. Dershowitz, N., and Manna, Z.: Proving Termination with Multiset Orderings. Commun. ACM 22(8), 465–476 (1979). DOI: 10.1145/359138.359142
11. Feret, J.: Confidentiality Analysis of Mobile Systems. In: Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings, pp. 135–154 (2000). DOI: 10.1007/978-3-540-45099-3_8
12. Halbwachs, N., and Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, pp. 339–348 (2008). DOI: 10.1145/1375581.1375623
13. Journault, M., Miné, A., Monat, M., and Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19), pp. 1–17, New York, USA (2019). http://www-apr.lip6.fr/~mine/publi/article-mine-al-vstte19.pdf, to appear
14. Miné, A.: Field-sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems. LCTES '06, pp. 54–63. ACM, Ottawa, Ontario, Canada (2006). DOI: 10.1145/1134650.1134659

15. Péron, M.: Contributions à l'analyse statique de programmes manipulant des tableaux. (Contributions to the Static Analysis of Programs Handling Arrays). Grenoble Alpes University, France (2010).
16. Platzer, A., and Tan, Y.K.: Differential Equation Axiomatization: The Impressive Power of Differential Ghosts. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, pp. 819–828 (2018). DOI: 10.1145/3209108.3209147
17. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Cousot, R., and Schmidt, D.A. (eds.) Static Analysis, pp. 366–382. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
18. Venet, A.: Automatic Analysis of Pointer Aliasing for Untyped Programs. Sci. Comput. Program. 35(2), 223–248 (1999)
19. Venet, A.: Automatic Determination of Communication Topologies in Mobile Systems. In: Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings, pp. 152–167 (1998). DOI: 10.1007/3-540-49727-7_9

A Some Details on the Implementation

The implementation works quite differently from the formal description, since partially applied functions are hard to debug. Moreover, it would be a shame to duplicate recursive calls to $\mathcal{U}$ in each domain. The implementation works as a conversation between domains and a controller that decides what to do. When a domain is asked to run a unary operation, it can return the result or a constraint-DAG and a context to pursue its computation after (we can think of it as a continuation). They are thus allowed to issue DAG several times depending on the known abstract constraints in the intermediate state.

The controller receives constraint DAGs and manages their execution. The chosen way of executing these graphs is to make all domains run each instruction at once before proceeding to the next one. This way, all abstract states stay consistent during intermediate computations, allowing mutual reduction at each ghost instruction using communication channels. Computation of ghost instructions may also issue sub-constraints DAGs, so the manager remembers a stack of contexts to handle properly recursive exploration of ghost constraints.

However, even if all instructions are run synchronously, non-interference stays crucial since the evaluation order is still arbitrary.

Roles have a slightly different meaning in the implementation. Whereas giving roles to variables is useful to formalize and understand their semantics, roles aren't really essential in the implementation. We could have the same guarantee while letting roles implicit: roles that do not appear as constructor of a sum type. However, roles allow optimizations by keeping independent flows mostly consistent and easing the unification process. But they allow optimizations by keeping independent flows mostly consistent and easing the unification process. This is only an issue of software engineering.

The controller is also in charge of sanitizing assignments so that the lvalue does not appear in the rvalue, using intermediate variables. Such variables get a special role that is handled similarly to the "Real" role (the artificial role of variables in $\mathbb{V}$). In addition, the controller (optionally) checks that assumptions on ghost instructions are indeed satisfied. A violated assumption result in an early and explanatory crash rather than an unexpected behavior that might be hellish to debug.

To adapt the many numeric domains to this framework, we use a functor that changes a simple domain to a domain that handles ghost variables but does not use them. Thus, there is no additional cost of adding a domain that does not need ghost variables in this framework compared to the former one.

Garbage collection occurs at each statement after ghost reduction. The goal is to remove useless ghost variables. There are two mechanisms:

- domains can declare which ghost variables they own are unused;
- ghost variables under a non living variable are deleted.

The former mechanism triggers the deletion of the variables concerned and, according to the latter, underlying variables are also deleted. To avoid costly iterations to find variables under a deleted variable, the first mechanism relies on domain good faith: when deleting a variable, domains spontaneously return the set of immediately ghostlier variables that have become useless. Deletion of these variables induce eventually deletion of all underlying variables, by recursive iterations. The implementation of this strategy has a negligible overhead (testing if a map is empty) when there is nothing to collect, which represents a vast majority of cases.