Abstract. The abstract interpretation framework based upon the approximation of a fixpoint collecting semantics using Galois connections and widening/narrowing operators on complete lattices [CC77a, CC79b] has been considered difficult to apply to Mycroft's strictness analysis [Myc80, Myc81] for which denotational semantics was though to be more adequate (because non-termination has to be taken into account), see e.g. [AH87], page 25.

Considering a non-deterministic first-order language, we show, contrary to expectation, and using the classical Galois connection-based framework, that Mycroft strictness analysis algorithm is the abstract interpretation of a relational semantics (a big-steps operational semantics including non-termination which can be defined in G\textsuperscript{=\text{=\textsuperscript{\text{SOS}}} either in rule-based or fixpoint style by induction on the syntax of programs [CC92])

An improved version of Johnsson's algorithm [Joh81] is obtained by a subsequent dependence-free abstraction of Mycroft's dependence-sensitive method.

Finally, a compromise between the precision of dependence-sensitive algorithms and the efficiency of dependence-free algorithms is suggested using widening operators.

Keywords: Abstract interpretation; Relational semantics; Strictness analysis; Galois connection; Dependence-free and dependence-sensitive analysis; Widening.

Abstract interpretation [CC77a, CC79b] is a method for designing approximate semantics of programs which can be used to gather information about programs in order to provide safe answers to questions about their run-time behaviours. These semantics can be used to design manual proof methods or to specify automatic program analyzers. When the semantic analysis of programs is to be automated, the answers can only be partial or approximate (that is correct/safe/sound but incomplete) since questions such as termination for all input data are undecidable.

By considering that non-terminating and erroneous behaviors are equivalent, call-by-need can be replaced by call-by-value in functional programs either whenever the actual argument is always evaluated at least once in the function body or upon the later recursive calls (so that if the evaluation of the actual argument does not terminate or is erroneous then so does the function call) or whenever the function call does
not terminate or is erroneous (whether the actual argument is evaluated or not). Alan Mycroft's strictness analysis [Myc80, Myc81] is an abstract interpretation designed to recognize these situations. Observe the importance of taking non-termination \( \bot \) into account: left-to-right addition is strict in its second argument since it does not terminate either when evaluation of its first argument is guaranteed not to terminate (in which case the second argument is not needed) or the evaluation of its first argument does terminate but that of the second does not. In the classical definition of strictness, errors \( \Omega \) must also be identified with non-termination \( \bot \). Otherwise left-to-right addition would not be strict in its second argument since \( \text{true} + \bot = (1/0) + \bot = \Omega \).

Given a recursive function declaration \( f(x, y) \equiv e \), Alan Mycroft has designed an abstract function \( f^I(x, y) \equiv e^I \) on the abstract domain \( \{0, 1\} \) such that:

- if \( f^I(0, b) = 0 \) for \( b \in \{0, 1\} \) then \( f \) is strict in \( x \);
- if \( f^I(a, 0) = 0 \) for \( a \in \{0, 1\} \) then \( f \) is strict in \( y \);
- if \( f^I(1, 1) = 0 \) then \( f \) never terminates;
- if \( f^I(a, b) = 1 \) for \( a, b \in \{0, 1\} \) then the abstraction is too approximate and no conclusion can be drawn.

The abstract value \( e^I \) of an expression \( e \) is determined as follows:

- If the expression \( e \) is reduced to a constant \( k \) then its abstract value \( e^I \) is 1 which is the best possible approximation to the fact that it is an integer.
- If the expression \( e \) is reduced to a formal parameter \( x \) then its evaluation is erroneous or does not terminate if and only if the evaluation of the actual argument is erroneous or does not terminate so that its abstract value \( e^I \) is \( x \) which denotes the value of the abstract actual argument.
- If the expression \( e \) is a basic operation such as \( e_1 + e_2 \) (always needing its two arguments \( e_1 \) and \( e_2 \)) and the abstract values \( e_1^I \) and \( e_2^I \) of expressions \( e_1 \) and \( e_2 \) are obtained recursively, then \( (e_1 + e_2)^I = (e_1^I \land e_2^I) \) where \( (0 \land 0) = (0 \land 1) = (1 \land 0) = 0 \) and \( (1 \land 1) = 1 \) since the evaluation of \( e_1 + e_2 \) is erroneous or does not terminate when that of \( e_1 \) or that of \( e_2 \) is erroneous or does not terminate, which we conclude from \( e_1^I = 0 \) or \( e_2^I = 0 \).
- The evaluation of a conditional \( e \equiv (e_1 \rightarrow e_2, e_3) \) is erroneous or does not terminate when the evaluation of the condition \( e_1 \) is erroneous (for example non-boolean) or does not terminate so that \( e_1^I = 1 \), then erroneous termination or non-termination of \( e \) is guaranteed only if those of both \( e_2 \) and \( e_3 \) are guaranteed so that \( e^I = 0 \) only if \( e_2^I = 0 \) and \( e_3^I = 0 \). Therefore \( e^I \) can be defined as \( (e_1^I \land (e_2^I \lor e_3^I)) \) where \( (0 \lor 0) = 0 \) and \( (0 \lor 1) = (1 \lor 0) = (1 \lor 1) = 1 \).

For the traditional addition example:

\[
 f(x, y) \equiv ((x = 0) \rightarrow y, (1 + f(x - 1, y)))
\]

we have:

\[
 f^I(x, y) = (x \land 1) \land (y \lor (1 \land f^I(x \land 1, y)))
\]

After simplifications (such as \( x \land 1 = (1 \land x) = x \), \( (x \land 0) = (0 \land x) = 0 \), etc.), we get:

\[
 f^I(x, y) = x \land (y \lor f^I(x, y))
\]