Abstract Interpretation with a Theorem Prover

Hugh Anderson

Department of Computer Science, National University of Singapore
hugh@comp.nus.edu.sg

Abstract This paper presents an approach to the implementation of the *abstract interpretation* style of program analysis by first constructing a logic for representing the process of abstract analysis, and then embedding this logic in the theorem prover **HOL**. Programs to be analysed undergo a two-phase process, first being mechanically transformed to an analysis *model*, and then this being used to test or verify program properties. A specific advantage of this approach is that it allows abstract interpretation to be used in a consistent framework with other analysis methods, such as Hoare Logic or exhaustive state space analysis.

1 Introduction

Software developments are often so complex that program developers are unsure of the behaviour of code they have constructed. Testing, though useful, cannot guarantee the behaviour of developed code unless the testing is exhaustive, and this is generally not possible for large software developments.

An alternative strategy is to attempt to confirm behaviours of code by analysis using representations of the semantics of the code components. In one approach, the operation of the program is represented in an abstract manner, and mathematical techniques are used to derive properties of the code. These properties may be considered to be partial specifications of the code. Examples of this approach include Hoare reasoning [8], and abstract interpretation, elaborated by Cousot and Halbwachs in [3]. Graf and Saidi have presented a method in [10] which automatically constructs abstract state graphs suitable for checking with a model checker.

This paper explores a transformational approach to analysis within a unified program development environment, by constructing a logic for analysis, coding this logic as a shallow embedding in the theorem prover **HOL** [5], and then using this to derive an efficient analysis model from a program. This model has a functional form and may be used to test and verify properties of a program.

This paper has the following structure: Sections 2 and 3 briefly introduce abstract interpretation and the mechanical theorem prover **HOL**. Section 4 presents elements of a logic for abstract interpretation analysis, showing a sample analysis of a program within the logic. Section 5 shows sample codings of logic elements in **HOL**, and section 6 is the conclusion.

2 Abstract Interpretation

The technique of abstract interpretation approximates the *exact* analysis of programs, by reasoning on some abstract semantics of the program. An example of abstract interpretation is found in the analysis of the semantics of a program restricted over a representation of program state given as a set of linear inequalities or equalities between the variables of the program. For example, consider two unsigned integer variables $x \geq 2$ and y and the assignments:

$$y := (x * x) + 1;$$
$$x := x + x;$$

An *exact* static analysis of the state of the program variables after these assignments may involve keeping track of a series of (x, y) pairs: $\{(4,5), (8,17), ...\}$ (depending on other program elements). However, it may also be given as a set of linear inequalities, written as:

$$\{y \geq x + 1\}$$

Note that this assertion is always true, but it tells us less about the behaviour of the assignment statements. The reason for doing this sort of software approximation is that further analysis on the machine state may be less computationally expensive.

Linear programming functions such as *convex-hull* and *projection* are also useful in this context. Consider the analysis of program state $\{Q\}$ at the beginning of this do-loop[1]:

$$\{y \geq 1 \wedge x = 2\}$$
do $\{Q\}B \rightarrow$

$$y := (x * x) + 1;$$
$$x := x + x;$$

od

If we were to consider the case of an *exact* representation of this state, $\{Q\}$ would either be (initially) $\{y \geq 1 \wedge x = 2\}$ or (on successive iterations) the values $(4,5), (8,17), ...$, which may be represented by the inequality $y \geq x + 1$.

In the case of the approximate abstract interpretation of this state, the convex hull of the equations $y \geq 1 \wedge x = 2$ and $y \geq x + 1$ may be used instead. We interpret this graphically in Figure 1, where the spaces represented by the two sets of equations are shown, and an enclosing (convex hull) half-space.

The convex hull for a larger set of inequalities involving large numbers of variables may be efficiently calculated using a linear programming software library such as the **cddlib** package found in Fukuda [4].

[1] Note that here we switch between predicate and set representations of the linear inequalities when the meaning is clear, writing $\{y \geq 1 \wedge x = 2\}$ for the two linear equations $\{y \geq 1, x = 2\}$.

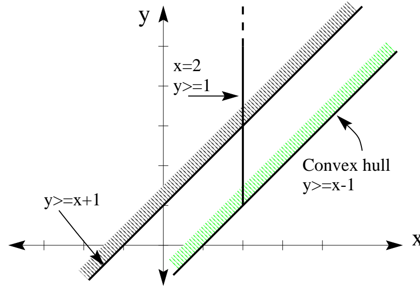

Figure 1. The convex hull of $y \geq 1 \wedge x = 2$ and $y \geq x + 1$ is the half-space $y \geq x - 1$

There is also a graphical interpretation of *projection*, where equations in an "n"-dimensional space are projected onto an "$n - 1$"-dimensional space. For example, the projection of $y \geq 1 \wedge x = 2$ onto y ($y = x^{\perp}$, the orthogonal complement of x) is $y \geq 1$.

In this paper, the particular abstract interpretation style described and implemented in **HOL** is the one just outlined, with the program state represented as sets of linear inequalities, and transformations using convex hull and projection.

3 The HOL Theorem Prover

HOL is a theorem prover assistant written by Mike Gordon in the mid 1980s, and derived from Milner's LCF [6]. **HOL** is implemented in the language **ML**, and it is common to develop **HOL** systems in a blend of **HOL** and **ML**, here called **HOL/ML**. **HOL** provides tools which only allow the construction of theorems which follow from original axioms and definitions. The core of **HOL** consists of 8 inference rules and 5 axioms, and all later proofs and theories are derived from these.

HOL supports both a forward proof style, in which we construct new theorems from existing ones by constructing functions in **HOL/ML** with existing theorems and axioms as parameters, and a backward proof style, in which we set up a goal, and then break it into (separately proved) subgoals. In **HOL**, the steps made during this backward reasoning process are called tactics.

3.1 Extending Theories in HOL

HOL operates in two modes, *draft* and *proof* mode. In draft mode it is possible to introduce (possibly) inconsistent axioms. In proof mode, this is not possible. Theories are extended through the addition of types, constants and inference rules, and this may lead to inconsistent theories. Back and Wright [2] discuss the use of conservative extensions to a theory to ensure consistency:

> *"The advantage of a conservative extension is that it has a standard model whenever the original theory has one. This means that a conservative extension can never introduce inconsistency."*

The approach taken here is a mix; we include underived elements into the theory to access external efficient libraries, but after each use, a check is done of the derived result, to see if it is still consistent with the original.

4 A Logic for Analysis

In the logic presented here, programs may be represented in both a conventional *specification* style using pre and postcondition state representations, and in an *executable* style with a simple imperative language, along the lines given in [9]. For example the Morgan notation

$v\colon[\mathsf{true}\ ,\ v > 10]$

specifies that in the frame[2] v, for any precondition, the postcondition should be $v > 10$. This style of specification is commonly used in the context of the Refinement Calculus to construct code from specifications. For example, we may *refine* this specification using an *assignment introduction* refinement rule:

$\sqsubseteq$ Assignment introduction
$v := 11$

The $\sqsubseteq$ symbol in $S \sqsubseteq C$ indicates that C is a *refinement* of S. The refinement relation ordering is defined in weakest precondition terms by the requirement for $\mathrm{post}(C) \Rightarrow \mathrm{post}(S)$ for initial states satisfying $\mathrm{pre}(S)$.

We introduce a similar logic for analysis, adopting similar notation, although our rules operate in a reverse manner, tending to derive (more abstract) specifications from code. The $\sqsupseteq$ symbol in $C \sqsupseteq S$ indicates that S is an *abstraction* of C. The abstraction relation ordering is the converse of refinement. If $C \sqsupseteq S$ and $C \sqsubseteq S$, then $C \equiv S$.

As an example, the previous code segment might be transformed for the purpose of program analysis, using one of many possible rules, as follows:

$\equiv$ Equivalent assignment postcondition introduction
$v\colon[P\ ,\ \exists v' : P[v'/v] \wedge v = 11[v'/v]]$

The meaning attached to the notation $P[v'/v]$ is that of substitution, replacing each free occurrence of the variable v in an arbitrary formula P with the expression v'. Our example may be simplified to:

$\equiv$ Simplification
$v\colon[P\ ,\ \exists v' : P[v'/v] \wedge v = 11]$

This may not seem all that interesting, however, it may be used in the context of program analysis by unifying P with true to get

$\sqsupseteq$ Unification - strengthen precondition/weaken postcondition
$v\colon[\mathsf{true}\ ,\ v = 11]$

[2] A *frame* indicates those variables or state elements that may change.

That is - an assertion that the code segment $v := 11$ has a property matched by the specification $v\text{:}\left[\mathsf{true}\ ,\ v = 11\right]$. This analysis is one of many that might be performed on assignment code, and reflects a strongest postcondition style of program analysis. In this paper, two sample types of analysis rules are described:

1. **Equivalence analysis rules** - $\equiv$ - In this sort of transformation rule, the program state is represented by predicates over the program variables, and program statements are transformed in relation to the way in which they transform these predicates.
2. **Abstraction analysis rules** - $\sqsupseteq$ - In this sort of transformation rule, an abstraction of the program state is represented by a set of linear inequalities reflecting the relationships between the variables in the program. Program statements are transformed in relation to the way they transform these linear inequalities.

The $\sqsupseteq$ abstraction transformations are used to transform either a code segment, or a specification, to some form more amenable to analysis. For example, we may only be interested in the relative values of variables in a program, and not their absolute values, or we may be only interested in a subset of the variables.

The $\equiv$ equivalence transformation rules are used to transform the resultant abstract specifications to a final specification form. This final specification is an *analysis* of the original program - what it specifies is a (weaker) true assertion about the original specification.

Intermediate transformations of an original source program are considered to be models of the program at different levels of abstraction, and the goal of the transformation phase of the analysis process is to produce a functional abstract specification of the program. This specification is then executed to test properties of the program.

In the following section of the paper, sample rules for equivalence and abstraction are given, and then an example analysis uses these two sorts of rules to transform a small program. The resultant functional specification reveals an unexpected property of the code.

4.1 Equivalence Rules

In the equivalence rules, an attempt is made to capture all relevant behaviour of the program. The refinement calculus textbooks give detailed descriptions of rules suitable for program development which are not repeated here, however, here are two sample rules with specific application in the area of analysis of assignment statements.

1. An equivalence rule for the assignment $v := e$ using strongest postconditions:

$$\equiv \quad \text{Equivalent assignment postcondition introduction}$$
$$v\text{:}\left[P\ ,\ \exists\, v' : P[v'/v] \wedge v = e[v'/v]\right]$$

2. An equivalence rule for the assignment $v := e$ using weakest preconditions:

$$\equiv \text{ Equivalent assignment precondition introduction}$$
$$v\text{:}\left[\, P[e/v] \;,\; P \,\right]$$

There are other rules for loops, if statements and so on.

4.2 Abstraction Rules

As discussed in Section 2, the program state is represented here as a set of linear inequalities, and the transforming operations on these are standard linear programming ones such as *convex hull* or *projection* operations. The resultant expressions are computationally easy to evaluate, but may tell us *less* about our programs.

1. A rule for abstract interpretation style analysis of assignments of the form $v := e$ that are not invertible, such as $v := 0$:

   $$\sqsupseteq \text{ Abstract ni-assignment postcondition introduction}$$
   $$v\text{:}\left[\, P \;,\; \text{proj}_{v^\perp}(P) \wedge v = e \,\right]$$

   In this expression, the notation $\text{proj}_{v^\perp}(P)$ represents the projection of the expression P onto the orthogonal complement of v.
2. A rule for assignments which are invertible, such as $v := v + 1$. We may generalize such assignments as $v := f(v)$, where f is an invertible function:

   $$\equiv \text{ Abstract i-assignment postcondition introduction}$$
   $$v\text{:}\left[\, P \;,\; P[f^{-1}(v)/v] \,\right]$$

   In this expression, the notation f^{-1} represents the inverse of f. This rule may only be applied if $\exists f^{-1}\ \forall x :\ f(f^{-1}(x)) = x$.
3. A rule for projecting a specification $f\text{:}\left[\,\mathsf{true} \;,\; Q\,\right]$ onto the orthogonal complement of (say) x. This sort of abstraction is used to perform analysis on a subset of the variables in a frame, while retaining as much information as possible about the frame:

   $$\sqsupseteq \text{ Abstract projection onto } x^\perp$$
   $$g\text{:}\left[\, \mathsf{true} \;,\; \text{proj}_{x^\perp}(Q) \,\right]$$

   In this expression g is defined by $g \cup x = f \wedge g \cap x = \{\}$ (i.e. the frame f without the variable x).

4. A rule for the *do-loop* $\mathbf{do}\ B \rightarrow f\colon[P\ ,\ Q]\ \mathbf{od}$ introduces the $\text{conv}(R)$ operator which returns the *convex hull* of R:

   $\sqsupseteq$ Abstract iteration postcondition introduction

   $$f\colon[R\ ,\ R];$$
   $$\mathbf{do}\ f\colon[\text{conv}(R \cup Q) \wedge B\ ,\ Q]\ \mathbf{od};$$
   $$f\colon[\text{conv}(R \cup Q) \wedge \neg B\ ,\ \text{conv}(R \cup Q) \wedge \neg B]$$

   The $f\colon[R\ ,\ R]$ component of the refinement is an artifact to introduce a state variable name. The last component of the refinement retains information about the *do-loop*.

Note that we cannot mix refinement and abstraction rules and expect the resultant expression to still be a refinement of the original expression.

4.3 Example Analysis

In this section, a small example is analysed, demonstrating the two phases of analysis used in this approach. In the first *transformation* phase, a code implementation is given, and then transformed according to abstract analysis rules. In the second *execution* phase, the resultant specification is used to derive something possibly *bad* about the particular implementation - specifically that the result might be wrong in some circumstances.

In public key encryption schemes, large integer computations often have to be performed. For example, the evaluation of $\text{modulo}(P^Q, N)$ where P, Q and N are all large numbers. A simple implementation might involve calculating first P^Q, and then performing a **mod()** (*modulo*) machine operation[3]. However, the calculation of P^Q may involve very large numbers, difficult to manipulate on a computer. The following code is another implementation of this specification, and calculates $\text{modulo}(P^Q, N)$, leaving the result in variable c. A quality of this particular implementation is that the code never has to calculate P^Q - the largest calculation is always less than $N * P$:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ x, d : \mathbb{N} \bullet \\
\quad c, x, d := 1, 0, 0; \\
\quad \mathbf{do}\ x \neq Q \rightarrow x, d := x + 1, c * P; \\
\quad\quad\quad\quad c := \text{mod}(d, N) \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

[3] Note that the mathematical expression $\text{modulo}(x, y) = x$ if $y = 0$. This is different from the standard programmer's experience with the **mod()** operation which is that **mod(x,y)** is always less than y.

Applying abstraction rules to the assignments inside the do-loop results in this:

$\sqsupseteq$ Abstract assignment postcondition introduction

$$
\begin{array}{l}
\textbf{var } x, d : \mathbb{N} \bullet \\
\quad c, x, d := 1, 0, 0; \\
\quad \textbf{do } x \neq Q \rightarrow x, d, c\colon \left[M \ , \ \mathcal{L} \right]; \\
\qquad\qquad\qquad x, d, c\colon \left[\mathcal{L} \ , \ \mathrm{proj}_{c^\perp}(\mathcal{L}) \wedge c < N \right] \\
\quad \textbf{od}
\end{array}
$$

where $\mathcal{L}$ is $\mathrm{proj}_{d^\perp}(M[x-1/x]) \wedge d = c * P$. Note also that the specification state variable M can stand for anything, awaiting later unification with some concrete state. The first assignment in the *do-loop* requires a mix of both invertible and non-invertible assignment rules. The second assignment uses the **mod()** operator, and information is lost here, as we only represent state using linear inequalities. As a result, the only retained effect of $c := \mathrm{mod}(d, N)$ is that $c < N$. Application of the do-loop abstraction rule leads to:

$\sqsupseteq$ Abstract iteration postcondition introduction

$$
\begin{array}{l}
\textbf{var } x, d : \mathbb{N} \bullet \\
\quad c, x, d := 1, 0, 0; \\
\quad x, d, c\colon \left[R \ , \ R \right]; \\
\quad \textbf{do } x, d, c\colon \left[\mathcal{H} \wedge x \neq Q \ , \ \mathcal{K} \right]; \\
\qquad x, d, c\colon \left[\mathcal{K} \ , \ \mathrm{proj}_{c^\perp}(\mathcal{K}) \wedge c < N \right] \\
\quad \textbf{od}; \\
\quad x, d, c\colon \left[\mathcal{H} \wedge x = Q \ , \ \mathcal{H} \wedge x = Q \right]
\end{array}
$$

Where $\mathcal{H}$ is shorthand for $\mathrm{conv}(R \cup \mathrm{proj}_{c^\perp}(d = c * P) \wedge c < N)$, and $\mathcal{K}$ is shorthand for $\mathcal{H} \wedge x - 1 \neq Q \wedge d = c * P$. After further simplification and abstractions including the projection onto $x^\perp d^\perp$, many of the terms disappear, resulting in this derivation of the original code:

$\equiv$ Simplification

$$
\begin{array}{l}
\textbf{var } x, d : \mathbb{N} \bullet \\
\quad x, d, c\colon \left[R \ , \ \mathcal{S} \wedge c = 1 \right]; \\
\quad \textbf{do } x, d, c\colon \left[\mathcal{T} \ , \ \mathcal{T} \right]; \\
\qquad x, d, c\colon \left[\mathcal{T} \ , \ \mathcal{T} \wedge c < N \right] \\
\quad \textbf{od}; \\
\quad x, d, c\colon \left[\mathcal{T} \ , \ \mathcal{T} \right]
\end{array}
$$

Where $\mathcal{T}$ is shorthand for $\mathrm{conv}((\mathcal{S} \wedge c = 1) \cup c < N)$ and $\mathcal{S}$ is shorthand for $\mathrm{proj}_{c^\perp}(R)$. In the prototype software, the order of application of rules can be

modified by the user of the system, but the transformations are done automatically. If we now only consider the first and last conditions, a derived specification of the whole code segment is:

$\equiv$ Simplification
$x, d, c\!: \left[R\ ,\ \mathcal{T} \right]$

The view here is that an analysis *model* has been produced by the transformation rules. This model specifies true properties of the original program. In the practical application of the logic, input programs are encoded according to the transformation rules into linked **HOL/ML** functions representing the relationship between pre and postconditions of the derived specification. The functions then comprise an engine for abstract modeling of the behaviour of the program. This completes the transformation phase of this example.
In the execution phase, the model is executed to test the behaviour of the program in the specified abstract domain. For example, if R is unified with $N \leq 1$, the analysis reduces to:

$\sqsupseteq$ Unification
$x, d, c\!: \left[N \leq 1\ ,\ c \leq N + 1 \right]$

This reveals a property of the implementation that was not apparent before, specifically that if $N \leq 1$, then c has a possibly incorrect value - it should always be less than N. The analysis process has pinpointed a problem with our code[4]. If R is unified with $N > 1$, then we verify that c will *always* be less than N:

$\sqsupseteq$ Unification
$x, d, c\!: \left[N > 1\ ,\ c < N \right]$

At this stage a choice may be made to either accept the behaviour of the code or change/correct it.

Note that the end result of the transformation phase of analysis is a model of the functional behaviour of the original program with respect to the particular abstraction used. In this case, the relationship between c and N was of particular interest, and the penultimate analysis *model* was able to confirm that our desired property ($c < N$) was guaranteed for $N > 1$.

5 On Using HOL

HOL is used in this development in two ways. Firstly as an expressive language in which to encode and simplify the transformations, and secondly to prove assertions made about pre or postconditions. The coding of the logic in **HOL/ML** is straightforward, often reducing to a simple translation from the mathematical representation of the element to a **HOL/ML** function. Some representative **HOL/ML** transforms are given in the next section to demonstrate the approach.

[4] If $N = 0$ and $Q = 0$, then the code returns $c = 1$, which is correct according to the mathematical definition of modulo($P^0, 0$), but is counter to an (unstated) assumption about the program that the resultant values will always be less than N.

5.1 Transforms in HOL

In the **HOL/ML** transform functions for an analysis tool, assertions about program state are manipulated as **HOL** *terms*. As an example of the techniques for constructing transform functions in **HOL/ML**, here are implementations of some of the transforms:

1. The first equivalence rule given for the assignment $v := e$ was:

   $\equiv$ Equivalent assignment postcondition introduction
   $v\textbf{:}\left[\, P \;,\; \exists\, v' : P[v'/v] \wedge v = e[v'/v] \,\right]$

   This may be interpreted as a transforming function which translates a precondition P to some postcondition. **HOL** has an embedded parser which can express this for us succinctly, and the following **HOL/ML** code is used for processing assignments of this form. The code defines a function with three parameters (P, e and v), and returns the required postcondition:

```
fun FpAssign (P:Term.term) (e:Term.term) (v:Term.term) =
   --`?v0. ((\(^v). ^P)v0) /\ (^v=(\(^v). ^e)v0)`--;
```

2. The second equivalence rule for the assignment $v := e$ was:

   $\equiv$ Equivalence assignment precondition introduction
   $v\textbf{:}\left[\, P[e/v] \;,\; P \,\right]$

   This may be interpreted as a transforming function which translates a postcondition P to some precondition. The **HOL/ML** implementation is:

```
fun RpAssign (P:Term.term) (e:Term.term) (v:Term.term) =
   --`(\(^v). ^P)^e`--;
```

In the chosen abstraction scheme, assertion state is represented by a linear set of inequalities. **HOL** has no native linear programming theory, but external libraries may be used, while still retaining high assurance that only **true** theorems may be proved. In this work, functions translate **HOL** terms to and from a structure representing a set of linear inequalities. Following this, various **LP**-based functions may be used to calculate the convex-hull or projection operations.

When using abstract interpretation analysis, the transform for assignment is optimized for various different types of expression. For example, an assignment like $x := x + 1$ is invertible, and involves no loss of state information, whereas an assignment like $x := a + b$ results in the loss of any relationships dependent on x', the previous value of x. Since our assertion state is represented by a linear set of inequalities we may remove the variable x' using projection. The code for projection is implemented separately from the **HOL** theory definitions, and may be subject to (programmer) error or inconsistency. For this reason, these external functions are called from **HOL**, and then tested afterwards for correctness within the **HOL** theory.

3. The first abstraction rule for non-invertible assignments was:

$\sqsupseteq$ Abstract ni-assignment postcondition introduction
$v\colon \left[P \;,\; \mathrm{proj}_{v^{\perp}}(P) \wedge v = e \right]$

The following **HOL/ML** code is used for processing assignments of this form:

```
fun FpAbsNI (P:Term.term) (e:Term.term) (v:Term.term) =
   --`proj(^v,^P) /\ (^v=^e)`--;
```

5.2 Proof in HOL

The emphasis in the previous section was in the use of **HOL/ML** as an expressive and efficient language for encoding the analysis model. However this is only part of the usefulness of **HOL** in this application. During the process of analysis, it may be useful to prove programmer-supplied assertions about the program.

For example, in program code for sorting an array, an assertion about a partition of an array may be used to confirm that the sort program is working correctly. For example - we might know that the array is divided into a sorted *left-part* from $A[0]$ to $A[P-1]$ and a semi-sorted *right-part* in which the leftmost element $A[P]$ is the least element of the *right-part*, and that all elements in the *left-part* are less than or equal to all elements in the *right-part*. Given this, **HOL** may be used to prove that the array is now sorted from $A[0]$ to $A[P]$. The assertion to be proved is quite complex:

$$\begin{aligned}\vdash\ & \mathit{psorted}\, A[0..P-1] \\ & \quad \wedge \mathit{pminindex}\, A[P..N-1]\, P \\ & \quad \wedge (\forall\, x\, y.\, x \in \{0..P-1\} \wedge y \in \{P..N-1\} \Rightarrow A[x] \leq A[y]) \\ & \qquad \Rightarrow \mathit{psorted}\, A[0..P]\end{aligned}$$

A **HOL** proof script for this is as follows:

```
val assertionB = prove
((--`(psorted A (0..P-1)
     /\ pminindex A (P..N-1) P
     /\ (!x y. (0..P-1)x /\ (P..N-1)y ==> A[x]<=A[y]))
     ==>psorted A (0..P)`--),
   ARW_TAC[index_min_partition_DEF,inrange_def,sorted_DEF]
   THEN (`A[P-1]<=A[P]` by ZAP_TAC(arith_ss)[])
   THEN Cases_on `j<P`
   THEN REPEAT (ZAP_TAC(arith_ss)[]));
```

A more complete explanation of this process of proof of assertions, and a **HOL** theory-of-arrays is found in [1].

6 Conclusion

This work is a part of a larger body of research into program analysis derived from Heintze, Jaffar and Voicu's [7] Conditional Hoare Logic reasoning framework. Various approaches to the management of a process of program analysis are being explored, and this paper reports on the notation and techniques used for representing abstract interpretation within the framework.

The end result of the analysis process is a model of the functional behaviour of the original program with respect to the particular abstraction used. This analysis *model* is used to confirm specific properties of the code under investigation through a testing process. In addition, the models may be directly manipulated in a program *proof* context, providing confirmation of user-supplied assertions.

The process takes place within a formal logic for analysis modeled on the refinement calculus, and the particular notation and methodology is of particular use when combined with other program analysis systems.

Acknowledgments

Thanks to Gill Dobbie, Dong Jin Song and the anonymous referees for their helpful and insightful comments on this paper.

References

1. H. Anderson. Partition theory for CHL. Internal report found at `http://www.comp.nus.edu.sg/~hugh/chlproject/TheoryOfArrays.pdf`, 2001.
2. R-J. Back and J. von Wright. *Refinement Calculus A Systematic Introduction*. Springer, 1998.
3. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Association for Computer Machinery, SIGACT/SIGPLAN Symp on Principles of Programming Languages (POPL)*, pages 84–97, Jan 1978.
4. K. Fukuda and A. Prodon. Double description method revisited. In M. Deza, R. Euler, and I. Manoussakis, editors, *Combinatorics and Computer Science*, volume 1120, pages 91–111. Springer-Verlag, 1996.
5. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
6. Michael J. C. Gordon, R. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: a mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1979.
7. N. Heintze, J. Jaffar, and R. Voicu. A framework for analysis and verification. In *Association for Computer Machinery, SIGACT/SIGPLAN Symp on Principles of Programming Languages (POPL)*, pages 26–39, Jan 2000.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
9. C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 1994.
10. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.