Abstract interpretation as anti-refinement

Arnaud Spiwack

Inria Paris-Rocquencourt
Ens, 45 rue d'Ulm, 75230 Paris Cedex 05, France
arnaud@spiwack.net

Abstract. This article shows a correspondence between abstract interpretation of imperative programs and the refinement calculus: in the refinement calculus, an abstract interpretation of a program is a specification which is a function.
This correspondence can be used to guide the design of mechanically verified static analyses, keeping the correctness proof well separated from the heuristic parts of the algorithms.

1 Introduction

A mathematical way to describe a static analysis is to see it as a program which tries to prove a theorem about programs. It may fail to do so, but if it succeeds, it effectively acts as a proof of the said theorem. The proof, however, is essentially impossible to check by a human.

To increase the level of trust in a static analysis tool, the tool can be mechanically verified, for instance in Coq [1], thus ensuring that the produced proof is always correct. In the design of a static analysis tool, some parts are crucial for correctness, while other are heuristic. For instance, a static analysis can choose to lose precision to gain performance. Hence, from the point of view of he who wants to ensure the correctness, a static analysis can be seen as an interplay between a correctness enforcer and an heuristic-providing oracle. The question addressed in this article is how to formalise this interplay.

To that end, we use the refinement calculus [2,3]. The refinement calculus is a well-established method for proving program properties. It comes with a natural notion of interaction, generally used to model the interaction between the implementer of a unit of code and its user. In the context of this article, the correctness enforcer plays the role of the implementer while the oracle is the user.

Specifically, this article shows the connection between static analysis by abstract interpretation [4] and the refinement calculus. Namely, it shows that an abstract domain constructs a *specification* of the analysed program, which happens to be given by a function. This correspondence is instrumental in the design of Cosa [5], a Coq formalisation of a shape analysis.

The two subjects have some notation overlap, hence some unconventional notations will be used. The author apologises, but hopes that practitioners of both subjects will not find the notations too surprising or confusing.

2 Predicate transformers

Edsger Dijkstra introduced the idea of using predicate transformers as semantics of imperative programs [6]. The idea is to associate to each program p a function $\mathsf{wlp}(p)$, its *weakest liberal precondition* operator, such that for a property P of program states, $\mathsf{wlp}(p)(P)$ is the weakest condition on the initial state, such that after running p, if p terminates, then P holds.

Weakest liberal precondition accounts for partial correctness. Alternatively, one could use the weakest precondition operator (which additionally imposes that p terminates) to account for total correctness. Termination is not our purpose here, and we will identify programs with their weakest liberal precondition operator.

Predicate transformer semantics is the starting point of refinement calculus [2], and is also commonly used in abstract interpretation – see [7] for a discussion of weakest liberal precondition in relation to abstract interpretation.

2.1 Basic definitions

We will call *predicate transformers* monotone functions in $\mathcal{P}(A) \to \mathcal{P}(B)$ for some sets A and B, and write $\mathcal{P}(A) \to^{+} \mathcal{P}(B)$ for the set of predicate transformers. The set $\mathcal{P}(A) \to \mathcal{P}(B)$ inherits the complete lattice structure of $\mathcal{P}(B)$ and $\mathcal{P}(A) \to^{+} \mathcal{P}(B)$, equipped with the lattice operations of $\mathcal{P}(A) \to \mathcal{P}(B)$, is also a complete lattice. We write $a \sqsubseteq b \iff \forall X^{\in \mathcal{P}(A)}.\ a(X) \subseteq b(X)$ for the inherited order.

We shall call the following operations of predicate transformers *regular operations*. They have a direct interpretation as program constructs. Programs will be interpreted as homogeneous predicate transformers $\mathcal{P}(A) \to^{+} \mathcal{P}(A)$, however the regular operations also work with general predicate transformers $\mathcal{P}(A) \to^{+} \mathcal{P}(B)$.

Sequence $(a; b)\,(X) = a(b(X))$
Reads as "do a then do b". The definition of sequence emphasises the fact that the weakest liberal precondition semantics is a *backward* semantics. Sequence is associative, and monotone:
- $(a; b)\,; c = a; (b; c)$
- $a \sqsubseteq a' \wedge b \sqsubseteq b' \implies a; b \sqsubseteq a'; b'$

Skip $1(X) = X$
Does not do anything. Skip is neutral for sequence:
- $1; a = a = a; 1$

Choice $(a + b)\,(X) = a(X) \cap b(X)$
Non-deterministic choice. Choice is associative, commutative and monotone. Moreover sequence distributes on the right over choice:
- $(a + b) + c = a + (b + c)$
- $a + b = b + a$
- $(a + b)\,; c = a; c + b; c$
- $p \sqsubseteq (a + b)\,; q \iff p \sqsubseteq a; q \wedge p \sqsubseteq b; q$

Hang $0(X) = \top$
Hang loops indefinitely. It is neutral for choice, sequence distributes on the right over it, and it is the largest predicate transformer:

- $0 + a = a = a + 0$
- $0; a = 0$
- $a \sqsubseteq 0$

Iteration a^*, for $a \in \mathcal{P}(A) \to^+ \mathcal{P}(A)$, is the largest fixed point of the (monotone) function which maps p to $1 + a; p$. It runs a in sequence a non-deterministic number of times (including none, and infinitely many). It has the following properties [2, Chapter 21]:

- $a^*; q = q + a; a^*; q$
- $p \sqsubseteq q + a; p \implies p \sqsubseteq a^*; q$

It should be noted that despite the name "regular operations", predicate transformers do not form a Kleene algebra under these operations. Indeed the left distributivity laws are missing: $a; (b + c) = a; b + a; c$ and $a; 0 = 0$ do not hold in general.

2.2 Programs

In this setting, a programming language consists in a set $\mathcal{S}$ of states together with a set $\mathcal{I} \subseteq \mathcal{P}(\mathcal{S}) \to^+ \mathcal{P}(\mathcal{S})$ of *basic instructions*. A program in the language $(\mathcal{S}, \mathcal{I})$ is an element of the subset of $\mathcal{P}(\mathcal{S}) \to^+ \mathcal{P}(\mathcal{S})$ generated by $\mathcal{I}$ and the regular operations.

The use of non-deterministic choice and iterations make the programs non-deterministic. This is a natural setting for both program refinement and abstract interpretation. However, a typical programming language will feature a set of tests $\mathcal{B}$ such that for all $b \in B$, there is $[\![b]\!] \in \mathcal{P}(\mathcal{S})$, and $\mathsf{guard}(b)$ is an instruction, such that $s \in \mathsf{guard}(b)(X) \iff s \in [\![b]\!] \implies s \in X$.

With this assumption, the usual deterministic programming constructs can be recovered: $\mathsf{if}\ b\ \mathsf{then}\ u\ \mathsf{else}\ v = (\mathsf{guard}(b); u) + (\mathsf{guard}(\neg b); v)$, and $\mathsf{while}\ b\ \mathsf{do}\ u = (\mathsf{guard}(b); u)^*; \mathsf{guard}(\neg b)$.

Example 1. As an example, let us consider a language with a single memory cell containing an integer. In other words, $\mathcal{S} = \mathbb{Z}$. It has two tests, pos and npos, whose semantics are given by:

- $[\![\mathsf{pos}]\!] \iff \{n \in \mathbb{Z} \mid n > 0\}$
- $[\![\mathsf{npos}]\!] \iff \{n \in \mathbb{Z} \mid n \leqslant 0\}$

and a operation dec, which decrements the integer held in the state. Its semantics is given by:

- $\mathsf{dec}(X) = \{n \in \mathbb{Z} \mid n - 1 \in X\}$

This language expresses, for example, the simple program whose effect is to decrease the integer held in the state until it is non-positive. We shall call this program d:

- $d = \mathsf{while\ pos\ do\ dec} = (\mathsf{guard}(\mathsf{pos}); \mathsf{dec})^*; \mathsf{guard}(\mathsf{npos})$

2.3 Relations

A relation is usually seen as a subset of $A \times B$, however, it will be more convenient to see them, equivalently, as functions of $A \to \mathcal{P}(B)$.

Given a relation $r \in A \to \mathcal{P}(B)$, we can extend it to a predicate transformer in two ways:

- $\langle r \rangle \in \mathcal{P}(A) \to^{+} \mathcal{P}(B)$ defined by $\langle r \rangle (X) = \bigcup_{x \in X} r(x)$
- $[r] \in \mathcal{P}(B) \to^{+} \mathcal{P}(A)$ defined by $[r] (Y) = \{x \in A \mid r(x) \subseteq Y\}$

The predicate transformers $\langle r \rangle$ and $[r]$ form a Galois connection *i.e.*:

- $\forall X^{\in \mathcal{P}(A)}, Y^{\in \mathcal{P}(B)}.\ \langle r \rangle (X) \subseteq Y \iff X \subseteq [r] (Y)$

or equivalently:

- $\forall X^{\in \mathcal{P}(A)}.\ X \subseteq [r] (\langle r \rangle (X))$
- $\forall Y^{\in \mathcal{P}(B)}.\ \langle r \rangle ([r] (Y)) \subseteq Y$

In fact, every Galois connection between powersets is of that form. This is due to the general fact about complete lattices that a left adjoint – like $\langle r \rangle$ – is the same thing as a function which preserves joins. In the case of powersets, a function which preserves joins is characterised by its action on singletons, hence is of the form $\langle r \rangle$.

Identifying a function f to its graph, we hence have a Galois connection between $\langle f \rangle$ and $[f]$. These are better known as the direct image and the inverse image of f, which we will write f_* and f^{-1} respectively. We shall make use of the following consequence of their being a Galois connection:

- $x \in f^{-1}(X) \iff f(x) \in X$

The properties of Galois connections can also be read directly in terms of the predicate transformer lattice:

- $\langle r \rangle ; p \sqsubseteq q \iff p \sqsubseteq [r] ; q$
- $p ; [r] \sqsubseteq q \iff p \sqsubseteq q ; \langle r \rangle$
- $f_* ; p \sqsubseteq q \iff p \sqsubseteq f^{-1} ; q$
- $p ; f^{-1} \sqsubseteq q \iff p \sqsubseteq q ; f_*$

3 Abstract interpretation

Abstract interpretation [4] is a framework for static analysis in which the objects of study are called *domains*. As general as the definitions in this section are, they fail to capture the full generality of abstract interpretation. However, they are sufficient for most purposes – at least for imperative languages.

Fixing a programming language $(\mathcal{S}, \mathcal{I})$, the powerset $\mathcal{P}(\mathcal{S})$ is called the concrete domain and the interpretation of a program as a predicate transformer $\mathcal{P}(A) \to^{+} \mathcal{P}(A)$ is called the concrete semantics.

A departure from common practice is that the concrete semantics, the weakest liberal precondition, is backward – *i.e.* a function from a set of final states to corresponding initial states – whereas often the concrete semantics is chosen to be forward. This choice has been made to stay closer to the practice in refinement calculus. Having a backward concrete semantics does not, however, constrain the analysis to be backward too. In the rest of the paper we will mainly consider forward analysis. Moreover, forward semantics are usually constructed from a relational semantics, *i.e.* they are of the form $\langle r \rangle$, in which case $[r]$ will be our backward semantics.

An abstract domain is a set $\mathcal{S}^\sharp$ together with a concretisation function $\gamma : \mathcal{S}^\sharp \to \mathcal{P}(\mathcal{S})$ and extra material to construct an *abstract semantics* to each program. The abstract semantics of a program is a forward function $p^\sharp : \mathcal{S}^\sharp \to \mathcal{S}^\sharp$ which has the following correctness property:

- $\forall s^{\sharp \in \mathcal{S}^\sharp}.\ \forall S^{\in \mathcal{P}(\mathcal{S})}.\ S \subseteq \gamma\big(s^\sharp\big) \implies S \subseteq p\big(\gamma\big(p^\sharp\big(s^\sharp\big)\big)\big)$

Which can, equivalently be stated as:

- $\forall s^{\sharp \in \mathcal{S}^\sharp}.\ \gamma\big(s^\sharp\big) \subseteq p\big(\gamma\big(p^\sharp\big(s^\sharp\big)\big)\big)$

This phrasing of the correctness property may look a bit contorted to the practitioner of abstract interpretation. It is the consequence of having a backward concrete semantics and a forward abstract semantics. When the concrete semantics is of the form $p = [p_0]$, then this correctness property coincides with the more familiar one:

- $\forall s^{\sharp \in \mathcal{S}^\sharp}.\ \langle p_0 \rangle \left(\gamma\big(s^\sharp\big)\right) \subseteq \gamma\big(p^\sharp\big(s^\sharp\big)\big)$

Abstract domains are meant to be composed. For that reason, the abstract semantics $p^\sharp$ is computed out of more atomic functions, which are, in particular, stable by Cartesian product. Writing $s \leqslant s' \iff \gamma(s) \subseteq \gamma(s')$ for the order induced on $\mathcal{S}^\sharp$ by the concretisation function, the abstract domain comes equipped with the following:

Join An operator $\sqcup \in \mathcal{S}^\sharp \times \mathcal{S}^\sharp \to \mathcal{S}^\sharp$ such that:
- $s^\sharp \leqslant s^\sharp \sqcup t^\sharp$
- $t^\sharp \leqslant s^\sharp \sqcup t^\sharp$

Post-fixed point An operator $\mathsf{pfp} \in \big(\mathcal{S}^\sharp \to \mathcal{S}^\sharp\big) \to \big(\mathcal{S}^\sharp \to \mathcal{S}^\sharp\big)$ such that:
- $\forall f^{\in \mathcal{S}^\sharp \to \mathcal{S}^\sharp}, s^{\sharp \in \mathcal{S}^\sharp}.\ s^\sharp \leqslant \mathsf{pfp}(f)\big(s^\sharp\big)$
- $\forall f^{\in \mathcal{S}^\sharp \to \mathcal{S}^\sharp}, s^{\sharp \in \mathcal{S}^\sharp}.\ f\big(\mathsf{pfp}(f)\big(s^\sharp\big)\big) \leqslant \mathsf{pfp}(f)\big(s^\sharp\big)$

Typically, the post-fixed point operator is derived from a widening operator $\nabla \in \mathcal{S}^\sharp \times \mathcal{S}^\sharp \to \mathcal{S}^\sharp$, which has the following properties:
- $s^\sharp \leqslant s^\sharp \nabla t^\sharp$
- $t^\sharp \leqslant s^\sharp \nabla t^\sharp$
- For every increasing sequence $(x_n)_{n \in \mathbb{N}}$, the sequence $(y_n)_{n \in \mathbb{N}}$ defined by $y_0 = x_0$ and $y_{n+1} = y_n \nabla x_{n+1}$ verifies $\exists n^{\in \mathbb{N}}.\ y_{n+1} \leqslant y_n$.

Then, taking, mutually recursively, $x_0 = s^\sharp$, $x_{n+1} = f(y_n)$, and y_n such as above, we can then define $\mathsf{pfp}(f)\big(s^\sharp\big)$ as any y_n such that $y_{n+1} \leqslant y_n$.

Transfer functions An abstract semantics $i^\sharp$ of the instruction $i \in \mathcal{I}$

The abstract semantics $p^\sharp$ of the program p is defined by induction on p where the base case is given by the transfer functions. The correction of $p^\sharp$ follows from the properties stated above.

- $(a;b)^\sharp\big(s^\sharp\big) = b^\sharp\big(a^\sharp\big(s^\sharp\big)\big)$
- $(a+b)^\sharp\big(s^\sharp\big) = \big(a^\sharp\big(s^\sharp\big)\big) \sqcup \big(b^\sharp\big(s^\sharp\big)\big)$
- $1^\sharp\big(s^\sharp\big) = s^\sharp$
- $0^\sharp\big(s^\sharp\big)$ can be chosen arbitrarily
- $(a^*)^\sharp\big(s^\sharp\big) = \mathsf{pfp}\big(a^\sharp\big)\big(s^\sharp\big)$

Example 2. Let us define an abstract domain for the example language of Section 2.2: we shall abstract the state – a single integer – by the signs it may take. More precisely, we take for $S^\sharp$ the non-empty sets in $\mathcal{P}(\{-,0,+\})$ and the concretisation is defined as:

- $\gamma\big(s^\sharp\big) = \big\{n \in \mathbb{Z} \mid \mathsf{sign}(n) \in s^\sharp\big\}$

The abstract transfer function for guard instructions constrain the abstract state to the relevant signs.

- $\mathsf{guard}^\sharp(\mathsf{pos})\big(s^\sharp\big) = s^\sharp \cap \{+\}$
- $\mathsf{guard}^\sharp(\mathsf{npos})\big(s^\sharp\big) = s^\sharp \cap \{-,0\}$

The abstract transfer function for the decrementing command maps positive to non-negative and non-positive to negative:

- $\mathsf{dec}_0(+) = \{0,+\}$
- $\mathsf{dec}_0(0) = \{-\}$
- $\mathsf{dec}_0(-) = \{-\}$
- $\mathsf{dec}^\sharp\big(s^\sharp\big) = \bigcup_{x \in s^\sharp} \mathsf{dec}_0(x)$

Since the abstract state space is a powerset, we can use union as the abstract join, and since it is finite, union is also a widening:

- $s^\sharp \nabla t^\sharp = s^\sharp \sqcup t^\sharp = s^\sharp \cup t^\sharp$

Now that the abstract domain is set up, let us run the abstract interpretation on the program d from Section 2.2 with the input state $\{0,+\}$:

1. Entering the loop with initial state $\{0,+\}$
2. Applying $\mathsf{guard}^\sharp(\mathsf{pos})$: state becomes $\{+\}$
3. Applying $\mathsf{dec}^\sharp$: state becomes $\{0,+\}$
4. Invariant found after one iteration: $\{0,+\} \cup \{0,+\} = \{0,+\}$
5. Applying $\mathsf{guard}^\sharp(\mathsf{npos})$: final state is $\{0\}$

4 Data refinement

Refinement calculus [2,3] is a discipline to prove the correctness of imperative programs, in a spirit close to Hoare logic. It arises from the remark that, if most predicate transformers do not represent programs, they still represent program *specifications*. Specifications are then *refined* into more precise specifications, and eventually into programs.

A key point of the refinement calculus is that the refined specification need not act on the same state as the abstract one. It is typical to use ideal objects – like multisets – on the abstract side, and more concrete datatypes – like linked lists – on the refined side.

We say [3] that $a \in \mathcal{P}(A) \to^+ \mathcal{P}(A)$ is refined by $b \in \mathcal{P}(B) \to^+ \mathcal{P}(B)$ through the *coupling invariant* $\iota \in \mathcal{P}(A) \to^+ \mathcal{P}(B)$, written $a \sqsubseteq_\iota b$, when $\iota; a \sqsubseteq b; \iota$. Intuitively ι is an action which transforms concrete states into abstract states, so $\iota; a \sqsubseteq b; \iota$ reads "doing b then abstracting the state is more precise than abstracting the state then doing a". To emphasise that the type of the state has changed, this relation is often called a *data refinement*.

Example 3. Specifications of imperative programs are typically given as pairs of a precondition and a postcondition. For instance: under the precondition that the initial state is a non-positive integer, the postcondition that the state is 0 holds after the program has been run. Both preconditions and postconditions can be expressed systematically as (backward) predicate transformers; they can be paired up into a full specification using sequence:

- $F_{\mathsf{post}}(X) = \{p \in \mathbb{Z} \mid 0 \in X\}$
- $F_{\mathsf{pre}}(X) = \{n \in \mathbb{Z} \mid n \leqslant 0 \wedge n \in X\}$
- $F = F_{\mathsf{pre}}; F_{\mathsf{post}}$

So that $F_{\mathsf{post}}(X)$ is either all of $\mathbb{Z}$ if $0 \in X$ or the empty set otherwise, and $F_{\mathsf{pre}}(X)$ simply ignores the states in X which do not verify the precondition.

The program d from Section 2.2 meets the specification F, however, the state is represented as the *opposite* integer. Hence we have an ι which reflects this representation:

- $\iota_0(n) = -n$
- $\iota = \iota_{0*}$

As per the definition of refinement, the statement that the program d implements the specification reads

- $F \sqsubseteq_\iota \mathsf{while\ pos\ do\ decr}$

It is equivalent to the statement that the precondition entails the weakest liberal precondition of $d = \mathsf{while\ pos\ do\ decr}$:

- $\forall n^{\in\mathbb{Z}}.\ n \geqslant 0 \implies n \in \mathsf{wlp}(d)(\{0\})$

which is the typical proof obligation in a Hoare logic setting.

The take away from data refinement is that it does not matter what coupling invariant is used, as long as *all the function use the same coupling invariant*. Or, more realistically, under some separation property, if all the function *which have access to some part A of the state* all have coupling invariants which agree on A.

In practice there are two reasons to refine the type of (a part of) the state: it may be that it is an ideal type, say finite sets of integer, which may be refined into an actual concrete data type, for instance list of integers. Or it may be that the proposed data type is not efficient, and will be refined into a more efficient representation – list of integers could be refined into binary trees.

5 Abstract interpretation in refinement calculus

The main result of this article is that abstract interpretation can be characterised in the language of the refinement calculus: an abstract interpretation of a program p is a *specification* verified by p which is also a function.

Theorem 1. *The soundness condition of abstract interpretation is a refinement condition:* $p^{\sharp^{-1}} \sqsubseteq_{\langle\gamma\rangle} p \iff \forall s^\sharp{}^{\in\mathcal{S}^\sharp}.\ \gamma\big(s^\sharp\big) \subseteq p\big(\gamma\big(p^\sharp\big(s^\sharp\big)\big)\big)$

Proof. We have the following equivalent characterisation, thanks to the Galois connection properties:

- $p^{\sharp^{-1}} \sqsubseteq_{\langle\gamma\rangle} p \iff p^{\sharp^{-1}}; [\gamma] \sqsubseteq [\gamma]\,; p$

From which it follows that:

$p^{\sharp^{-1}} \sqsubseteq_{\langle\gamma\rangle} p$
$\iff$ (Definition of sequence)
$\forall Y^{\in\mathcal{P}(\mathcal{S})}.\ p^{\sharp^{-1}}([\gamma]\,(Y)) \subseteq [\gamma]\,(p(Y))$
$\iff$ (Definition of inclusion)
$\forall Y^{\in\mathcal{P}(\mathcal{S})}, s^\sharp{}^{\in\mathcal{S}^\sharp}.\ s^\sharp \in p^{\sharp^{-1}}([\gamma]\,(Y)) \implies s^\sharp \in [\gamma]\,(p(Y))$
$\iff$ (Definition of $[\gamma]$ and basic property of $p^{\sharp^{-1}}$)
$\forall Y^{\in\mathcal{P}(\mathcal{S})}, s^\sharp{}^{\in\mathcal{S}^\sharp}.\ p^\sharp\big(s^\sharp\big) \in [\gamma]\,(Y) \implies \gamma\big(s^\sharp\big) \subseteq p(Y)$
$\iff$ (Definition of $[\gamma]$)
$\forall Y^{\in\mathcal{P}(\mathcal{S})}, s^\sharp{}^{\in\mathcal{S}^\sharp}.\ \gamma\big(p^\sharp\big(s^\sharp\big)\big) \subseteq Y \implies \gamma\big(s^\sharp\big) \subseteq p(Y)$
$\iff$ ($\Rightarrow$ by $Y = \gamma\big(p^\sharp\big(s^\sharp\big)\big)$ and $\Leftarrow$ by monotonicity of p)
$\forall s^\sharp{}^{\in\mathcal{S}^\sharp}.\ \gamma\big(s^\sharp\big) \subseteq p\big(\gamma\big(p^\sharp\big(s^\sharp\big)\big)\big)$

In [8], Cousot & Cousot describe abstract interpretation of inference rule systems. Their approach to defining abstract interpretation resembles refinement calculus, they use, in particular, the remark that inference rule systems can be represented as predicate transformers. Theorem 1 further illuminates the connection.

Although so far we have mostly considered forward analyses, a similar characterisation to Theorem 1 holds for backward analysis:

Theorem 2. $p^\sharp_* \sqsubseteq_{\langle\gamma\rangle} p \iff \forall s^\sharp \in \mathcal{S}^\sharp.\ \gamma\big(p^\sharp\big(s^\sharp\big)\big) \subseteq p\big(\gamma\big(s^\sharp\big)\big)$

In traditional refinement calculus, the process consists in starting with an abstract definition, and refine it towards a more concrete definition, weakening the preconditions, strengthening the postconditions while making the state more suitable for execution. In static analysis, refinement calculus is used somewhat backwards: starting from a concrete implementation, it is refined into a more abstract definition, in effect strengthening the precondition and weakening the postconditions, while still making the state more suitable for execution.

6 Conclusion

A previous work by Sylvain Boulmé and Michaël Périn [9] uses refinement calculus as a mean to check, in Coq, the correctness of a certificate validation procedure for certificate meant to be output by an abstract interpreter. Although this work is at the intersection of abstract interpretation and refinement calculus, it does not try to establish a connection between refinement calculus and the correctness condition of the abstract interpretation procedure.

The present article shows that the language of abstract interpretation can be recast in terms of the refinement calculus. This has been used in the formalisation of Cosa [5], a Coq verified implementation of an abstract domain for shape analysis. Cosa targets Compcert C [10], and uses numerical domains by David Pichardie & *al* [11].

Cosa relies on a variant of the refinement calculus introduced by Peter Hancock based not on predicate transformers but on so-called *interaction structures* [12]. Compared to predicate transformers, interaction structures carry more information: the set of predicate transformers can be seen as a quotient of the set of interaction structures. The additional information contained in interaction structures can be used to derive a datatype of *strategies* which the oracle is charged with providing, hence formalising the separation between the oracle, which has no bearing on the correctness and does not need to be mechanically verified, and the rules constituting the domain which ensure correctness.

Interaction structures were initially developed as a variant of refinement calculus suitable for type theory. Thanks to the results of this article, interaction structures can be also leveraged for abstract interpretation.

References

1. The Coq development team: The Coq Proof Assistant
2. Back, R.J., von Wright, J.: Refinement calculus: a systematic introduction. (1998)
3. von Wright, J.: The lattice of data refinement. Acta Informatica **135** (1994) 105–135
4. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of logic and computation **2**(4) (1992) 511–547
5. Spiwack, A.: Cosa (2013)
6. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (August 1975) 453–457

7. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation (Extended Abstract). Electronic Notes in Theoretical Computer Science **6** (January 1997) 77–102
8. Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretations. Proceedings of the 19th ACM SIGPLAN-SIGACT ... (1992)
9. Boulmé, S., Périn, M.: Refinement calculus for a simple certification of static polyhedral analysis with code transformations. Technical report, Verimag (2013)
10. Leroy, X., Blazy, S., Dargaye, Z., Tristan, J.B.: CompCert
11. Blazy, S., Laporte, V., Maroneze, A., Pichardie, D.: Formal verification of a C value analysis based on abstract interpretation. Static Analysis (2013)
12. Hancock, P., Hyvernat, P.: Programming interfaces and basic topology. Annals of Pure and Applied Logic **137**(1-3) (May 2009) 1–55