Abstract Interpretation of Annotated Commands

Tobias Nipkow*

Fakultät für Informatik, Technische Universität München

Abstract. This paper formalizes a generic abstract interpreter for a while-language, including widening and narrowing. The collecting semantics and the abstract interpreter operate on annotated commands: the program is represented as a syntax tree with the semantic information directly embedded, without auxiliary labels. The aim of the paper is simplicity of the formalization, not efficiency or precision. This is motivated by the inclusion of the material in a theorem prover based course on semantics.

1 Introduction

The purpose of this work is to formalize the basics of abstract interpretation in a theorem prover in as simple a manner as possible. The background is a course on semantics [10] that is completely based on Isabelle/HOL [11]. The first 4 weeks of the course are dedicated to the theorem prover; the rest of the course focuses on the semantics of a simple while-language and on its applications (e.g. compiler correctness). In particular, the last 4 weeks are dedicated to abstract interpretation. Hence the need to concentrate on the essence and simplify the technicalities. A second desideratum was to stick with the unifying representation of programs as abstract syntax trees employed throughout the course. Finally we wanted to visualize the stepwise computation of the semantics and the abstract interpreter as directly as possible. As a result we chose syntax trees annotated with (concrete or abstract) semantic information and a Jacobi-like iteration strategy. That is, displaying the annotated program after each iteration step animates the stepwise approximation of the result. This paper presents the formalization of a collecting semantics, a derived small-step operational semantics, and a stepwise development of a series of abstract interpreters, up to and including widening and narrowing. Just like previous formalizations, we only consider concretization, not abstraction, and verify only correctness, not optimality of the interpreter. Due to space limitations, this is not a tutorial paper and readers are assumed to be familiar with abstract interpretation [4,5,8].

Abstract interpretation is a vast research area, but only a few formalizations have been published, primarily the impressive work by Pichardie [12,13,3], who employs Coq's expressive type and module system to great effect. The key differences to our approach are that Pichardie labels the nodes of the syntax tree

* Research supported by NICTA and by DFG GK PUMA

whereas we annotate the tree directly with information, his whole approach is denotational (i.e. nested iterations) whereas ours is based on one global iteration, the termination proofs for widening and narrowing are very different, and overall his model is more refined and ours is simpler, which reflects the different aims. Bertot [1] presents an approach that is also based on annotating the program directly but is otherwise very different from ours: Bertot's reference point is a Hoare logic, not a collecting semantics. There have also been a number of specific applications of abstract interpretation, eg [9,2], but without a formalization of the generic theory.

2 Notation

The logic HOL of the Isabelle proof assistant conforms largely to everyday mathematical notation. This section summarizes non-standard notation.

The function space is denoted by $\Rightarrow$. Type variables are denoted by $'a$, $'b$, etc. The notation $t :: \tau$ means that term t has type τ. Type constructors follow postfix syntax, eg $'a\ set$ is the type of sets of elements of type $'a$. Lists over type $'a$, type $'a\ list$, come with the empty list $[]$, the infix constructor $\cdot$, and enumeration syntax $[x_1, \ldots, x_n]$. The **datatype** $'a\ option = None \mid Some\ 'a$ is predefined. The notation $[\![A_1, \ldots, A_n]\!] \Longrightarrow B$ is an implication with the premises A_i and the conclusion B.

3 Annotated Commands

There are arithmetic and boolean expressions, where *vname* = *string*:

datatype *aexp* = *N int* | *V vname* | *Plus aexp aexp*
datatype *bexp* = *Bc bool* | *Not bexp* | *And bexp bexp* | *Less aexp aexp*

Their evaluation is defined as usual: *aval* :: *aexp* ⇒ *state* ⇒ *int* and *bval* :: *bexp* ⇒ *state* ⇒ *bool*, where *state* = *vname* ⇒ *int*. There are commands (type *com*) and *annotated commands*, with the customary concrete syntax; annotations of type $'a$ are enclosed in braces:

datatype $'a\ acom =$
  SKIP { $'a$ }
| *string* ::= *aexp* { $'a$ }
| $'a\ acom$; $'a\ acom$
| *IF bexp THEN* $'a\ acom$ *ELSE* $'a\ acom$ { $'a$ }
| { $'a$ } *WHILE bexp DO* $'a\ acom$ { $'a$ }

Type *com* is not shown as it is identical to *acom*, but without the annotations.

Annotations positioned at the end of a command refer to the very end of that command, not to some subcommand (eg the *ELSE* branch or the *WHILE* body). The annotation in front of *WHILE* is meant to hold the invariant.

There are many alternatives as to the placement and number of annotations. Our choice fits our formalization of semantics and abstract interpretation, but other choices are possible.

There are a number of auxiliary functions: *post* :: $'a$ *acom* $\Rightarrow$ $'a$ extracts the post-annotation of a command (*post* (c_1; c_2) = *post* c_2), *strip* :: $'a$ *acom* $\Rightarrow$ *com* removes all annotations, and *anno* :: $'a \Rightarrow$ *com* $\Rightarrow$ $'a$ *acom* annotates a command with the same annotation everywhere.

We say that c_1 and c_2 are *strip-equal* if *strip* c_1 = *strip* c_2.

4 Collecting Semantics

The purpose of the collecting semantics is to collect the set of all reachable states at some program point as an annotation. Both the collecting semantics and later the abstract interpreter are defined by iterated simultaneous "micro-step" execution of all atomic commands, similar to the Jacobi method for linear equations. This is very different from a denotational approach where whole subcommands are executed in one go. We define a function *step* :: *state set* $\Rightarrow$ *state set acom* $\Rightarrow$ *state set acom* that pushes a set of initial states one step into an annotated command c and propagates the *state set* annotations inside c one step further:

step S (*SKIP* $\{P\}$) = *SKIP* $\{S\}$
step S (x ::= e $\{P\}$) = x ::= e $\{\{s' \mid \exists s{\in}S.\ s' = s(x := aval\ e\ s)\}\}$
step S (c_1; c_2) = *step* S c_1; *step* (*post* c_1) c_2
step S (*IF* b *THEN* c_1 *ELSE* c_2 $\{P\}$) = *IF* b *THEN step* $\{s \in S \mid bval\ b\ s\}$ c_1
ELSE step $\{s \in S \mid \neg\ bval\ b\ s\}$ c_2
$\{post\ c_1 \cup post\ c_2\}$
step S ($\{Inv\}$ *WHILE* b *DO* c $\{P\}$) = $\{S \cup post\ c\}$
WHILE b *DO step* $\{s \in Inv \mid bval\ b\ s\}$ c
$\{\{s \in Inv \mid \neg\ bval\ b\ s\}\}$

Annotations for *IF* and *WHILE* are (in principle) redundant, but the invariant is conceptually important and the post-annotations allow a uniform definition of *post* for arbitrary annotations.

The beauty of annotated commands is the ability to visualize the semantics by evaluating *step*. This is possible thanks to Isabelle's evaluation mechanism, which can handle finite sets. Here is a small (contrived) example, further examples follow. Given the command *cs-ex* =

$''x''$ ::= *Plus* (*V* $''x''$) (*N* 1) $\{\{\lambda x.\ 5, \lambda x.\ 6, \lambda x.\ 7\}\}$;
$''x''$ ::= *Plus* (*V* $''x''$) (*N* 2) $\{\emptyset\}$

evaluation of *show-acom* $[''x'']$ (*step* $\{\lambda x.\ -1, \lambda x.\ 1\}$ *cs-ex*) yields

$''x''$ ::= *Plus* (*V* $''x''$) (*N* 1) $\{\{[(''x'', 0)], [(''x'', 2)]\}\}$;
$''x''$ ::= *Plus* (*V* $''x''$) (*N* 2) $\{\{[(''x'', 7)], [(''x'', 8)], [(''x'', 9)]\}\}$

In the input, states are functions, but in the output, the pretty-printing function *show-acom* converts states into variable-value pairs, for a given list of variables.

In order to find least fixed-points of *step*, we extend orderings $\le$ on type $'a$ to $'a\ acom$:

$$\begin{array}{ll}
SKIP\ \{S\} \le SKIP\ \{S'\} & \longleftrightarrow\ S \le S' \\
x ::= e\ \{S\} \le x' ::= e'\ \{S'\} & \longleftrightarrow\ x = x' \wedge e = e' \wedge S \le S' \\
c_1;\ c_2 \le d_1;\ d_2 & \longleftrightarrow\ c_1 \le d_1 \wedge c_2 \le d_2 \\
\multicolumn{2}{l}{IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{S\} \le IF\ b'\ THEN\ d_1\ ELSE\ d_2\ \{S'\}} \\
\multicolumn{2}{l}{\longleftrightarrow\ b = b' \wedge c_1 \le d_1 \wedge c_2 \le d_2 \wedge S \le S'} \\
\multicolumn{2}{l}{\{I\}\ WHILE\ b\ DO\ c\ \{P\} \le \{I'\}\ WHILE\ b'\ DO\ c'\ \{P'\}} \\
\multicolumn{2}{l}{\longleftrightarrow\ b = b' \wedge c \le c' \wedge I \le I' \wedge P \le P'}
\end{array}$$

In all other cases $c \le c'$ is defined to be *False*. We can now compare commands annotated with state sets. The underlying ordering on the state sets is $\subseteq$. A simple inductive proof shows monotonicity of *step*:

Lemma *If* $c_1 \le c_2$ *and* $S_1 \subseteq S_2$ *then* $step\ S_1\ c_1 \le step\ S_2\ c_2$.

To show that *step* has a least fixed point we turn *acom* into a complete lattice.

4.1 Indexed Complete Lattices

Only subsets of *acom* form a complete lattice, namely $\{c' \mid strip\ c' = c\}$ for any c. Hence we define a little theory of *indexed* complete lattices parameterized by

$$L :: 'i \Rightarrow 'a\ set \quad \text{and} \quad Glb :: 'i \Rightarrow 'a\ set \Rightarrow 'a$$

where $'i$ is the index type and $L\ i$ the carrier set. We assume that *Glb* is the greatest lower bound and that $L\ i$ is closed under *Glb*:

$$\begin{array}{l}
[\![A \subseteq L\ i;\ a \in A]\!] \Longrightarrow Glb\ i\ A \le a \\
[\![b \in L\ i;\ \forall\, a{\in}A.\ b \le a]\!] \Longrightarrow b \le Glb\ i\ A \\
A \subseteq L\ i \Longrightarrow Glb\ i\ A \in L\ i
\end{array}$$

In this context we can prove that $lfp\ f\ i = Glb\ i\ \{a \in L\ i \mid f\ a \le a\}$ is indeed the least fixed and post-fixed point. Note that we define *post-fixed point* to mean $f\ x \le x$, which is customary in the abstract interpretation literature, although usually this is called a pre-fixed point.

4.2 Application to Collecting Semantics

The Glb of a set of annotated commands is taken pointwise, assuming the commands are all *strip*-equal. More generally, any function on annotation sets can be lifted to sets of annotated commands in this pointwise manner (where $f\ `\ M$ is the image of a function over a set):

$$\begin{array}{l}
lift :: ('a\ set \Rightarrow 'a) \Rightarrow com \Rightarrow 'a\ acom\ set \Rightarrow 'a\ acom \\
lift\ F\ SKIP\ M = SKIP\ \{F\ (post\ `\ M)\} \\
lift\ F\ (x ::= a)\ M = x ::= a\ \{F\ (post\ `\ M)\}
\end{array}$$

$$
\begin{array}{l}
\mathit{lift}\ F\ (c_1;\ c_2)\ M = \mathit{lift}\ F\ c_1\ (\mathit{sub}_1\ `\ M);\ \mathit{lift}\ F\ c_2\ (\mathit{sub}_2\ `\ M) \\
\mathit{lift}\ F\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2)\ M = \mathit{IF}\ b\ \mathit{THEN}\ \mathit{lift}\ F\ c_1\ (\mathit{sub}_1\ `\ M) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{ELSE}\ \mathit{lift}\ F\ c_2\ (\mathit{sub}_2\ `\ M) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{F\ (\mathit{post}\ `\ M)\} \\
\mathit{lift}\ F\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ M = \{F\ (\mathit{invar}\ `\ M)\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{WHILE}\ b\ \mathit{DO}\ \mathit{lift}\ F\ c\ (\mathit{sub}_1\ `\ M) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{F\ (\mathit{post}\ `\ M)\}
\end{array}
$$

Subcommands and the invariant are accessed by auxiliary functions:

$$
\begin{array}{l}
\mathit{sub}_1\ (c_1;\ c_2) = c_1 \\
\mathit{sub}_1\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \{S\}) = c_1 \\
\mathit{sub}_1\ (\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\}) = c \\
\mathit{sub}_2\ (c_1;\ c_2) = c_2 \\
\mathit{sub}_2\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \{S\}) = c_2 \\
\mathit{invar}\ (\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\}) = I
\end{array}
$$

Lemma Type $'a\ \mathit{set}\ \mathit{acom}$ is a complete lattice indexed by com where $L\ c = \{c' \mid \mathit{strip}\ c' = c\}$ and $\mathit{Glb} = \mathit{lift}\ \bigcap$.

Of course this works for any complete lattice of annotations, but we only need it for sets. We can now define the collecting semantics as a least fixed-point:

$$
\begin{array}{l}
\mathit{CS} :: \mathit{com} \Rightarrow \mathit{state}\ \mathit{set}\ \mathit{acom} \\
\mathit{CS}\ c = \mathit{lfp}\ (\mathit{step}\ \mathit{UNIV})\ c
\end{array}
$$

where UNIV is the set of all elements of a type, in this case the set of all states. That is, the set of initial states are all states. This is a standard choice but any other set is equally possible.

4.3 Small-Step Semantics

The collecting semantics can be specialized to a small-step semantics executing a command c starting in a state s: annotate c with $\emptyset$ everywhere, make a single step with initial state set $\{s\}$ (now s has been "injected" into c), but now keep stepping c with empty initial state set:

$$\mathit{steps}\ s\ c\ n = (\mathit{step}\ \emptyset)^n\ (\mathit{step}\ \{s\}\ (\mathit{anno}\ \emptyset\ c))$$

This describes $n+1$ steps of a small-step operational semantics. The resulting command will take one of two forms: either it is annotated with $\emptyset$ everywhere, which means that the execution terminated and the state has "dropped out" at the end; or it contains exactly one non-empty annotation, which is a singleton $\{s'\}$ that shows exactly where the execution currently is.

We can animate the small-step semantic just like the full collecting semantics, by evaluating *steps*. The output below is generated by executing

$$\texttt{value}\ \ \mathit{show\text{-}acom}\ [''x'']\ (\mathit{steps}\ (\lambda x.\ 0)\ \mathit{ss\text{-}ex}\ n)$$

in Isabelle, for increasing n, which is very effective in class. The first 4 iterations produce the following output:

$\{\{[(''x'', 0)]\}\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 1)\ DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 2)\ \{\emptyset\}$
$\{\emptyset\}$

$\{\emptyset\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 1)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 2)\ \{\{[(''x'', 2)]\}\}$
$\{\emptyset\}$

$\{\{[(''x'', 2)]\}\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 1)\ DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 2)\ \{\emptyset\}$
$\{\emptyset\}$

$\{\emptyset\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 1)\ DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 2)\ \{\emptyset\}$
$\{\{[(''x'', 2)]\}\}$

One more step, and the single state drops out.

The whole point of this operational semantics is to justify the least-fixed point construction of *CS* with respect to it. More precisely, we show that *CS* overapproximates the operational semantics:

Lemma $steps\ s\ c\ n \leq CS\ c$

The two semantics actually coincide, but we only need one direction. Later we show that the abstract interpreter overapproximates the collecting semantics. Together this proves that the abstract interpreter overapproximates the small-step semantics.

The above small-step semantics is rather non-standard (but attractively simple). Cachera and Pichardie [3] present a proof relating a standard small-step semantics to a collecting semantics. Their proof should carry over to our framework if their program points are simulated by our annotations.

5 Abstract Interpretation

This and the following two sections develop and refine a generic abstract interpreter. Initially, boolean expressions are not analysed. This is corrected in a second step. In a last step, widening and narrowing are added.

5.1 Orderings

The various orderings we need are defined as type classes. The notation $\tau :: C$ means that type τ is of class C.

A type $'a$ is a *preorder* ($'a :: preord$) if there is a reflexive and transitive relation $\sqsubseteq :: 'a \Rightarrow 'a \Rightarrow bool$. We do not assume antisymmetry because we want to cover types with multiple different representations for the same abstract element, e.g. pairs as intervals, where all pairs (l, h) with $h < l$ represent the empty interval.

Any relation $\sqsubseteq$ on type $'a$ extends to type $'a\ acom$ exactly like $\leq$ in the definition of the collecting semantics in Section 4.

Lemma If $'a :: preord$ then $'a\ acom :: preord$.

In Isabelle, such lemmas are expressed as so-called instance statements. They allow the type checker to infer the class of complex types automatically.

Our abstract domains will initially be semilattices. Later we extend them to lattices. A type $'a$ is a *semilattice with top* ($'a :: SL\text{-}top$) if it is a preorder and there is a least upper bound (join) operation $\sqcup :: 'a \Rightarrow 'a \Rightarrow 'a$, i.e.

$$x \sqsubseteq x \sqcup y \qquad y \sqsubseteq x \sqcup y \qquad [\![x \sqsubseteq z;\ y \sqsubseteq z]\!] \Longrightarrow x \sqcup y \sqsubseteq z$$

and there is a top element $\top :: 'a$, i.e. $x \sqsubseteq \top$.

Both *option* and function types preserve semilattices:

Lemma If $'a :: SL\text{-}top$ then $'a\ option :: SL\text{-}top$.

The extension adjoins *None* as the least element.

Lemma If $'a :: SL\text{-}top$ then $'b \Rightarrow 'a :: SL\text{-}top$.

The orderings extends pointwise in the usual manner.

5.2 Abstract Interpretation with Functional Abstract States

We start with an abstract interpreter that operates on abstract states that are functions. It is not yet executable, but a first, conceptually simple design that is made executable in a second step.

The abstract interpreter is parameterized with a type $'av :: SL\text{-}top$ of abstract values that comes with a concretization function γ. In Isabelle this is expressed as a locale:

locale *Val-abs* =
fixes $\gamma :: 'av{::}SL\text{-}top \Rightarrow val\ set$
assumes $a \sqsubseteq b \Longrightarrow \gamma\ a \subseteq \gamma\ b$ **and** $\gamma\ \top = UNIV$

The **fixes** part declares the parameters, the **assumes** part states assumptions on the parameters. As explained in the introduction, we only model half the abstract interpretation theory: we drop the abstraction function α and do not calculate abstract interpreters from concrete ones but merely prove given abstract interpreters correct.

In the context of this locale we define abstract interpreters for *aexp* and *acom*. They operate on a lifted abstract state of type $'av\ st\ option$ where

$'av\ st = vname \Rightarrow 'av$

Type *option* allows us to model unreachable program points by annotating them with *None*, the counterpart to $\emptyset$ in the collecting semantics.

The concretization function γ is extended to $'av\ option\ st\ acom$ in the canonical manner, preserving monotonicity:

$\gamma_f :: 'av\ st \Rightarrow state\ set$
$\gamma_f\ S = \{s \mid \forall x.\ s\ x \in \gamma\ (S\ x)\}$

$\gamma_o :: 'av\ st\ option \Rightarrow state\ set$
$\gamma_o\ None = \emptyset$
$\gamma_o\ (Some\ S) = \gamma_f\ S$

$\gamma_c :: 'av\ st\ option\ acom \Rightarrow state\ set\ acom$
$\gamma_c\ c = map\text{-}acom\ \gamma_o\ c$

where *map-acom f c* applies f to all annotations in c.

Now we come to the actual interpreters. An abstraction of *aval* requires abstractions of the basic arithmetic operations. Hence locale *Val-abs* is actually richer than we pretended above: it contains abstractions of N and *Plus*, too:

fixes $num' :: val \Rightarrow 'av$
assumes $n \in \gamma\ (num'\ n)$
fixes $plus' :: 'av \Rightarrow 'av \Rightarrow 'av$
assumes $[\![n_1 \in \gamma\ a_1;\ n_2 \in \gamma\ a_2]\!] \Longrightarrow n_1 + n_2 \in \gamma\ (plus'\ a_1\ a_2)$

The abstract interpreter for *aexp* is standard

$aval' :: aexp \Rightarrow 'av\ st \Rightarrow 'av$

$aval'\ (N\ n)\ S = num'\ n$
$aval'\ (V\ x)\ S = S\ x$
$aval'\ (Plus\ a_1\ a_2)\ S = plus'\ (aval'\ a_1\ S)\ (aval'\ a_2\ S)$

and its correctness ($s \in \gamma_f\ S \Longrightarrow aval\ a\ s \in \gamma\ (aval'\ a\ S)$) is trivial.

The abstract interpreter for annotated commands is defined like the collecting semantics in two stages. We start with an abstraction of *step*, where the notation $f(x := y)$ is predefined and means function update:

$step' :: 'av\ st\ option \Rightarrow 'av\ st\ option\ acom \Rightarrow 'av\ st\ option\ acom$

$step'\ S\ (SKIP\ \{P\}) = SKIP\ \{S\}$
$step'\ S\ (x ::= e\ \{P\})$
$= x ::= e\ \{\mathsf{case}\ S\ \mathsf{of}\ None \Rightarrow None \mid Some\ S \Rightarrow Some\ (S(x := aval'\ e\ S))\}$
$step'\ S\ (c_1;\ c_2) = step'\ S\ c_1;\ step'\ (post\ c_1)\ c_2$
$step'\ S\ (IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{P\})$
$= IF\ b\ THEN\ step'\ S\ c_1\ ELSE\ step'\ S\ c_2\ \{post\ c_1 \sqcup post\ c_2\}$
$step'\ S\ (\{Inv\}\ WHILE\ b\ DO\ c\ \{P\})$
$= \{S \sqcup post\ c\}\ WHILE\ b\ DO\ step'\ Inv\ c\ \{Inv\}$

Correctness of *step'* wrt *step* is proved by induction on c:

Lemma If $S \subseteq \gamma_o\ S'$ and $c \leq \gamma_c\ c'$ then $step\ S\ c \leq \gamma_c\ (step'\ S'\ c')$

The abstract interpreter is defined by fixed-point iteration of $step'$. This raises the termination question. Because proof assistants like Coq and Isabelle/HOL build on logics of total functions, previous formalizations (e.g. the work by Pichardie) built the termination requirement into the ordering $\sqsubseteq$. We define the iteration for arbitrary orderings and prove termination separately. The slight advantage in a teaching context is that it allows us to postpone the discussion of termination. Our trick is to use $while\text{-}option :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$ from the Isabelle/HOL library. It satisfies the recursion equation

$$while\text{-}option\ b\ c\ s = (\mathsf{if}\ b\ s\ \mathsf{then}\ while\text{-}option\ b\ c\ (c\ s)\ \mathsf{else}\ Some\ s)$$

which makes it executable. Mathematically, $while\text{-}option\ b\ c\ s = None$ in case the recursion does not terminate. We define a generic post-fixed point finder

$$\begin{array}{l} pfp :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option \\ pfp\ f = while\text{-}option\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ f \end{array}$$

and as a special case the abstract interpreter:

$$\begin{array}{l} AI :: com \Rightarrow 'av\ st\ option\ acom\ option \\ AI\ c = pfp\ (step'\ \top)\ (\bot_c\ c) \end{array}$$

where $\bot_c = anno\ None$ (note that $\bot_c$ is one symbol). Iteration starts with $\bot_c\ c$, the least annotated version of c, thus making sure we obtain the least post-fixed point (if f is monotone). This is nice to know, but not used later on: for correctness, any post-fixed point will do. We iterate $step'\ \top$, corresponding to $step\ UNIV$ in the collecting semantics.

Theorem (Correctness of AI wrt CS) $AI\ c = Some\ c' \Longrightarrow CS\ c \leq \gamma_c\ c'$

It follows essentially because CS is defined as the *least* (post-)fixed point, AI returns a post-fixed point, and $step'$ and $step$ operate in lock-step.

This is the initial version of our generic abstract interpreter. Unfortunately it is not executable: in each iteration of pfp we need to test if the old and the new version of the annotated command are related by $\sqsubseteq$. This in turn requires us to compare all annotations, which are (optional) functions. But $\sqsubseteq$ on functions is not computable if the domain is infinite, which $vname$ is. Before we fix this, a remark on monotonicity.

So far, monotonicity at the abstract level has not entered the picture: it is not needed for correctness of the basic abstract interpreter but will be required for termination. We define an extension of locale *Val-abs* (locales are hierarchical) where we also assume monotonicity of the abstract operations

$$\textbf{assumes}\ [\![a_1 \sqsubseteq b_1;\ a_2 \sqsubseteq b_2]\!] \Longrightarrow plus'\ a_1\ a_2 \sqsubseteq plus'\ b_1\ b_2$$

and call this the *monotone framework*. In this framework we can prove monotonicity of $step'$:

Lemma If $S \sqsubseteq S'$ and $c \sqsubseteq c'$ then $step'\ S\ c \sqsubseteq step'\ S'\ c'$.

5.3 Abstract Interpretation with Computable Abstract States

We replace *vname* $\Rightarrow$ *'av* by finite functions because the state only needs to record values of variables that actually occur in the command being analysed. We could parameterize our abstract interpreter wrt a type of finite functions [12], but since we do not intend to provide multiple implementations, we fix a particularly simple model and redefine *'a st* as follows:

datatype *'a st* = *FunDom* (*vname* $\Rightarrow$ *'a*) (*vname list*)

That is, we record the domain of the finite function as a list. The two projection functions are *fun* (*FunDom f xs*) = *f* and *dom* (*FunDom f xs*) = *xs*. Function update is easy:

update F x y =
FunDom ((*fun F*)(*x* := *y*)) (*if x* $\in$ *set* (*dom F*) *then dom F else x*·*dom F*)

where *set* converts a list into a set and "·" is Cons. Function application is called *lookup* and requires *'a* to have a $\top$ element which is returned outside the domain:

lookup F x = (*if x* $\in$ *set* (*dom F*) *then fun F x else* $\top$)

Why $\top$? This reflects that our analysis assumes that uninitialized variables can have arbitrary values.

Lemma If *'a* :: *SL-top* then *'a st* :: *SL-top*.

The ordering is again pointwise (but expressed with lookup). The join intersects the domains because outside the domain lookup returns $\top$.

The development of the abstract interpreter stays exactly the same, except that application and update on type *'av st* are called *lookup* and *update*. We have arrived at our first executable abstract interpreter. The initial development in terms of abstract states as functions was merely presented for didactic reasons, to keep it as simple as possible and introduce improvements gradually.

In addition we also prove a generic termination theorem. It is phrased directly in terms of measures because this is most convenient for our applications. In the context of the monotone framework (see end of previous subsection) we obtain

Theorem $\exists\, c'.$ *AI c* = *Some c'* if there is a measure *m* :: *'av* $\Rightarrow$ *nat* such that $x \sqsubseteq y \wedge \neg\, y \sqsubseteq x \longrightarrow m\ y < m\ x$ and $x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow m\ x = m\ y$.

The fact that *while-option b f x* = *Some y* means termination follows from the recursion equation for *while-option* (see above) together with the fact that *while-option b f x* = *None* in case *b* (f^k *x*) for all *k*.

6 Backward Analysis of Boolean Expressions

So far we have not analyzed boolean expressions at all. Now we take them into account by defining an analysis that "filters" an abstract state *S* wrt some

boolean expression b and some intended result r of b: the resulting abstract state S' should be more precise than S, i.e. $\gamma_o\ S' \subseteq \gamma_o\ S$, but no state that makes b evaluate to r must be lost: if $s \in \gamma_o\ S$ and $bval\ b\ s = r$ then also $s \in \gamma_o\ S'$. This filtering of abstract states corresponds to an intersection and is realized by the dual of the join, the *meet*. We also need to model the situation that some variable has no possible value, which corresponds to a least abstract element $\bot$. Therefore we upgrade from a semilattice to a lattice. A type $'a$ is a *lattice with top and bottom* ($'a$ :: *L-top-bot*) if it is a semilattice with top and there is a greatest lower bound (meet) operation $\sqcap :: 'a \Rightarrow 'a \Rightarrow 'a$, i.e.

$$x \sqcap y \sqsubseteq x \qquad x \sqcap y \sqsubseteq y \qquad [\![x \sqsubseteq y;\ x \sqsubseteq z]\!] \Longrightarrow x \sqsubseteq y \sqcap z$$

and there is a bottom element $\bot :: 'a$, i.e. $\bot \sqsubseteq x$.

We specialize the *Val-abs* interface further by requiring $'av$::*L-top-bot* and by adding two further assumptions:

assumes $\gamma\ a_1 \cap \gamma\ a_2 \subseteq \gamma\ (a_1 \sqcap a_2)$ **and** $\gamma\ \bot = \emptyset$

The first assumption actually implies $\gamma\ (a_1 \sqcap a_2) = \gamma\ a_1 \cap \gamma\ a_2$. Moreover we require abstract filter functions for all basic arithmetic and boolean operations:

fixes *test-num'* :: *int* $\Rightarrow$ $'av$ $\Rightarrow$ *bool*
fixes *filter-plus'* :: $'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$
fixes *filter-less'* :: *bool* $\Rightarrow$ $'av \Rightarrow 'av \Rightarrow 'av \times 'av$
assumes *test-num'* $n\ a = (n \in \gamma\ a)$
assumes *filter-plus'* $a\ a_1\ a_2 = (b_1,\ b_2) \Longrightarrow$
$[\![\ n_1 \in \gamma\ a_1;\ n_2 \in \gamma\ a_2;\ n_1 + n_2 \in \gamma\ a\ ]\!] \Longrightarrow n_1 \in \gamma\ b_1 \wedge n_2 \in \gamma\ b_2$
assumes *filter-less'* $(n_1 < n_2)\ a_1\ a_2 = (b_1,\ b_2) \Longrightarrow$
$[\![\ n_1 \in \gamma\ a_1;\ n_2 \in \gamma\ a_2\ ]\!] \Longrightarrow n_1 \in \gamma\ b_1 \wedge n_2 \in \gamma\ b_2$

The filter functions are similar to inverse functions: but instead of computing the arguments from the result, they are given both the arguments and the result and should return the filtered arguments where values that cannot lead to the given result may be removed. The **assumes** clauses express it the other way around: the *filter-plus'* clause says that values in the conretization of a_1 and a_2 that lead into $\gamma\ a$ must not be filtered out. This assumptions guarantees soundness. Based on the basic filtering functions we can now filter wrt *aexp* and later *bexp* as explained in the introduction of this section:

afilter :: *aexp* $\Rightarrow$ $'av$ $\Rightarrow$ $'av$ *st option* $\Rightarrow$ $'av$ *st option*

afilter $(N\ n)\ a\ S =$ (*if test-num'* $n\ a$ *then* S *else None*)
afilter $(V\ x)\ a\ S =$
(*case* S *of None* $\Rightarrow$ *None*
| *Some* $S \Rightarrow$
  let $a' =$ *lookup* $S\ x \sqcap a$
  in if $a' \sqsubseteq \bot$ *then None else Some* (*update* $S\ x\ a'$))
afilter $(Plus\ e_1\ e_2)\ a\ S =$
(*let* $(a_1,\ a_2) =$ *filter-plus'* a (*aval''* $e_1\ S$) (*aval''* $e_2\ S$)
in afilter $e_1\ a_1$ (*afilter* $e_2\ a_2\ S$))

where $aval''$ is just a lifted version of $aval'$:

$$aval''\ e\ None = \bot$$
$$aval''\ e\ (Some\ S) = aval'\ e\ S$$

Note that the test $a' \sqsubseteq \bot$ in the *afilter* $(V\ x)$ clause prevents an imprecision. We could always return $Some\ (update\ S\ x\ a')$, as some authors do [14]. But if a' is $\bot$, this is really an unreachable state. However, this information can be overwritten in subsequent assignments, and when the resulting state is joined with another execution path, e.g. at the end of a conditional, the unreachable state can lead to a loss of precision. Hence we avoid creating states with $\bot$ components and work with the least state *None* instead.

Filtering with *bexp* is similar:

$$bfilter :: bexp \Rightarrow bool \Rightarrow 'av\ st\ option \Rightarrow 'av\ st\ option$$

$$bfilter\ (Bc\ v)\ res\ S = (\mathsf{if}\ v = res\ \mathsf{then}\ S\ \mathsf{else}\ None)$$
$$bfilter\ (Not\ b)\ res\ S = bfilter\ b\ (\neg\ res)\ S$$
$$bfilter\ (And\ b_1\ b_2)\ res\ S =$$
$$(\mathsf{if}\ res\ \mathsf{then}\ bfilter\ b_1\ True\ (bfilter\ b_2\ True\ S)$$
$$\ \mathsf{else}\ bfilter\ b_1\ False\ S \sqcup bfilter\ b_2\ False\ S)$$
$$bfilter\ (Less\ e_1\ e_2)\ res\ S =$$
$$(\mathsf{let}\ (res_1,\ res_2) = filter\text{-}less'\ res\ (aval''\ e_1\ S)\ (aval''\ e_2\ S)$$
$$\ \mathsf{in}\ afilter\ e_1\ res_1\ (afilter\ e_2\ res_2\ S))$$

Note that the then-case in $bfilter\ (And\ b_1\ b_2)$ is a tricky way to express $bfilter\ b_1\ True \sqcap bfilter\ b_2\ True$, thus obviating the need to define $\sqcap$ on abstract states. It is debatable if this trick is a good idea in a teaching context.

Two of the defining equations for $step'$ are now refined

$$step'\ S\ (IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{P\}) =$$

$$IF\ b\ THEN\ step'\ (bfilter\ b\ True\ S)\ c_1\ ELSE\ step'\ (bfilter\ b\ False\ S)\ c_2$$
$$\{post\ c_1 \sqcup post\ c_2\}$$

$$step'\ S\ (\{Inv\}\ WHILE\ b\ DO\ c\ \{P\}) =$$

$$\{S \sqcup post\ c\}$$
$$WHILE\ b\ DO\ step'\ (bfilter\ b\ True\ Inv)\ c$$
$$\{bfilter\ b\ False\ Inv\}$$

but the definition of the abstract interpreter AI itself is unchanged. The correctness proof stays largely the same but requires two new lemmas:

Lemma If $s \in \gamma_o\ S$ and $aval\ e\ s \in \gamma\ a$ then $s \in \gamma_o\ (afilter\ e\ a\ S)$.

Lemma If $s \in \gamma_o\ S$ then $s \in \gamma_o\ (bfilter\ b\ (bval\ b\ s)\ S)$.

7 Widening and Narrowing

Widening is meant to ensure termination of fixed point iteration even in lattices of infinite height, eg intervals. More generally, it is meant to accelate convergence. Instead of computing $f^i(\bot)$ for $i = 0, 1, \ldots$ until a post-fixed point is found (see *pfp*), widening allows us to take bigger steps thus avoiding nontermination. These bigger steps may lose precision. Narrowing, another iteration, is meant to regain it.

A widening operator $\triangledown$ has type $'a \Rightarrow 'a \Rightarrow 'a$ and satisfies $x \sqsubseteq x \triangledown y$ and $y \sqsubseteq x \triangledown y$. A narrowing operator $\triangle$ has type $'a \Rightarrow 'a \Rightarrow 'a$ and satisfies $y \sqsubseteq x \Longrightarrow y \sqsubseteq x \triangle y$ and $y \sqsubseteq x \Longrightarrow x \triangle y \sqsubseteq x$. For convenience we put both of them in class *WN* and make it a subclass of *SL-top*.

Normally the axioms of widening and narrowing also include an ascending chain condition. We have again chosen to separate the termination argument. (Strictly speaking, widening would not need any axioms for correctness but only for termination.) Both operators can be extended to type *option* and *st*:

Lemma If $'a :: WN$ then $'a\ st :: WN$.

Lemma If $'a :: WN$ then $'a\ option :: WN$.

For the didactic reason of simplicity we have chosen not to apply widening or narrowing selectively at individual annotations but simultaneously everywhere. This can be less precise than more selective strategies [3] but is much simpler.

We define a function $map2\text{-}acom :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom$ that applies a function simultaneously to the corresponding annotations of two *strip*-equal annotated commands. This permits us to lift $\triangledown$ and $\triangle$ to $\triangledown_c$ and $\triangle_c$ on annotated commands: $c_1 \triangledown_c c_2 = map2\text{-}acom\ (op \triangledown)\ c_1\ c_2$ and $c_1 \triangle_c c_2 = map2\text{-}acom\ (op \triangle)\ c_1\ c_2$, where $(op \bowtie)$ is the function some infix operator $\bowtie$ stands for.

Iterative widening and narrowing on *acom* are expressed as loops:

$$iter\text{-}widen\ f = while\text{-}option\ (\lambda c.\ \neg\ f\ c \sqsubseteq c)\ (\lambda c.\ c \triangledown_c f\ c)$$
$$iter\text{-}narrow\ f = while\text{-}option\ (\lambda c.\ \neg\ c \sqsubseteq c \triangle_c f\ c)\ (\lambda c.\ c \triangle_c f\ c)$$

This formalizes one of the widening variants proposed by Cousot [6, footnote 6]. Pichardie and Monniaux [7] propose other formalizations.

The overall analysis performs widening first and then narrowing:

$$pfp\text{-}wn\ f\ c =$$
$$(\mathsf{case}\ iter\text{-}widen\ f\ (\bot_c\ c)\ \mathsf{of}\ None \Rightarrow None \mid Some\ c' \Rightarrow iter\text{-}narrow\ f\ c')$$

Later we show that the *None* case cannot arise under certain assumptions about widening. By definition, $iter\text{-}widen\ f\ (\bot_c\ c)$ finds a post-fixed point c' of f if it terminates. Assuming f is monotone, induction together with the narrowing properties shows that $iter\text{-}narrow\ f\ c'$ finds another post-fixed point of f below c' if it terminates.

In the context of the monotone framework we define *AI-wn* with the help of *pfp-wn* instead of *pfp*, as previously:

$$AI\text{-}wn = pfp\text{-}wn\ (step'\ \top)$$

The correctness ($AI\text{-}wn\ c = Some\ c' \Longrightarrow CS\ c \le \gamma_c\ c'$) proof is as before.

7.1 Termination

Correctness of widening and narrowing was easy. Termination is quite technical, although we have adopted an approach that does not refer to infinite chains but is phrased in terms of measure functions. For widening, each type needs to come with a measure function m into *nat* such that

$$x \sqsubseteq y \Longrightarrow m\ y \le m\ x$$
$$\neg\ y \sqsubseteq x \Longrightarrow m\ (x \bigtriangledown y) < m\ x$$

The first measure property guarantees that the measure cannot go up with a widening step: the first widening axiom implies $m\ (x \bigtriangledown y) \le m\ x$ (the second widening axiom is never needed). The second measure property guarantees that with every widening step of *iter-widen*, the measure goes down. The second property is the one we need, the first one is only auxiliary.

Both measure properties together allow us to lift them to composite data types, especially abstract states and annotated commands. Both types are just glorified tuples and hence we can explain the mechanism in terms of pairs without having to bother with the technical details of the more complex types. Everything on pairs is defined componentwise, including the measure function and the function f whose post-fixed point we seek:

$$((y_1,\ y_2) \sqsubseteq (x_1,\ x_2)) = (y_1 \sqsubseteq x_1 \wedge y_2 \sqsubseteq x_2)$$
$$(x_1,\ x_2) \bigtriangledown (y_1,\ y_2) = (x_1 \bigtriangledown y_1,\ x_2 \bigtriangledown y_2)$$
$$m\ (x_1,\ x_2) = m_1\ x_1 + m_2\ x_2$$
$$f\ (x_1,\ x_2) = (f_1\ x_1,\ f_2\ x_2)$$

The first measure property, anti-monotonicity, lifts trivially to pairs. Let us now consider the second measure property and assume $\neg\ f\ (x_1,\ x_2) \sqsubseteq (x_1,\ x_2)$, i.e. either $\neg\ f_1\ x_1 \sqsubseteq x_1$ or $\neg\ f_2\ x_2 \sqsubseteq x_2$. In the first case we have $m_1(x_1 \bigtriangledown f_1\ x_1) < m_1\ x_1$ (by the second measure property) and $m_2(x_2 \bigtriangledown f_2\ x_2) \le m_2\ x_2$ (by the first measure property) and thus $m\ ((x_1,\ x_2) \bigtriangledown f\ (x_1,\ x_2)) = m_1\ (x_1 \bigtriangledown f_1\ x_1) + m_2\ (x_2 \bigtriangledown f_2\ x_2) < m_1\ x_1 + m_2\ x_2$. The second case is dual.

This way we can lift the two measure properties from the basic domain of abstract values up to annotated commands. However, there are some technicalities. The x and y in the measure properties need to fulfill additional invariants, in particular at the *acom* level: both must be *strip*-equal annotated commands over the same fixed finite set of variables. Hence the full measure theorem becomes

> If *finite* X, $strip\ c' = strip\ c$, $c \in Com\ X$, $c' \in Com\ X$ and $\neg\ c' \sqsubseteq c$, then $m\ (c \bigtriangledown_c c') < m\ c$.

where m is the measure function on *acom* and $Com\ X$ is the set of commands whose annotations mention only variables in X. Of course $step'$ preserves these invariants.

Termination of narrowing is proved in a similar manner, using measure functions called n that must also satisfy two properties:

$x \sqsubseteq y \Longrightarrow n\ x \leq n\ y$
$y \sqsubseteq x \Longrightarrow \neg\ x \sqsubseteq x \vartriangle y \Longrightarrow n(x \vartriangle y) < n\ x$

Again, the first property lifts trivially but it is the second one we are really after. It is lifted to pairs in a similar manner as for widening, using the second narrowing axiom. Obtaining the final measure theorem for narrowing on the *acom* level is again technical in the same way as for widening. At the end of the day, here is the unconditional termination statement for *AI-ivl'*, the instantiation of *AI-wn* with intervals:

Theorem $\exists\, c'.\ \mathit{AI\text{-}ivl'}\ c = \mathit{Some}\ c'$

7.2 Intervals

We have instantiated the various frameworks above with the standard analyses, in particular intervals. Our definition of intervals is extremely basic:

datatype *ivl* = *I* (*int option*) (*int option*)

where *None* represents infinity. For readability we install some syntactic sugar: $\{i...j\}$ stands for *I* (*Some i*) (*Some j*); infinite lower or upper bounds are simply dropped. For example, $\{i...\}$ is *I* (*Some i*) *None*. The only drawback is that the empty interval has many representations, but this is why our value abstraction is based on preorders, not partial orders. We refrain from giving the details of the operations on intervals. They follow the literature, except for the representation.

Just like for the small-step semantics, we can animate the computation of the abstract interpreter by iterating the step function and widening/narrowing. We evaluate *show-acom* $((\lambda c.\ c \bigtriangledown_c \mathit{step\text{-}ivl}\ \top\ c)^n\ (\bot_c\ \mathit{testc}))$ for increasing n. The pretty-printing function *show-acom* shows an abstract state as a list of pairs (*x*,*ivl*) — no need to supply the list of variables, it is part of the abstract state.

For $n = 1$ we obtain the program annotated with *None* everywhere except after the first assignment:

$''x'' ::= N\ 7\ \{\mathit{Some}\ [(''x'', \{7...7\})]\};$
$\{\mathit{None}\}$
$\mathit{WHILE}\ \mathit{Less}\ (V\ ''x'')\ (N\ 100)\ \mathit{DO}\ ''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 1)\ \{\mathit{None}\}$
$\{\mathit{None}\}$

The next step merely initializes the invariant:

$''x'' ::= N\ 7\ \{\mathit{Some}\ [(''x'', \{7...7\})]\};$
$\{\mathit{Some}\ [(''x'', \{7...7\})]\}$
$\mathit{WHILE}\ \mathit{Less}\ (V\ ''x'')\ (N\ 100)\ \mathit{DO}\ ''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 1)\ \{\mathit{None}\}$
$\{\mathit{None}\}$

Now the invariant filtered with the loop condition is propagated to the end of the loop body:

$''x'' ::= N\ 7\ \{Some\ [(''x'', \{7 \ldots 7\})]\};$
$\{Some\ [(''x'', \{7 \ldots 7\})]\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ [(''x'', \{8 \ldots 8\})]\}$
$\{None\}$

In the next step, widening has an effect and combines $\{7 \ldots 7\}$ and $\{8 \ldots 8\}$ into the new invariant $\{7 \ldots\}$:

$''x'' ::= N\ 7\ \{Some\ [(''x'', \{7 \ldots 7\})]\};$
$\{Some\ [(''x'', \{7 \ldots\})]\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ [(''x'', \{8 \ldots 8\})]\}$
$\{None\}$

One more iteration yields a (post-)fixed point of *step-ivl*:

$''x'' ::= N\ 7\ \{Some\ [(''x'', \{7 \ldots 7\})]\};$
$\{Some\ [(''x'', \{7 \ldots\})]\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ [(''x'', \{8 \ldots\})]\}$
$\{Some\ [(''x'', \{100 \ldots\})]\}$

Switching to narrowing now, we obtain a second (post-)fixed point of *step-ivl* after 3 more iterations:

$''x'' ::= N\ 7\ \{Some\ [(''x'', \{7 \ldots 7\})]\};$
$\{Some\ [(''x'', \{7 \ldots 100\})]\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ [(''x'', \{8 \ldots 100\})]\}$
$\{Some\ [(''x'', \{100 \ldots 100\})]\}$

8 Conclusion

The above material was covered in 4 weeks in an MSc course on semantics via a theorem prover. Much of it worked well, although a few points are still a bit technical. In particular, we did not cover termination formally, especially for widening/narrowing. We intend to streamline this issue further in the future.

The Isabelle theories are available online at http://isabelle.in.tum.de/dist/library/HOL/HOL-IMP/ (the relevant theories are named `*ITP`) and in the Isabelle distribution in `src/HOL/IMP/Abs_Int_ITP/`.

Acknowledgement David Pichardie's exemplary review and his many explanations greatly improved my understanding of his work and of abstract interpretation in general. Brian Huffman's comments improved the presentation.

References

1. Bertot, Y.: Structural abstract interpretation: A formal study using Coq. In: Bove, Barbosa, Pardo, Pinto (eds.) Language Engineering and Rigorous Software Development (ALFA Summer School). Lect. Notes in Comp. Sci., vol. 5520, pp. 153–194. Springer-Verlag (2008)
2. Bertot, Y., Grégoire, B., Leroy, X.: A structured approach to proving compiler optimizations based on dataflow analysis. In: Types for Proofs and Programs (TYPES 2004). Lect. Notes in Comp. Sci., vol. 3839, pp. 66–81. Springer-Verlag (2006)
3. Cachera, D., Pichardie, D.: A certified denotational abstract interpreter. In: Kaufmann, M., Paulson, L. (eds.) Interactive Theorem Proving (ITP 2010). Lect. Notes in Comp. Sci., vol. 6172, pp. 9–24. Springer-Verlag (2010)
4. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, Steinbrüggen (eds.) Calculational System Design. IOS Press (1999)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM Symp. Principles of Programming Languages. pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) Programming Language Implementation and Logic Programming (PLILP '92). Lect. Notes in Comp. Sci., vol. 631, pp. 269–295. Springer-Verlag (1992)
7. Monniaux, D.: A minimalistic look at widening operators. Higher-Order and Symbolic Computation 22, 145–154 (2009)
8. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
9. Nipkow, T.: Verified bytecode verifiers. In: Honsell, F. (ed.) Foundations of Software Science and Computation Structures (FOSSACS 2001). Lect. Notes in Comp. Sci., vol. 2030, pp. 347–363. Springer-Verlag (2001)
10. Nipkow, T.: Teaching semantics with a proof assistant: No more LSD trip proofs. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI 2012). Lect. Notes in Comp. Sci., vol. 7148, pp. 24–38. Springer-Verlag (2012)
11. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lect. Notes in Comp. Sci., vol. 2283. Springer-Verlag (2002)
12. Pichardie, D.: Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés. Ph.D. thesis, Université Rennes 1 (2005)
13. Pichardie, D.: Building certified static analysers by modular construction of well-founded lattices. In: Proc. 1st International Conference on Foundations of Informatics, Computing and Software (FICS'08). ENTCS, vol. 212, pp. 225–239 (2008)
14. Seo, S., Yang, H., Yi, K.: Automatic construction of Hoare proofs from abstract interpretation results. In: Ohori, A. (ed.) Programming Languages and Systems (APLAS 2003). Lect. Notes in Comp. Sci., vol. 2895, pp. 230–245. Springer-Verlag (2003)