Deductive Verification with the Help of Abstract Interpretation

Lucas Baudin

Deductive Verification with the Help of Abstract Interpretation

Lucas Baudin

École normale supérieure

Abstract. Abstract interpretation is widely used in static analysis tools. However, to the best of our knowledge, it has not been extensively experimented in an interactive deductive verification context. This paper describes an analysis by abstract interpretation to infer loop invariants in the Why3 tool. The analysis takes advantage of the logic annotations present in the source code, including potential hand-written loop invariants. The inferred invariants are added to loops as if written by the user. They are checked by external provers and thus the analysis by abstract interpretation does not need to be trusted.
Our analysis goes beyond numerical invariants. We describe two functors that add uninterpreted functions and quantifiers to a numerical abstract domain. The resulting domain is generic enough to reason about algebraic types, Booleans, or arrays. We show that it achieves a level of expressivity comparable to that of ad-hoc domains for arrays found in the literature.

Introduction

Verifying a program against its specifications is a complex and undecidable problem in general. Several formalisms have been proposed to express and prove properties of programs. A popular one is Hoare logic along with the so-called weakest preconditions calculus. It is particularly suited for deductive verification tools.

However, in this context, loop invariants are typically required to get provable weakest preconditions. Therefore, exhibiting and proving loop invariants is a critical step to verify programs. Nonetheless, it is well known to be an undecidable problem in general. Padon proved it decidable for programs with specific operations on linked list [20], but it is undecidable as soon as the data structures are slightly more complex [14]. To the best of our knowledge, abstract interpretation has never been extensively experimented as an help to perform deductive verification. This paper shows how it is possible to use abstract interpretation to infer loop invariants and use them in a deductive verification context. We implemented this mechanism in the deductive verification platform Why3.

There are a number of relevant differences between the programming language of Why3 and languages commonly analysed by abstract interpretation. It is an ML-like language which does not distinguish expressions and statements, allowing a lighter abstract interpretation. The type system guarantees that aliases

are statically known, thus no memory model is needed. Moreover, as every function is annotated with a precondition and a postcondition, the analysis can be intraprocedural. It is therefore modular and can afford to be more costly.

The Why3 tool generates verification conditions that have to be verified by external provers. Invariants generated via abstract interpretation are not different from user-provided ones and are also checked. Thus, the code that perform the analysis by abstract interpretation does not need to be trusted. The inferred invariants are then added to the other verification conditions, which can help provers to check them. Therefore, there is no need for our analysis to emit alarms when a safety property may not hold: a verification condition will be generated anyway, and may or may not be proven.

Why3 targets safety properties as well as functional properties. The latter pose interesting challenges as they typically require much more sophisticated invariants than those needed for safety properties. Therefore, we describe two functors to extend a classical numerical domains with uninterpreted functions and quantifiers. Invariants inferred with the resulting domain are sufficient to prove properties on programs with arrays, algebraic types, or Booleans.

The first section briefly introduces the Why3 platform. Section 2 explains the analysis and points out the major differences with regard to standard abstract interpretation. Then, Sec. 3 describes two functors that, when used together, can be used to infer properties on complex data types such as arrays. Section 4 gives implementation details and experimental results.

1 Deductive Verification with Why3

Why3 [1] is a platform for deductive program verification. It provides two languages. One is called WhyML and is used to write programs. It belongs to the ML language family with a syntax close to that of OCaml and supports numerous features, including algebraic data types, mutable record types, exceptions, effectful functions, and imperative programming structures (`for` and `while` loops). The other language is used to write logic: theorems, lemmas, axioms, function contracts, and loop invariants. It is based on an extension of first-order logic [9].

Programs written in WhyML are annotated with specifications written in the logic language. These specifications are expressed in Hoare logic [15], *i.e.* functions are given a precondition and a postcondition. The scope of the specifications is up to the user: they can only be about safety properties (such as an array access) or functional correctness. Why3 generates verification conditions that guarantee that the specifications are met by the code of the program. They are checked by external provers: automatic solvers (SMT or ATP based solvers) or interactive theorem provers (Coq, PVS or, Isabelle/HOL).

The logic language can also be used on its own. Several other tools, such as Frama-C [19] and SPARK, directly generate verification conditions in this language and benefit from the integration of the wide range of theorem provers in Why3.

As an example, this WhyML program creates an array of size `n` and initializes the cell `i` with the value `i`:

```
let identity (n: int) : array int
    requires { n >= 0 }
    ensures  { forall i: int. 0 <= i < n -> result[i] = i }
    =
        let a = Array.make n 0 in
        let k = ref 0 in
        while !k < n do
            variant   { n - !k }
            invariant { 0 <= !k }
            invariant { forall i: int.
                            0 <= i < !k -> result[i] = i }
            a[!k] <- !k;
            k := !k + 1;
        done;
        a
```

Listing 1. Initializing an Identity Array.

The postcondition (resp. the precondition) of the function is written with `ensures { ... }` (resp. `requires { ... }`). A loop variant must be supplied to prove termination. Here, a loop invariant is needed to prove the postcondition of the function.

All data types are immutable, apart from record fields that are explicitly marked as mutable, as in OCaml. A reference is a record with a single, mutable field. An array is a record with two fields: its length (which is immutable), and a map from the indices to the content of every cell of the array. These maps are not implemented but axiomatized, which is sufficient to reason about arrays. If one wish to run the program, it can be extracted to OCaml and then WhyML arrays are mapped to native OCaml arrays.

Verification conditions (VCs) are generated to ensure that a function satisfies its postcondition and satisfies the precondition of the functions that it calls. This is done by computing weakest preconditions [23]. The resulting VCs are formulas expressed in the logic language of Why3.

In Why3, weakest preconditions are computed with an intraprocedural analysis. As a consequence, if the postcondition of a function is not precise enough, then the weakest precondition can not be proven at the call site. For instance, if one uses the function identity with `identity(50)[42]`, the safety of the array access can not be proven: `identity` is under-specified and its postcondition does not guarantee that the length of the returned array is `50`. Thus, a verification condition can be unprovable even if the property that we are trying to prove is true.

An analogous problem occurs with `for` and `while` loops. The traditional calculus of weakest preconditions requires that the user provides a loop invariant. An empty invariant is often not sufficient to prove safety properties or a function postcondition.

Let us have a closer look at the loop of the `identity` function. Without the loop invariants, the weakest precondition is:

$$\forall n.\ \underbrace{n \geq 0}_{\text{pre.}} \Rightarrow \forall a\, \forall k.\ \underbrace{a.\text{length} = n}_{\texttt{Array.make post.}} \Rightarrow \underbrace{!k \geq n}_{\text{loop exit cond.}} \Rightarrow \underbrace{\forall i.\ 0 \leq i < n \Rightarrow a[i] = i}_{\text{post.}}$$

This can not be proven. We need an additional invariant, such as:

$$I(a, k) \equiv \forall i.\ 0 \leq i < !k \Rightarrow a[i] = i$$

Then, the weakest precondition becomes:

$$\forall n.\ \underbrace{n \geq 0}_{\text{pre.}} \Rightarrow \forall a \forall k.\ \underbrace{a.\text{length} = n}_{\texttt{Array.make post.}} \Rightarrow \underbrace{!k \geq n}_{\text{loop exit cond.}} \wedge \underbrace{I(a, k)}_{\text{loop inv.}} \Rightarrow \underbrace{\forall i.\ 0 \leq i < n \Rightarrow a[i] = i}_{\text{post.}}$$

This one can be proven. Of course, the invariant must also be proven. Two additional VCs are generated to do so: the invariant must be verified before the loop and then be preserved by any loop step. For instance, the preservation VC has the form $I(a', k') \wedge \phi(a', k', a, k) \Rightarrow I(a, k)$. Variable a' (resp. k') is the value of a (resp. k) at the previous iteration, and ϕ is a formula obtained through the weakest precondition calculus on the loop. Here, among other things, ϕ expresses the fact that a is the same array as a' but for the $!k'$-nth element. In Why3, a loop invariant must hold before the loop test, which means that it is valid when the loop is exited, even if the code inside the loop is never executed (if the loop condition is always `false`).

To completely verify this function, one must also prove that there is no out-of-bounds array writes. The VC associated to this property can not be proven without another loop invariant such as $0 \leq !k$. The next section describes an analysis via abstract interpretation to find numerical invariants such as this last one. Section 3 augments this analysis and shows how the first one can be inferred as well.

2 Abstract Interpretation for WhyML

Abstract interpretation is a well-established framework [4], which has already been widely used to verify properties on various kinds of programs. This section introduces this framework for WhyML. We emphasize that this language is different from languages abstract interpretation is commonly applied to. It is a high-level language featuring rich types, with no difference between statements and expressions. Furthermore, we are in a context of deductive verification, thus we want to take advantage of the logic annotations present in the source code.

2.1 Abstract Domains

Abstract interpretation is a process that soundly approximates the semantics of a program [4]. An abstract interpreter computes abstract states that are approximations of memory states.

Multiple abstract domains have been proposed. Striking a balance between precision of these abstractions and cost of operations on abstract domains remains a challenge. It is especially the case for huge codebases if the analysis done by abstract interpretation is not modular.

An abstract domain is defined as an approximation of a concrete domain, which represents memory states. For several reasons explained throughout this article, our concrete domain is a set of formulas. That may seem rather unusual. Yet these formulas can be seen as set of states nonetheless. A formula ϕ corresponds to the set of memory states $\mathcal{M}$ such that $\mathcal{M} \vDash \phi$. For instance, the formula $x = 1 \wedge y >= 0$ represents the memory states $\{x \mapsto 1, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}$, etc.

More precisely, let $\mathbb{V}$ be a set of variables, fixed before the analysis. It contains at least variables of the program, but it also comprises additional variables such as fields of records. Then, our concrete domain $\mathbb{D}$ is the set of quantifier-free formulas over variables in $\mathbb{V}$. We use the implication as an order relation:

$$\forall\, (a, b) \in \mathbb{D}.\, a \subseteq b \overset{def}{\Leftrightarrow} a \to b$$

An abstract domain $\mathbb{D}^\sharp$ is defined with an abstraction function $\alpha : \mathbb{D} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ and a concretization function $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$. It is also parameterized by the set of variables $\mathbb{V}$, and provides abstract operations given in Fig. 1. In the literature, an abstraction function maps a set of concrete states to an abstract state. Here, an abstraction function maps a first-order formula (that represents a set of memory states) and an abstract state to an abstract state that intuitively represents their intersection. The usual abstraction function is therefore $\alpha(\cdot, \top)$. We define the abstraction of a formula ϕ as the abstract state $\alpha(\phi, \top)$. As we will demonstrate it later, α is a generalization of what is usually called a transfer function. Functions α and γ must be monotonic and satisfy:

$$\forall d \in \mathbb{D}^\sharp. \forall a \in \mathbb{D}.\, a \wedge \gamma(d) \subseteq \gamma(\alpha(a, d))$$

$$\forall b, d \in \mathbb{D}^\sharp.\, \alpha(\gamma(b), d) \sqsubseteq b \text{ and } \alpha(\gamma(b), d) \sqsubseteq d$$

If $d = \top$ these constraints becomes $a \subseteq \gamma(\alpha(a, \top))$ and $\alpha(\gamma(b), \top) \sqsubseteq b$, *i.e.* γ and $\alpha(\cdot, \top)$ form a Galois connection.

Example: Polyhedra. The polyhedra abstract domain was originally proposed by Cousot and Halbwachs [5] to deal with conjunctions of linear equalities. Operations on this domain have an exponential worst case complexity. However, it is more precise than most other domains such as intervals or octogons where operations are less costly. Thus, it proved to be useful on many of our examples. This domain belongs to the kind of numeric conjunctive domains, meaning that a list of inequalities can be extracted from an abstract state. Thus, the concretization of this abstract state is the conjunction of these inequalities.

Let $\phi \equiv x > y \wedge (0 < y < 5 \vee 7 < y < 10)$. The abstraction of this formula in the polyhedra domain is done inductively of the formula. The abstraction of the atom $x > y$ is $\alpha(x > y, \top) = [\![y - x < 0]\!]$. Then we compute an abstraction for

$$
\begin{array}{rl}
\text{partial order } \sqsubseteq\colon & \mathbb{D}^\sharp \times \mathbb{D}^\sharp \\
\text{top, bottom } & \top, \bot : \mathbb{D}^\sharp \\
& \forall\, a \in \mathbb{D}^\sharp.\ \bot \sqsubseteq a \sqsubseteq \top \\
\text{join operator } & \sqcup : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp \\
& \forall\, a, b, c \in \mathbb{D}^\sharp.\ (c \sqsubseteq a \vee c \sqsubseteq b) \Rightarrow c \sqsubseteq a \sqcup b \\
\text{widening operator } & \triangledown : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp \\
& \forall\, a, b \in \mathbb{D}^\sharp.\ a \sqcup b \sqsubseteq a \triangledown b \text{ and there exists no infinite} \\
& \text{increasing chain } y_1, y_2, \ldots \text{ such that } y_{n+2} = y_n \triangledown y_{n+1} \\
\text{forget operator } & \mathit{forget} : \mathbb{V} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp \\
& \forall\, a \in \mathbb{D}^\sharp.\ a \sqsubseteq \mathit{forget}(v, a) \\
& \text{and } v \text{ is not a free variable of } \gamma(\mathit{forget}(v, a))
\end{array}
$$

Fig. 1. Operations on Abstract Domains.

every term of the disjunction: $\alpha(0 < y < 5, [\![y - x < 0]\!]) = [\![y - x < 0; 0 < y < 5]\!]$ and $\alpha(7 < y < 10, [\![y - x < 0]\!]) = [\![y - x < 0; 7 < y < 10]\!]$. Finally, we must join those two domains to get the abstraction of ϕ:

$$
\begin{aligned}
\alpha(\phi, \top) &= [\![y - x < 0; 0 < y < 5]\!] \sqcup [\![y - x < 0; 7 < y < 10]\!] \\
&= [\![y - x < 0; 0 < y < 10]\!]
\end{aligned}
$$

Thus, $\gamma(\alpha(\phi, \top)) = x > y \wedge 0 < y < 10$.

More generally, abstraction functions compute domains inductively on formulas. Disjunctions (resp. conjunctions) correspond to the join operator (resp. to the composition of abstraction). Atoms of the formulas are handled differently on different domains: for instance, some numerical domains such as intervals or octogons do not support every linear equalities. When an atom is not handled by the abstraction function (non linear equalities, non numeric equality, etc.), then it returns $\top$.

Disjunctive Completion. As soon as the formula to abstract contains a disjunction or an implication, a conjunctive domain is not sufficient to express it. To make the analysis more precise, the abstract domain needs to support disjunctions. This can be done by making the disjunctive completion of existing domains [6]. In our implementation, this completion is available as a functor for existing abstract domains.

An abstract state of the disjunctive completion of an abstract domain $\mathbb{D}^\sharp$ is a list of abstract states of $\mathbb{D}^\sharp$. For instance, the formula $0 < x \Rightarrow 0 < y$ is abstracted to $\top$ in the polyhedron abstract domain. However, in the disjunctive completion of this domain, it is abstracted to $\{[\![0 \geq x]\!], [\![0 < y]\!]\}$. The concretization of this abstract state is the formula $0 \geq x \vee 0 < y$ and is equivalent to the orignal formula.

$\langle expr \rangle$ ::= '`let`' $\langle variable \rangle$ '`=`' $\langle expr \rangle$ '`in`' $\langle expr \rangle$
| $\langle variable \rangle$ '`<-`' $\langle variable \rangle$
| constant
| '`while`' $\langle expr \rangle$ '`do`' $\langle expr \rangle$ '`done`'
| '`if`' $\langle variable \rangle$ '`then`' $\langle expr \rangle$ '`else`' $\langle expr \rangle$

Fig. 2. Grammar of the subset of WhyML considered.

2.2 Abstract Semantics

The proofs of programs written in WhyML is performed in an intermediate, simpler language whose grammar is given in Fig. 2. The representation of a program in this language is obtained by introducing new variables (*a.k.a.* proxy variables) using `let` for every argument of assignments (and as we will see next, function calls) and conditions. This representation allows us to give a simpler definition of our abstract semantics.

WhyML supports some imperative traits such as loops and mutable variables. Aliases between those mutable variables are statically controlled and are part of the type system [10]. Memory locations are identified using singleton regions. Two mutable variables are either known to have different regions, or known to have the same one. Thus, it is sufficient for correctness (with regard to the aliases) to have a unique variable in the underlying abstract domain for every region.

Let **result** be a new variable different from the other variables of the program. This variable is used to specify the abstract value of an expression.

$$
\begin{aligned}
[\![\texttt{let } v \texttt{ = e in e'}]\!]^\sharp(d) &= \mathit{forget}(v, [\![\texttt{e'}]\!]^\sharp(\mathit{forget}(\mathbf{result}, \alpha(v = \mathbf{result}, [\![\texttt{e}]\!]^\sharp(d))))) \\
[\![v' \texttt{ <- } v]\!]^\sharp(d) &= \alpha(v' = v, \mathit{forget}(v', d)) \\
[\![\text{constant}]\!]^\sharp(d) &= \alpha(\mathbf{result} = \text{constant}, d) \\
[\![\texttt{while e do e' done}]\!]^\sharp(d) &= \mathit{forget}(\mathbf{result}, \alpha(\mathbf{result} = \mathit{false}, [\![e]\!]^\sharp(d_{loop}))) \\
&\text{with } d_{loop} = \mathit{fp}_d(\lambda d'.d' \sqcup [\![e']\!]^\sharp(\mathit{forget}(\mathbf{result}, \alpha(\mathbf{result} = \mathit{true}, [\![e]\!]^\sharp(d'))))) \\
&\text{where } \mathit{fp}_d(f) \text{ is a fixpoint greater than } d \text{ of a function } f \\
[\![\texttt{if } v \texttt{ then e else e'}]\!]^\sharp(d) &= [\![e]\!]^\sharp(\alpha(v = \mathit{true}, d)) \sqcup [\![e']\!]^\sharp(\alpha(v = \mathit{false}, d))
\end{aligned}
$$

Fig. 3. Abstract Semantics.

The abstract semantics of our subset of WhyML is described in Fig. 3. As there is no difference between expression and statements in ML-like languages, branching in conditions is done according to the value of a variable. Thus, the abstract domain chosen should support Boolean values, which has already been studied [17]. An alternative to support Boolean values as a sum data type is described in the next section.

The functions α and *forget* used together are used to express every abstract transfer functions. The proof of the correctness of this abstract semantics with regard to a concrete semantics of WhyML is not specific to an abstract domain.

To keep proofs small, we assume that the fp_d operator is monotonic in d and that $fp_d(f) = d$ if d is a fixpoint of f. For instance, it can be implemented with Kleene's iterations. To ensure termination, the join operator can be replaced by a widening operator after a fixed number of iterations.

The following theorem expresses the soundness of the analysis. The proof of the theorem, as well as details about the big-step semantics used and the satisfiability relation, can be found in Appendix C.

Theorem 1 (Soundness). *Let $\mathcal{M}$ be a memory state, e a WhyML expression whose execution terminates, and d an abstract domain. If $\mathcal{M} \vDash \gamma(d)$, then:*

$$[\![e]\!](\mathcal{M}) \vDash \gamma([\![e]\!]^\sharp(d))$$

Therefore, if a memory state $\mathcal{M}$ satisfies the precondition of a function, then the memory state after the execution of the function satisfies the abstract state computed starting from the abstraction of the precondition.

Corollary 1. *Let $\mathcal{M}$ be a memory state, e a WhyML expression whose execution terminates, and ϕ a first-order formula. If $\mathcal{M} \vDash \phi$, then:*

$$[\![e]\!](\mathcal{M}) \vDash \gamma([\![e]\!]^\sharp(\alpha(\phi, \top)))$$

2.3 Loop Invariants

The analysis of a WhyML function is carried out by computing its abstract semantics, starting with the abstraction of the precondition of this function. That is, if the expression and the precondition of a function are `e` and ϕ, then the analysis computes the abstract domain $[\![\mathtt{e}]\!]^\sharp(\alpha(\phi, \top))$. This computation is decomposed into the computation of the abstract domains resulting from the subexpressions. The abstract domains that correspond to the content of loops, that is the abstract domains called d_{loop} in the abstract semantics, are extracted. Theorem 1 ensures that their concretization are loop invariants.

2.4 Function Calls

WhyML annotations allow the user to specify a postcondition and describe effects for every function. We have already seen assignments; they can be seen as elementary functions: for an assignment `a <- v`, the associated postcondition is $a = \mathtt{v}$ while the effect is writing in `a`.

An example worth explaining are references (*i.e.* pointers to locations in memory) and their common functions `(!)` (access to the content of a reference), `(:=)` (set a value), and `incr` (increment an integer reference). Figure 2[1]

[1] The `writes{...}` annotations are actually not necessary, as they are inferred automatically. They are added here for clarity.

shows the implementations of those functions written in WhyML. As in OCaml, a reference is a record with a single, mutable field, named `contents` in WhyML.

```
let (!) (r: ref 'a) : 'a
  ensures { result = r.contents }
  = r.contents

let (:=) (r: ref 'a) (v:'a) : unit
  ensures { !r = v }
  writes { r }
  = r.contents <- v

(* increment an integer reference *)
let incr (r: ref int) : unit
  ensures { !r = old !r + 1 }
  writes { r }
  = r := !r + 1
```

Listing 2. Code for References.

Those functions have simple postconditions that describe their effect. The return value is specified with the keyword `result`. While the specification of `(!)` is straightforward, the specifications of `(:=)` is more complex: there is an additional annotation, `writes { ... }`, which specifies that there is a side effect in `r`. Notation `old !r` appears in the postcondition of `incr` and refers to the value of `r.contents` before the function call, while `!r` refers to the value of the reference once the function returns to the caller.

Figure 4 gives the semantics of a function call. In the intermediate AST we work with, every argument is a variable, and reference to `old` variables are encoded as ghost variables and extra arguments. For example, let us take the expression `incr i`. It is translated to `let a = { i } in incr a i` as soon as it is parsed, meaning that `i` is copied to `a` and then `a` is used as an extra argument for `incr`. Let us suppose that before evaluating this expression, the domain is $[\![!i \geq 0]\!]$. Then, it is first transformed into $[\![!i \geq 0; !a = !i]\!]$, then, as $!i$ is written by `incr`, it must be forgotten and the domain is $[\![!a \geq 0]\!]$. As the postcondition of `incr` as been changed to $!i = !a + 1$, the resulting domain is $[\![!i = !a + 1; !a \geq 0]\!]$. The final domain, once `a` has been forgotten is, $[\![!i \geq 1]\!]$, as expected.

$$[\![\mathtt{g}(v_1, v_2, \ldots, v_n)]\!]^\sharp(d) = \alpha(t_{\mathtt{g}}[(a_1, \ldots, a_n) \leftarrow (v_1, \ldots, v_n)], \mathit{forget}((b_i), d))$$
where $t_{\mathtt{g}}(a_1, \ldots, a_n)$ is the postcondition of `g`,
and (b_i) are the variables written by `g`

Fig. 4. Abstract Semantics for Function Calls.

2.5 Increasing Precision with User-Written Invariants

As mentioned above, WhyML programs are annotated with logic formulas. We just explained how we built on postconditions to interpret function calls. Hand-written loop invariants and assertions (which may be needed to help automatic provers [3]) are also present in the source code and can be used to improve the analysis. Every annotation is added to the AST as a ghost expression of type unit. We give `assert`$\{t\}$ or `invariant`$\{t\}$ the abstract semantics $\lambda d.\, \alpha(t, d)$. This feature is especially useful when a needed widening is hard to find automatically: a user-written invariant can be used to narrow the abstract states.

It is safe to do so, as those invariants will also have to be proved by external provers. Of course, some of those proof may use invariants inferred via abstract interpretation, but there is no circular dependency between them. In our implementation, the generated invariants are added alongside user-written invariants, thus they do not have any special role compared to the user-written ones. So, doing this is safe as long as Why3 is safe.

WhyML also supports more complex control structures, such as `for` loops, exceptions (that can convey values) and pattern matching. They are implemented in a standard fashion in the control flow graph.

3 Uninterpreted Functions and Quantifiers

We now introduce two functors that add uninterpreted functions and quantifiers to a numerical abstract domain. They are useful to express properties on non-linear arithmetic, arrays, lists, etc. These two functors are generic enough to express a broad range of properties. For instance, uninterpreted functions on arrays allow the use of the `get` function and thus expressing constraints on the value of an array at a particular index. Quantifiers extend this by allowing constraints to be expressed on ranges of indices. Furthermore, algebraic types can also be characterized with these new domains.

Of course, operations on them are not as precise or fast as a dedicated, ad-hoc domain for every data structure. However we show in the end of this section that our domain is at least as expressive as some of the proposed dedicated domains for arrays.

3.1 Uninterpreted Functions

In what follows, we describe how to augment a numerical domain with uninterpreted functions. It is carried out by doing the reduced product between the base domain and a union-find structure. Our abstract domain is similar to that of Chang and Leino [2]. However, we assume here that we already have a union-find structure on terms with operations given in Fig. 3.1. We also assume that the order relation $\sqsubseteq_{\mathcal{U}}$ is such that there is no infinite increasing chain. This is reasonable, as, intuitively, a union-find greater than another one has less equalities. We assume we already have an abstract domain $\mathbb{D}^\sharp$ with the associated α, γ

functions, and operators. We now define a new abstract domain $\mathbb{D}^\sharp_{UF}$ on a set of terms $\mathcal{T}$.

We bound the number of variables that we add to $\mathbb{V}$ to represent uninterpreted terms. Bounding the number of those variables is important, as creating a potentially infinite number of variables could break convergence properties. The finite set of those variables is written $\mathbb{V}_{UF}$.

Abstract states of $\mathbb{D}^\sharp_{UF}$ are made of an abstract state of $\mathbb{D}^\sharp$, a union-find on terms, and a mapping from variables in $\mathbb{V}_{UF}$ to terms. That is, $\mathbb{D}^\sharp_{UF}$ is formally defined as $\mathbb{D}^\sharp \times \mathcal{U} \times \mathcal{V}$ where $\mathcal{U}$ is the set of union-find structures and $\mathcal{V}$ is the set of partial bijections between $\mathcal{T}$ and $\mathbb{V}_{UF}$. A union-find is represented with its equivalent classes (so $\{\}$ is a union-find with no terms equal) and elements of $\mathcal{V}$ are represented by the variable-to-term assocations, *i.e.* $\{x \mapsto v\}$ is the partial bijection that associates the term x to the variable v.

Abstraction of a formula with uninterpreted functions needs only to be defined on atoms as said precedently. If the atom is an equality between two non-integer terms, then the resulting abstract states has those two terms in the same equivalent class, that is $\alpha_{UF}(x = y, (d, u, v)) = (d', \mathsf{union}(x, y, u), v)$, where d' is the abstract states with the equalities discovered in the union-find by the union operation. If the atom is anything else, then the uninterpreted terms are replaced on-the-fly by unused variables of $\mathbb{V}_{UF}$ and the resulting domain is made of the abstraction of the formula of the underlying domain, the empty union-find (or if the atom is an equality, a union-find whose only equivalent class are the two members of the equality) and the term-variables partial bijections built during the abstraction. If there are not enough variables in $\mathbb{V}_{UF}$, then the atom is abstracted to $\top$.

The concretization of an abstract state with uninterpreted functions is defined as the conjunction of the concretization of both the union-find and the state of the underlying numeric domain. Variables that correspond to terms must be replaced accordingly.

Before every binary operation on two states, the latter have to be made consistent with regards to their correspondence between terms and variables via variable renaming in the underlying domain. This operation is called consistent : $\mathbb{D}^\sharp_{UF} \times \mathbb{D}^\sharp_{UF} \to \mathbb{D}^\sharp_{UF} \times \mathbb{D}^\sharp_{UF}$. It should not modify the concretization of the abstract domain. Thus, we can assume that these two domains have the same variable mapping when writing the operators.

Operators are given in Fig. 6. To forget a term, variables that depend on this term in the variable mapping are forgotten in the abstract states. They are also removed from the variable mapping. The term is also remove from the union-find with the $\mathsf{forget}_{\mathcal{U}}$ operator.

Theorem 2 (Correction). *The operators defined for $\mathbb{D}^\sharp_{UF}$ are correct with regards to the requirements expressed in Sec. 1.*

Proof. See Appendix A.

Machine Integers. The default integer module of Why3 is a module of arbitrary sized integers. Why3 does not provide any hardcoded support for machine in-

$$
\begin{aligned}
&\text{set of union-find } \mathcal{U}\\
&\qquad\qquad \mathsf{union}\colon \mathcal{T}\times\mathcal{T}\times\mathcal{U}\to\mathcal{U}\\
&\qquad\qquad \mathsf{find}\colon \mathcal{T}\times\mathcal{U}\to\mathcal{T}\\
&\qquad\qquad \mathsf{forget}_{\mathcal{U}}\colon \mathcal{T}\times\mathcal{U}\to\mathcal{U}\\
&\qquad\qquad \sqcup_{\mathcal{U}}\colon \mathcal{U}\times\mathcal{U}\to\mathcal{U}\\
&\qquad \text{order relation } \sqsubseteq_{\mathcal{U}}
\end{aligned}
$$

Fig. 5. Union-Find on Terms.

$$
\begin{aligned}
\text{variable mapping} &\quad \mathcal{V} = \text{ partial bijections } \mathcal{T}\leftrightarrow\mathbb{V}_{UF}\\
\text{abstract states} &\quad \mathbb{D}^\sharp\times\mathcal{U}\times\mathcal{V}\\
\text{join} &\quad (d,u,v)\sqcup_{UF}(d',u',v) = (d\sqcup d', u\sqcup_{\mathcal{U}} u', v)\\
\text{widening} &\quad (d,u,v)\nabla_{UF}(d',u',v) = (d\nabla d', u\sqcup_{\mathcal{U}} u', v)\\
\text{top} &\quad (\top,\{\},\{\})\\
\text{bottom} &\quad (\bot,\{\},\{\})\\
\text{order} &\quad (d,u,v)\sqsubseteq(d',u',v) \iff d\sqsubseteq d'\wedge u\sqsubseteq_{\mathcal{U}} u'
\end{aligned}
$$

Fig. 6. Abstract States of the Uninterpreted Function Domain.

tegers. Instead, they are axiomatized in a module. There is an abstract type that represents them along with functions to perform operations on them. The module also contains a logic coercion function `to_int`. It converts machine integers to unbounded integers. Functions that correspond to operations on machine integers have postconditions that use the coercion function to express the result of the operation. They also have preconditions to ensure that there is no unexpected behavior such as an overflow.

An example program that manipulates machine integers can be obtained by replacing every int by int32 in the identity function of the Sec. 1. However, the analysis will be carried out on the terms `Int32.to_int n`, `Int32.to_int !k`. The coercion function is seen as an uninterpreted function, and those terms are handled by the domain. The function call `!k + Int32.one` has the postcondition `Int32.to_int !k + Int32.to_int Int32.one = Int32.to_int result`. The computed loop invariant is then `Int32.zero <= Int32.to_int !k <= Int32.to_int n`. However, this addition also has a precondition, which is `Int32.to_int !k + Int32.to_int Int32.one <= Int32.MAX_INT`. An additional verification condition is generated to ensure that this precondition is met. As the loop condition is `Int32.to_int !k < Int32.to_int n` and that `Int32.to_int n <= Int32.MAX_INT` holds, it can easily be verified by an automatic theorem prover.

Thus, the functor that adds uninterpreted functions makes it possible to verify programs that use machine integers. The analysis generates invariants that contain the coercion function. The safety of the `identity` function with machine integers can be entirely verified, up to the absence of overflows.

Algebraic Data Types. WhyML supports algebraic data types and, in particular, sum types with several constructors. Constructors are pairwise distinct, *i.e.* if we

declare `type a = A | B`, then $\neg$ (`A` = `B`). Thus, when a class of the union-find contains two different constructors of the same type, the whole domain can be reduced to $\bot$. A direct application of this feature is the support for Booleans:

```
type bool = True | False
```

Let a, b be two Boolean variables. Formula $a \wedge \neg b \equiv a = \mathit{true} \wedge b = \mathit{false}$ can then be represented exactly by our domain, with $(\top, \{\{a, \mathit{true}\}, \{b, \mathit{false}\}\}, \{\})$. Similarly, formula $a \wedge b \vee \neg a \wedge \neg b$ is represented by $\{\top, \{\{a, b\}\}, \{\}\}$. This last example shows that this domain is relational for Boolean values.

3.2 Quantifiers

In this subsection, we introduce a functor that adds support for a universal quantifier to an existing abstract domain. Generalization for an arbitrary but fixed number of quantifiers (including existential quantifiers) is straightforward. In practice, this number can be chosen by computing the maximum number of quantifiers in the logic annotations found in the program (including the post-conditions of every function called). For instance, postconditions for arrays are given in Listing 3 and only contain a single quantifier.

```
type array 'a =
  { length: int; mutable elts: map int 'a; }

val set (a: array 'a) (i: int) (v: 'a) : unit
  writes { a}
  requires { 0 <= i < length a}
  ensures {
    forall k: int.
      (0 <= k < length a ->
        (i <> k -> a[k] = (old a)[k] /\
        (i =  k -> a[k] = v)) }

val make (n: int) (v: 'a) : array 'a
  requires { n >= 0 }
  ensures {
      length result = n /\
      forall k: int.
        (0 <= k < length result -> result[k] = v) }
```

Listing 3. Operations on Arrays.

Dealing with quantifiers is known to be a hard problem. While it is in principle decidable for Presburger arithmetic via quantifier elimination, even SMT solvers do not necessarily implement a complete algorithm as it is very costly [8]. In presence of uninterpreted functions, it is undecidable [22]. Thus, our abstract domain is designed to perform strong approximations.

Our new abstract domain does not instantiate quantifiers, nor create them. It only transports quantifiers found in logic annotations. As we do not try to emit

alarms when an assertion is not verified, this is not limiting. Checking assertions (and the instantiation problem) is left to external provers.

This functor can be used on top of the uninterpreted functions functor described previously. Thus, quantified postconditions about arrays (such as the one in Listing 3) can be precisely represented in our abstract domain. Therefore, it is possible to infer non-trivial loop invariants for functions dealing with arrays. For instance, to prove the postcondition of the `identity` function described in Sec. 1, an invariant about the array content is necessary. The invariant inferred by abstract interpretation with quantifiers and uninterpreted functions is:

$$\forall i.\, (0 < i < !k \Rightarrow a[i] = i) \wedge (!k \leq i < x \Rightarrow a[i] = 0)$$

Term $a[i]$ is an uninterpreted function. In fact, it is syntactic sugar for `get a.elts i`. The field `elts` of `a` is a map. Maps are axiomatized with `get` and `set` functions. The quantified variable i comes from the postcondition of the array initialization and write operations, as explained in what follows.

We suppose we have an abstract domain $\mathbb{D}^\sharp$ with the operators of Sec. 1. We now define a new abstract domain $\mathbb{D}^\sharp_\forall$, with the concrete domain being $\mathbb{D}_\forall = \mathcal{F}_\forall$, the set of formulas in prenex form with a single universal quantifier. A new variable ω is added to express constraints on the quantified variable. Thus, the abstract domain $\mathbb{D}^\sharp$ is now parametrized by $\mathbb{V}_\forall = \mathbb{V} \cup \{\omega\}$. The concretization $\gamma_\forall$ is defined as follows:

$$\gamma_\forall(d) = \forall\omega.\gamma(d)$$

To abstract a formula, we need to instantiate the quantified variables to ω and then proceed to the abstraction in $\mathbb{D}^\sharp$:

$$\alpha_\forall(\forall i.\, \psi(i), d) = \alpha(\psi[i \leftarrow \omega], d)$$
$$\alpha_\forall(\psi, d) = \alpha(\psi, d) \text{ when no quantifiers appear in } \psi$$

Operators on $\mathbb{D}^\sharp_\forall$ are the operators of $\mathbb{D}^\sharp$. As it can be seen in the following theorem, introducing a quantified variable does not imperil the correctness properties.

Theorem 3 (Correction). *The operators defined for $\mathbb{D}^\sharp_\forall$ are correct with regards to the requirements expressed in Sec. 1.*

Proof. Trivial using the properties of the abstract domain $\mathbb{D}^\sharp$.

4 Experimental Results

Our analysis has been implemented as a Why3 plugin. The abstract domains are based on the APRON domains [16] as well as the ELINA domains [21]. The algorithms transforms a WhyML function into a control flow graph, and then uses the Fixpoint library to compute the abstract domains at every point of the source code. The loop invariants are then extracted from the edges that correspond to beginning of loops.

We applied our method to several existing programs of the Why3 gallery[2]. The result are summarized in Fig. 7. Inferred invariants are invariants that can be removed from the source code and be inferred automatically. It can be surprising to notice that on some examples, intervals perform better than polyhedron-based domains. This is caused by the widening operator: the polyhedron one is more sophisticated than the interval one, and does not keep the same inequalities.

Module	User Invariants	Inferred	Polyhedron	Intervals
Arm.M	8	3	2	1
AssigningMeaningsToPrograms	4	2	2	2
BinarySearch	2	1	1	0
BinarySearchAnyMidPoint	2	1	1	0
BinarySearchInt32	3	2	2	1
CheckingALargeRoutine	5	2	1	1
CoincidenceCount	3	2	2	0
Conjugate	8	2	2	0
Decrease1	2	1	1	0
Euler002	8	2	2	2
Ewd673	2	2	2	2
Fibonacci	24	8	3	6
FoVeOOS 2011 - 1	2	1	1	0
Flag	6	2	2	0
Fact	1	1	1	0
InsertionSort	5	1	1	0
McCarthy	2	2	2	1
MergeSort	7	2	2	0
Mjrty	6	2	2	1
Muller	4	2	2	2
RemoveDuplicate	5	1	1	0
Sieve	7	4	2	2
RelaxedPrefix	5	4	3	1
TwoWaySort	3	2	2	1
WhiteAndBlackBalls	2	1	1	0

Fig. 7. Experimental Results.

5 Related Work

Code analysis with abstract interpretation is widespread in static analysis tools, such as Astrée [7]. These tools typically emit alarms when the analysis is not sufficient to prove the desired properties. On the contrary, our analysis is not responsible for these alarms. It only produces invariants for the Why3 platform. Moreover, static analysis tools usually target safety properties such as array

[2] http://toccata.lri.fr/gallery/why3.en.html

bounds checking. The Why3 platform targets safety as well as functional properties, which means that more sophisticated invariants can be useful. As we are in a context of deductive verification, our analysis is intraprocedural (as it only uses what is contained in the called functions contracts). On the contrary, traditional static analysis tools have to feature an interprocedural analysis as there is no additional information about the functions called. Thus, our analysis typically targets a smaller amount of code and can afford to be more costly.

Interaction between abstract interpretation and deductive verification was investigated by Yannick Moy [18] to infer loop invariants and preconditions. However, his work was mainly targeted at the C language and the analysis he described aimed to combine both deductive verification and standard abstract interpretation of unannotated programs.

Support for uninterpreted functions in abstract interpretation has already been experimented. For instance, Gange et al. used it to infer properties on non-linear arithmetic [12]. Chang and Leino [2] proposed a domain very similar to ours, but mainly targeted properties on the heap. They did not, to the best of our knowledge, experiment other uses.

Gulwani et al. proposed a way to deal with quantifiers in an arbitrary abstract domain [13]. However, they decided to introduce an under-approximated join operator, while we stick to more classical requirements for the base abstract domain. The formula that could be expressed in their domain also had a specific form, specially tailored for array invariants. Moreover, as it was not in a deductive verification context, they did not have logic formula with quantifiers and had to decide on various heuristics to come up with quantifiers in the abstract domain in the first place.

Other methods to infer loop invariants that are not based on abstract interpretation exist. Furia et al. [11] proposed to use the postconditions of the currently analyzed function to guess loop invariants. While this is an interesting approach as it uses information about the goal of the analysis to orient it, it is less generic as it relies on patterns to propose candidate invariants that have to be evaluated.

6 Conclusion

This article explains how abstract interpretation can be used in a context of deductive verification where the source code is annotated with logic formulas. The interaction with Why3 makes it useful even if it is not very precise and is not sufficient to prove the whole program. We introduced two functors to deal with uninterpreted functions and quantifiers. Those are very generic and can be used to infer properties about arrays as well as algebraic data types. We demonstrated on several examples that it allows the user to omit cumbersome but necessary loop invariants. We have a modular implementation for Why3 built on APRON and ELINA domains and the Fixpoint library.

Our quantifier implementation is very lightweight and benefits from all the work done in numerical domains for domain operations. However, it could be

greatly improved with heuristics to perform instantiations, which would make the domain more precise.

Many examples uses the disjunctive completion of a numerical domain. It is a long known solution to increase the precision of the analysis, and, despite its cost, it proved to be useful in practice. Surprisingly, there is no off-the-shelf implementation that we are aware of.

We did not try to support local, potentially recursive functions, and global recursive functions. It would be interesting to generate preconditions for the first ones, as their entire calling context is known. Automatically generating postconditions for all these functions would surely be an interesting feature. It could be done by translating these functions in the control flow graph.

Finally, an aspect of our work that one may find interesting is the ability for the user to interact with the abstract interpretation analysis: choosing the abstract domain settings and inspecting every invariant generated in the Why3 IDE. This is an important part for a tool such as Why3 that mixes automatic and interactive deductive proofs of the user source code.

Acknowledgments. I am very grateful to Jean-Christophe Filliâtre for his feedback throughout the writing of this paper. I would also like to thank François Bobot, David Bühler, Antoine Miné, and Boris Yakobowski for the interesting discussions we had.

References

1. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
2. Bor-Yuh Evan Chang and K Rustan M Leino. Abstract interpretation with alien expressions and heap structures. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 147–163. Springer, 2005.
3. Martin Clochard, Léon Gondelman, and Mário Pereira. The Matrix Reproved (Verification Pearl). In *VSTTE 2016*, Toronto, Canada, July 2016.
4. P. Cousot and R. Cousot. 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*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.
7. Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

8. Bruno Dutertre. Solving exists/forall problems with yices. In *13th International Workshop on Satisfiability Modulo Theories (SMT 2015)*, July 2015.
9. Jean-Christophe Filliâtre. One logic to use them all. In *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, pages 1–20, Berlin, Heidelberg, 2013. Springer-Verlag.
10. Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Technical report, Université Paris Sud, 2016.
11. Carlo Alberto Furia and Bertrand Meyer. *Inferring Loop Invariants Using Post-conditions*, pages 277–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
12. Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. An abstract domain of uninterpreted functions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 85–103. Springer, 2016.
13. Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *ACM SIGPLAN Notices*, volume 43, pages 235–246. ACM, 2008.
14. John Hatcliff, Gary T Leavens, K Rustan M Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys (CSUR)*, 44(3):16, 2012.
15. Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
16. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*, pages 661–667. Springer, 2009.
17. Laurent Mauborgne. Binary decision graphs. In *International Static Analysis Symposium*, pages 101–116. Springer, 1999.
18. Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.
19. Yannick Moy and Claude Marché. *The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual*. INRIA & LRI, 2011. http://krakatoa.lri.fr/.
20. Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of inferring inductive invariants. In *ACM SIGPLAN Notices*, volume 51, pages 217–231. ACM, 2016.
21. Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 46–59, New York, NY, USA, 2017. ACM.
22. Norihisa Suzuki and David Jefferson. Verification decidability of presburger array programs. *J. ACM*, 27(1):191–205, January 1980.
23. Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

A Correctness of the Uninterpreted Functions Functor

There are two requirements for an abstract domain to be valid. Its operators must satisfy the properties of Fig. 1 and the concretization and abstraction functions must be monotonic and satisfy:

$$\forall d \in \mathbb{D}^\sharp. \forall a \in \mathbb{D}. a \wedge \gamma(d) \subseteq \gamma(\alpha(a, d)) \tag{1}$$

$$\forall b, d \in \mathbb{D}^\sharp. \alpha(\gamma(b), d) \sqsubseteq b \text{ and } \alpha(\gamma(b), d) \sqsubseteq d \tag{2}$$

We only present the proof of Property 1. The other ones are either analogous or trivially proved using definitions of the underlying domain $\mathbb{D}^\sharp$ and the union-find structures $\mathcal{U}$.

Proof. Let $d^\sharp \in \mathbb{D}^\sharp_{UF}$ and $a \in \mathbb{D}$. We pose $(d, u, v) = d^\sharp$. As said above, we only need to prove the property on atoms.

If $a \equiv x = y$ with x, y non-numerical variables, then $\alpha(a, d^\sharp) = (d', \mathsf{union}(x, y, u), v)$ with $d' = \alpha(\phi, d)$ (where ϕ is a formula that expresses a number of equalities that are consequences of $x = y$, discovered with the congruence closure in u). Concretization $\gamma(\alpha(a, d^\sharp))$ can be written $\psi \wedge \psi'$ with $\psi \equiv \gamma(d')$ (after renaming using v) and ψ' contains every equality of $\mathsf{union}(x, yu)$. Then $\psi' = x = y \wedge \phi \wedge \psi''$ where ψ'' contains every equality of u. Then $\gamma(d^\sharp) = \psi'' \wedge \gamma(d)$.

Therefore, we have to prove that $x = y \wedge \psi'' \wedge \gamma(d)$ implies $\gamma(d') \wedge x = y \wedge \phi \wedge \psi''$. Using Property 1 on d', we have that $\phi \wedge \gamma(d)$ implies $\gamma(d')$. Then $x = y \wedge \psi'' \Rightarrow \phi$ is sufficient to prove the property. It is true by construction, as ϕ is obtained with the congruence closure algorithm starting with x and y in u.

If a is another atom, then $\alpha(a, d^\sharp) = (\alpha(a', d), u, v')$ where v' is a new mapping that builds on v with potentially new variable bindings, and a' is a with the uninterpreted terms rewritten. Then properties on $\mathbb{D}^\sharp$ guarantee that Property 1 holds.

B Abstract and Concrete Semantics

$$\begin{aligned}
[\![v]\!](\mathcal{M}) &= \mathcal{M}(v), \mathcal{M} \\
[\![\texttt{let } v \texttt{ = e in e'}]\!](\mathcal{M}) &= [\![\texttt{e'}]\!](\mathcal{M}'(v \mapsto a)) \text{ where } a, \mathcal{M}' = [\![\texttt{e}]\!](\mathcal{M}) \\
[\![\texttt{if } v \texttt{ then e else e'}]\!](\mathcal{M}) &= \text{if } \mathcal{M}(v) \text{ then } [\![\texttt{e}]\!](\mathcal{M}) \text{ else } [\![\texttt{e'}]\!](\mathcal{M}) \\
[\![\texttt{while e do e' done}]\!](\mathcal{M}) &= \text{if a then } [\![\texttt{e'; while e do e' done}]\!](\mathcal{M}') \\
&\quad \text{else } (), \mathcal{M}' \\
&\quad \text{where } a, \mathcal{M}' = [\![e]\!](\mathcal{M}) \\
[\![v' \texttt{ <- } v]\!](\mathcal{M}) &= (), \mathcal{M}[v' \mapsto \mathcal{M}(v)] \\
[\![\text{c}]\!](\mathcal{M}) &= c, \mathcal{M}
\end{aligned}$$

Note that `a; b` is syntactic sugar for `let _ = a in b`.

Fig. 8. Big-Step Concrete Semantics.

Lemma 1. *Let e be a WhyML expression. Then $[\![e]\!]^\sharp$ is a monotonic function.*

Proof. This is easily shown by induction, as the semantics only comprises composition of monotonic functions.

A concrete big-step semantics is described in Fig. 8. A memory state $\mathcal{M}$ is a store, where variables are associated to values. Operations are *forget* (remove a variable from the store), $\mathcal{M}[v \mapsto a]$ (add or replace a binding from variable v to value a) and $\mathcal{M}(v)$ (value of the variable v in $\mathcal{M}$).

We assume that two variables *let*-bound at different places of an expression are different. Thus, a variable is always defined in a memory state $\mathcal{M}$ when it is expected to.

If ϕ is a first-order logical formula, then $\mathcal{M} \vDash \phi$ means that $\mathcal{M}$ (seen as a model) satisfies ϕ. Among other things, it implies that there is a formula ψ equivalent to ϕ such that every variable of ψ is defined in $\mathcal{M}$. Therefore, when we write $\mathcal{M} \vDash \phi$, it is assumed that every variable that appears in ϕ is defined in $\mathcal{M}$. We extend this with the **result** variable: let v be a value, then $(v, \mathcal{M}) \vDash \phi$ is defined as $\mathcal{M}[\mathbf{result} \mapsto v] \vDash \phi$ (**result** is not a variable of the program, therefore $\mathcal{M}$ can not contain it). Thus, $\mathcal{M} \vDash \phi$ implies $(v, \mathcal{M}) \vDash \mathbf{result} = v \wedge \phi$.

We now present a proof of Theorem 1, which ensures that our analysis is sound. We use $\mathbf{r}$ to designate **result** to facilitate readability.

Proof. Let $d \in \mathbb{D}^\sharp$, $\mathtt{e}$ an expression, and $\mathcal{M}$ be a concrete memory state, such that $\mathcal{M} \vDash \gamma(d)$.

We proceed by induction on the big-step semantics definition:

- If $\mathtt{e}$ is a constant: $\mathtt{e} = c$. Property 1 implies $\mathbf{r} = c \wedge \gamma(d) \subseteq \gamma(\alpha(\mathbf{r} = c, d))$, *i.e.* $\mathbf{r} = c \wedge \gamma(d) \Rightarrow \gamma(\alpha(\mathbf{r} = c, d))$ Then it is sufficient to prove that $(c, \mathcal{M})$ satisfies both members of the conjunction. By definition, $(c, \mathcal{M}) \vDash \mathbf{r} = c$.
  Variable $\mathbf{r}$ does not appear in $\gamma(d)$ as it is not a variable of $\mathcal{M}$. Then $(c, \mathcal{M}) \vDash \gamma(d)$.
  So $(c, \mathcal{M}) \vDash \gamma(\alpha(\mathbf{r} = c, d))$, *i.e.* $[\![c]\!](\mathcal{M}) \vDash [\![c]\!]^\sharp(d)$.
- If $\mathtt{e}$ is a loop: $\mathtt{while}\, e' \,\mathtt{do}\, e'' \,\mathtt{done}$.
  Let $(a, \mathcal{M}') = [\![e']\!](\mathcal{M})$.
  We pose $d' = [\![e']\!]^\sharp(d)$.
  By induction, $(a, \mathcal{M}') \vDash \gamma(d')$. So $(a, \mathcal{M}') \vDash \mathbf{r} = a \wedge \gamma(d')$.
  Using Property 1, $(a, \mathcal{M}') \vDash \gamma(\alpha(\mathbf{r} = a, d'))$.
  With the same notations as in the abstract semantics, $d \sqsubseteq d_{loop}$. So if a is false, then $(a, \mathcal{M}') \vDash \gamma(\alpha(\mathbf{r} = \mathit{false}, d'))$.
  If a is true, we pose $(\cdot, \mathcal{M}'') = [\![e'']\!](\mathcal{M}')$.
  Then $[\![\mathtt{e}]\!](\mathcal{M}) = [\![\mathtt{while}\, e' \,\mathtt{do}\, e'' \,\mathtt{done}]\!](\mathcal{M}'')$.
  Let us note $g(a) = [\![e'']\!]^\sharp(\mathit{forget}(\mathbf{r}, \alpha(\mathbf{r} = \mathit{true}, [\![e]\!]^\sharp(a))))$, and $f(a) = a \sqcup g(a)$ so as $d_{loop} = \mathit{fp}_d(f)$.
  By induction, $\mathcal{M}'' \vDash \gamma(g(d))$, then $[\![e]\!](M) \vDash \gamma(\mathit{forget}(\mathbf{r}, \alpha(\mathbf{r} = \mathit{false}, d'_{loop})))$, where $d'_{loop} = \mathit{fp}_{g(d)}(f)$.
  As α and *forget* are monotonic, it is sufficient to prove $d'_{loop} \sqsubseteq d_{loop}$.
  We have $g(d) \sqsubseteq f(d)$, then $g(d) \subseteq d_{loop}$ and $\mathit{fp}_{g(d)}(f) \sqsubseteq \mathit{fp}_{d_{loop}}(f) = d_{loop}$.
- Other cases are proved in a similar fashion.