Abstract. Abstract interpreters are tools to compute approximations for behaviors of a program. These approximations can then be used for optimisation or for error detection. In this paper, we show how to describe an abstract interpreter using the type-theory based theorem prover Coq, using inductive types for syntax and structural recursive programming for the abstract interpreter’s kernel. The abstract interpreter can then be proved correct with respect to a Hoare logic for the programming language.

1 Introduction

Higher-order logic theorem provers provide a description language that is powerful enough to describe programming languages. Inductive types can be used to describe the language’s main data structure (the syntax) and recursive functions can be used to describe the behavior of instructions (the semantics). Recursive functions can also be used to describe tools to analyse or modify programs. In this paper, we will describe such a collection of recursive function to analyse programs, based on abstract interpretation [7].

1.1 An Example of Abstract Interpretation

We consider a small programming language with loop statements and assignments. Loops are written with the keywords while, do and done, assignments are written with :=, and several instructions can be grouped together, separating them with a semi-column. The instructions grouped using a semi-column are supposed to be executed in the same order as they are written. Comments are written after two slashes //.

We consider the following simple program:

\[
\begin{align*}
  x &:= 0; & \text{ // line 1} \\
  \text{While } x < 10 & \text{ do } & \text{ // line 2} \\
  & x := x + 1 & \text{ // line 3} \\
  \text{done} & \text{ // line 4}
\end{align*}
\]

We want to design a tool that is able to gather information about the value of the variable \( x \) at each position in the program. For instance here, we know that

\[ x \text{ is a natural number.} \]
after executing the first line, \( x \) is always in the interval \([0,0]\); we know that before executing the assignment on the third line, \( x \) is always smaller than 10 (because the test \( x < 10 \) was just satisfied). With a little thinking, we can also guess that \( x \) increases as the loop executes, so that we can infer that before the third line, \( x \) is always in the interval \([0,9]\). On the other hand, after the third line, \( x \) is always in the interval \([1, 10]\). Now, if execution exits the loop, we can also infer that the test \( x < 10 \) failed, so that we know that \( x \) is larger than or equal to 10, but since it was at best in \([0,10]\) before the test, we can guess that \( x \) is exactly 10 after executing the program. So we can write the following new program, where the only difference is the information added in the comments:

```plaintext
// Nothing is known about x on this line
x := 0; // 0 <= x <= 0
while x < 10 do
    // 0 <= x <= 9
    x := x + 1 // 1 <= x <= 10
done
    // 10 <= x <= 10
```

We want to produce a tool that performs this analysis and produces the same kind of information for each line in the program. Our tool will do slightly more: first it will also be able to take as input extra information about variables before entering the program, second it will produce information about variables after executing the program, third it will associate an invariant property to all while loops in the program. Such an invariant is a property that is true before and after all executions of the loop body (in our example the loop body is \( x := x+1 \)). A fourth feature of our tool is that it will be able to detect occasions when we can be sure that some code is never executed. In this case, it will mark the program points that are never reached with a `false` statement meaning “when this point of the program is reached, the `false` statement can be proved (in other words, this cannot happen)”.

Our tool will also be designed in such a way that it is guaranteed to terminate in reasonable time. Such a tool is called a static analysis tool, because the extra information can be obtained without running the program: in this example, executing the program requires at least a thousand operations, but our reasoning effort takes less than ten steps.

Tools of this kind are useful, for example to avoid bugs in programs or as part of efficient compilation techniques. For instance, the first mail-spread virus exploited a programming error known as a buffer overflow (an array update was operating outside the memory allocated for that array), but buffer overflows can be detected if we know over which interval each variable is likely to range.

### 1.2 Formal Description and Proofs

Users should be able to trust the information added in programs by the analysers. Program analysers are themselves programs and we can reason about their correctness. The program analysers we study in this paper are based on abstract