Heuristics for Safety and Security Constraints

Flemming Nielson\textsuperscript{1,2}

Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Kongens Lyngby, Denmark

Hanne Riis Nielson\textsuperscript{3}

Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Kongens Lyngby, Denmark

Abstract

The flow logic approach to static analysis amounts to specifying the admissibility of solutions to analysis problems; when specified using formulae in stratified alternation-free least fixed point logic one may use efficient algorithms for computing the least admissible solutions. We extend this scenario to validate the fulfillment of safety and security constraints on admissible solutions; the modified development produces a least solution together with a boolean value indicating whether or not the constraints are validated or violated. The main contribution is the development of a deterministic heuristics for obtaining a solution that is close to the least solution while enforcing the safety or security constraints. We illustrate it on the Bell-LaPadula mandatory access control policy where the heuristics is used to suggest modifications to the security annotations of entities in order for the security policy to hold.

Keywords: Static Analysis, Flow Logic, Least Fixed Point logic, Security Properties, Bell-LaPadula Mandatory Access Control.

1 Introduction

The goals of the paper are perhaps best explained by means of an analogy. In the world of type systems one frequently distinguishes between soft typing and strong typing. In \textit{soft typing} all programs can be typed (possibly with an all encompassing top type) and the goal is to use types to provide as much meaningful information about subprograms as possible. In \textit{strong typing} the whole point of the type system is to reject certain programs as being ill-formed (including those that might lead to...
certain kinds of errors when executed) whereas providing meaningful information about subprograms is an important secondary aim. Indeed, the slogan of strong typing is that “well typed programs cannot go wrong” [9]. We might say that soft typing focuses on solving a type inference problem whereas strong typing focuses on enforcing the solvability of a type inference problem (which admittedly involves a solving phase as well) subject to additional constraints.

In this paper we consider the world of static analysis as embodied in data flow and control flow analysis. Here the view traditionally is that of solving an analysis problem in order to provide information that may be useful, e.g. in case of a compiler generating better than naive code. When viewed in the general framework of abstract interpretation [6,10] one usually establishes a Moore Family result showing that a least solution always exists. Given a problem \( \text{cls} \) we shall write

\[ S(\text{cls}) = \rho \]

for the least solution \( \rho \). In Section 2 we slightly extend our approach [12,11] based on formulae in stratified alternation-free least fixed point logic.

Our first contribution, in Section 3, is to extend the development so as to be more directly applicable to software validation, e.g. for enforcing safety and security policies. Quite frequently it is possible to formulate such policies as sets of safety and security constraints upon the solution \( S(\text{cls}) \). Given a problem \( \text{cls} \) with such additional constraints we shall write

\[ V(\text{cls}) = (\rho, b) \]

for the least solution \( \rho \), as computed by \( S(\text{cls}) \), together with a boolean value \( b \) indicating the truth value of the constraints. While it is possible to obtain this effect already in the setting of Section 2, by means of observation predicates as discussed towards the end of Section 3, the approach taken in Section 3 is to enlarge the formalism with more explicit notation for the safety and security constraints. This allows to write more readable specifications, that clearly separate between which constraints are to be interpreted in a soft versus strong manner. We illustrate the development by showing how to enforce the condition, that programs in a functional language never attempt to perform a function call unless the value applied is indeed a function.

This paves the way for our main contribution, in Section 4, where we consider how to deal with a problem \( \text{cls} \) that cannot be validated, i.e. a problem \( \text{cls} \) for which \( V(\text{cls}) = (\rho, \text{false}) \). Here our goal is to develop a deterministic heuristics for finding a small modification \( \varrho \) to the solution \( \rho \) such that the problem can be validated under the assumption that the behaviour of \( \varrho \) can be admitted. This idea has in part been inspired by the non-standard approach to fixpoints explored in [8] and the iterative computation of solutions in [4]; it amounts to an iterative approach to recalculating solutions, but unlike the rather ad-hoc presentation in [4], it allows to state the desired solution within the logical framework. The desired result of our
heuristics therefore is
\[ \mathcal{H}(\text{cls}) = (\rho, \varrho) \]
where \( \varrho \) is the small modification deemed necessary and \( \rho \) is the resulting least solution for which the safety and security constraints can be enforced; we may write this as \( \mathcal{V}(\text{cls} @ \varrho) = (\rho, \text{true}) \) where \( \text{cls} @ \varrho \) is a syntactic mechanism used to enforce that \( \rho \) includes the contribution demanded by \( \varrho \).

The main motivating example for this development is from the world of mandatory access control policies. We present an example showing how to formulate the Bell-LaPadula mandatory access control policy \([2,7]\) for enforcing confidentiality. In this policy, both programs (called subjects) and files (called objects) are given a security classification: high (meaning secret) or low (meaning public); this may result in classifications that actually violate the policy. We then use the development of Section 4 to obtain an indication, in the form of \( \varrho \), of which entities that need to be reclassified in order to satisfy the policy and stay as close as possible to the original intentions.

2 Flow Logic using ALFP

Flow Logic \([12]\) is a specification oriented approach to static analysis of programs. For a program \( P \) of interest the focus is on specifying when an analysis estimate \( A \) correctly describes the behaviour of \( P \) during evaluation. This takes the form of a judgement
\[ A \vdash P \]
that yields true whenever this is the case. Usually \( A \) is an element of a complete lattice whose greatest element \( \top \) satisfies the specification; hence the focus at this stage merely is to exclude analysis estimates that are observably incorrect and there is no attempt to demand that the analysis estimate is the best or least choice possible. The definition of \( A \vdash P \) often takes the form of recursive definitions (as in Figures 1, 2 and 3 to be developed subsequently) and in general a co-inductive definition interpretation is desired; when there is only limited use of higher-order features (as in the examples in the present paper) the co-inductive interpretation coincides with the more usual inductive interpretation.

The formal statement of correctness of the specification is often expressed with respect to an operational semantics of programs, \( P \rightarrow P' \) and takes the form
\[ A \vdash P \land P \rightarrow P' \Rightarrow A \vdash P' \]
for ensuring that the analysis estimate correctly describes all program derivations. The proof is usually fairly straightforward by induction on the structure of \( P \rightarrow P' \).

Having validated the correctness of the specification the next concern is to ensure that the specification admits more usable analysis estimates than \( \top \). Following the overall approach of Abstract Interpretation \([6]\) this takes the form of showing that the set of admissible solutions constitutes a Moore family:
\[ Y \subseteq \{ A \mid A \vdash P \} \Rightarrow \cap Y \subseteq \{ A \mid A \vdash P \} \]
(This corresponds to what is sometimes called the model intersection property of logics and is related to the existence of principal types in type systems [10].) It follows that there exists a least solution

$$A_0 = \cap \{ A \mid A \models P \}$$

and that it satisfies the specification, i.e. $A_0 \models P$.

Knowing that there exists a best analysis estimate our final concern will be how to compute it in a practical manner, e.g. in polynomial time in the size of the program $P$. One approach is to generate a set of constraints or (as we shall do in the present paper) a suitable logical formula for the program $P$. This is particularly direct if the specification of $A \models P$ has already been performed using a suitable logic and if the specification coincides with its inductive interpretation. In this case it is usually fairly straightforward to expand the defining clauses of $A \models P$ into an equivalent logical formula $cls$.

**Alternating Least Fixpoint Logic**

We now review a suitable logic that has been used in a number of static analyses: The Alternation-free fragment of Least Fixpoint Logic (ALFP) extends Horn clauses by allowing both existential and universal quantifications in preconditions, negative queries (subject to the notion of stratification), disjunctions of preconditions, and conjunctions of conclusions.

**Definition 2.1** Given a fixed countable set $\mathcal{X}$ of variables and a finite alphabet $\mathcal{R}$ of predicate symbols we define the set of ALFP formulae (or clause sequences), $cls$, together with clauses, $cl$, and preconditions, $pre$, by the grammar

$$\begin{align*}
pre & ::= R(x_1, \cdots, x_n) \mid \neg R(x_1, \cdots, x_n) \\
& \quad \mid pre_1 \land pre_2 \mid pre_1 \lor pre_2 \mid \exists x : pre \mid \forall x : pre \\
cl & ::= R(x_1, \cdots, x_n) \mid 1 \mid cl_1 \land cl_2 \mid \forall x : cl \mid pre \Rightarrow cl \\
cls & ::= cl_1, \cdots, cl_k
\end{align*}$$

where $x \in \mathcal{X}$, $R \in \mathcal{R}$ and $k$ is at least 1.

Occurrences of $R$ and $\neg R$ in preconditions are called *queries* and *negative queries*, respectively, whereas the other occurrences of $R$ are called *assertions* of the predicate $R$. We write $1$ for the always true clause.

**Stratification.** In order to ensure desirable theoretical and pragmatic properties in the presence of negation, we introduce a notion of stratification similar to the one in *Datalog* [5,1]. Intuitively, stratification ensures that a negative query is not performed until the predicate queried has been fully asserted. This is important for ensuring that once a precondition evaluates to *true* it will continue doing so even after further assertions of predicates.
**Definition 2.2** A formula $cls$ is stratified w.r.t. $rank$ whenever it has the form $cls = cl_1, \ldots, cl_k$, and the function $rank : R \to \{0, \ldots, k\}$ satisfies the following properties for all $i = 1, \ldots, k$:

(i) $rank(R) \geq i$ for every assertion $R$ in $cl_i$;
(ii) $rank(R) \leq i$ for every positive query $R$ in $cl_i$; and
(iii) $rank(R) < i$ for every negative query $\neg R$ in $cl_i$.

A formula $cls$ is stratified if there exists a ranking function $rank$ such that $cls$ is stratified w.r.t. $rank$.

Not all formulae are stratified and a formula may be stratified w.r.t. some ranking functions but not stratified w.r.t other ranking functions. Given a formula $cls = cl_1, \ldots, cl_k$ one can construct an optimal ranking function $rank$, i.e. one that makes $cls$ stratified w.r.t. $rank$ if and only if $cls$ is stratified, by setting $rank(R) = k$ if there are no (positive or negative) queries to $R$ in $cls$, otherwise setting $rank(R) = 0$ if there are no assertions to $R$ in $cls$, and setting $rank(R) = i$ if $cl_i$ is the rightmost clause containing an assertion to $R$.

**Stratifiability.** Sometimes a clause does not have the form of a formula that is stratified w.r.t. some ranking function $rank$ although it can easily be rearranged into such a formula. This is possible if for each subclause $\cdots R \cdots \Rightarrow \cdots S \cdots$ we have that $rank(R) \leq rank(S)$ and furthermore, if for each subclause $\cdots \neg R \cdots \Rightarrow \cdots S \cdots$ we have that $rank(R) < rank(S)$. We shall say that a clause $cl$ is stratifiable w.r.t $rank$ whenever these conditions are met. One approach to obtaining a stratified formula $cls$ from the clause $cl$ is simply to construct it as $cls = cl_1, \ldots, cl_k$ where each $cl_i$ is obtained from $cl$ by replacing assertions of rank different from $i$ with the clause $1$.

In a similar vein a clause $cl$ is stratifiable if it is possible to construct a ranking function $rank$ such that the clause is stratifiable w.r.t. $rank$. An easy test for this condition is to build a graph with predicate symbols as nodes (called $R$ and $S$ below) and two kinds of edges; there is a normal edge from $R$ to $S$ if the clause contains a subclause $\cdots R \cdots \Rightarrow \cdots S \cdots$ and there is a fat edge from $R$ to $S$ if the clause contains a subclause $\cdots \neg R \cdots \Rightarrow \cdots S \cdots$. Then the clause is stratifiable if and only if there is no loop containing a fat edge.

Stratifiable clauses are accepted by a preprocessor to the Succinct Solver [11] and are turned into appropriately stratified formulae which are then solved.

**Constraint Satisfaction**

We take a pure approach where the logic is interpreted over a universe $U$ of constants. Given interpretations $\rho$ and $\sigma$ for predicate symbols and variables, respectively, we define in Table 1 the following satisfaction relations:

$(\rho, \sigma) \models pre$ for preconditions $pre$,
$(\rho, \sigma) \models cl$ for clauses $cl$, and
$(\rho, \sigma) \models cls$ for formulas $cls$. 
\[(\rho, \sigma) \models R(x_1, \cdots, x_n) \iff (\sigma(x_1), \cdots, \sigma(x_n)) \in \rho(R)\]
\[(\rho, \sigma) \models \neg R(x_1, \cdots, x_n) \iff (\sigma(x_1), \cdots, \sigma(x_n)) \notin \rho(R)\]
\[(\rho, \sigma) \models pre_1 \land pre_2 \iff (\rho, \sigma) \models pre_1 \text{ and } (\rho, \sigma) \models pre_2\]
\[(\rho, \sigma) \models pre_1 \lor pre_2 \iff (\rho, \sigma) \models pre_1 \text{ or } (\rho, \sigma) \models pre_2\]
\[(\rho, \sigma) \models \exists x : pre \iff (\rho, \sigma[x \mapsto a]) \models pre \text{ for some } a \in U\]
\[(\rho, \sigma) \models \forall x : pre \iff (\rho, \sigma[x \mapsto a]) \models pre \text{ for all } a \in U\]
\[(\rho, \sigma) \models R(x_1, \cdots, x_n) \iff (\sigma(x_1), \cdots, \sigma(x_n)) \in \rho(R)\]
\[(\rho, \sigma) \models 1 \iff \text{true}\]
\[(\rho, \sigma) \models cl_1 \land cl_2 \iff (\rho, \sigma) \models cl_1 \text{ and } (\rho, \sigma) \models cl_2\]
\[(\rho, \sigma) \models \forall x : cl \iff (\rho, \sigma[x \mapsto a]) \models cl \text{ for all } a \in U\]
\[(\rho, \sigma) \models pre \Rightarrow cl \iff (\rho, \sigma) \models cl \text{ whenever } (\rho, \sigma) \models pre\]
\[(\rho, \sigma) \models cl_1, \cdots, cl_k \iff (\rho, \sigma) \models cl_1 \text{ and } \cdots \text{ and } (\rho, \sigma) \models cl_k\]

Table 1
Semantics of preconditions, clauses and formulae.

In particular, we write \(\rho(R)\) for the set of \(n\)-tuples \((a_1, \cdots, a_n)\) from \(U^n\) associated with the \(n\)-ary predicate \(R\) and \(\sigma(x)\) for the element of \(U\) denoted by the variable \(x\).

We shall mainly be interested in \emph{closed} formulae \(cls\), i.e. clause sequences that have no free variables. Hence the choice of the interpretation \(\sigma\) is immaterial, so we can fix an arbitrary interpretation \(\sigma_0\). We then call an interpretation \(\rho\) of the predicate symbols, a \emph{solution} to the formula \(cls\) provided \((\rho, \sigma_0) \models cls\).

Let \(\Delta\) be the set of interpretations \(\rho\) of predicate symbols in \(R\) over \(U\) and let \(\text{rank}\) be a fixed ranking function.

**Definition 2.3** The lexicographical ordering \(\subseteq\) is defined by \(\rho_1 \subseteq \rho_2\) if and only if there is some \(j \in \{0, \cdots, k\}\) such that the following properties hold:

- \(\rho_1(R) = \rho_2(R)\) for all \(R \in \mathcal{R}\) with \(\text{rank}(R) < j\)
- \(\rho_1(R) \subseteq \rho_2(R)\) for all \(R \in \mathcal{R}\) with \(\text{rank}(R) = j\)
- either \(j\) is maximal in \(\text{rank}\) or \(\rho_1(R) \subset \rho_2(R)\) for at least one \(R \in \mathcal{R}\) with \(\text{rank}(R) = j\)

The subset-ordering \(\subseteq\) is given by \(\rho_1 \subseteq \rho_2\) whenever \(\forall R \in \mathcal{R} : \rho_1(R) \subseteq \rho_2(R)\).

**Fact 2.4** If \(\rho_1 \subseteq \rho_2\) then \(\rho_1 \subseteq \rho_2\) (but not necessarily vice versa).
The functional Fact 2.6

We then have:

\[ \forall \text{ monotonic if } \]

\[ \rho \]

\[ \subseteq \]

\[ \rho \]

Proposition 2.5 (from [11])

The solution set \( \Delta_{\text{cls}} = \{ \rho \in \Delta \mid (\rho, \sigma_0) \models \text{cls} \} \) forms a Moore family, i.e. it is closed under greatest lower bounds (w.r.t. \( \sqsubseteq \)), whenever \( \text{cls} \) is a closed and stratified formula.

The Succinct Solver

In the sequel we shall be interested only in the least solution \( \rho \) as guaranteed by the above proposition; formally it is given by

\[ S(\text{cls}) = \cap \{ \rho \in \Delta \mid (\rho, \sigma_0) \models \text{cls} \} \]

and is the solution computed by the Succinct Solver [11].

For the purposes of this paper we need the basic idea behind the inductive construction of the solution specified above. It suffices with the following imprecise account of the operation of the Succinct Solver; referring to [11] for the details of the considerably more intelligent operation of the algorithm. Given a stratified clause \( \text{cls} \) we may construct two functionals \( N_{\text{cls}} \) and \( F_{\text{cls}} \). For this we shall write \( \rho = \rho_1 \cup \cdots \cup \rho_k \) where each \( \rho_i \) defines the predicates of rank \( i \) (where we assume for simplicity of presentation that all predicates have non-zero rank). We set \( N_{\text{cls}}(\rho_1 \cup \cdots \cup \rho_k) = (\rho_1 \cup \cdots \cup \rho_k) \) whenever \( \rho_1 \cup \cdots \cup \rho_k \) constitutes the new contribution to the predicates arising from one pass through \( \text{cls} \). Then we set

\[
F_{\text{cls}}(\rho) = \begin{cases} 
\rho & \text{if } N_{\text{cls}}(\rho) = (\bot, \ldots, \bot) \\
\rho \cup \varrho_i & \text{if } N_{\text{cls}}(\rho) = (\bot, \ldots, \varrho_i, \cdots) \text{ and } \varrho_i \neq \bot 
\end{cases}
\]

where the intention is that \( i \) indicates the first component of \( N_{\text{cls}}(\rho) \) that is not \( \bot \). The Succinct Solver may then be described as operating until stabilisation of \( F_{\text{cls}} \), i.e.

\[ S(\text{cls}) = \sqcup_i F^i_{\text{cls}}(\bot, \cdots, \bot) \]

For a partial order \( \leq \) (e.g. \( \subseteq \) or \( \sqsubseteq \)) we shall say that the functional \( F \) is \( \leq \)-monotonic if \( \forall \rho_1, \rho_2 : \rho_1 \leq \rho_2 \implies F(\rho_1) \leq F(\rho_2) \) and \( \leq \)-extensive if \( \forall \rho : \rho \leq F(\rho) \). We then have:

Fact 2.6 The functional \( F_{\text{cls}} \) is \( \sqsubseteq \)-extensive and \( \sqsubseteq \)-extensive. (The functional need not be \( \sqsubseteq \)-monotonic nor \( \sqsubseteq \)-monotonic.)

Proof. That \( F_{\text{cls}} \) is \( \sqsubseteq \)-extensive, i.e. \( \rho \sqsubseteq F_{\text{cls}}(\rho) \), is obvious by construction: we
have both $\rho \subseteq \rho$ and $\rho \subseteq \rho \cup g_i$. That $F_{\text{cls}}$ is $\sqsubseteq$-extensive, i.e. $\rho \subseteq F_{\text{cls}}(\rho)$, then follows using Fact 2.4.

That $F_{\text{cls}}$ need not be $\sqsubseteq$-monotonic can be shown by considering the scenario in the proof of Fact 2.4. Consider two predicates $R_1$ and $R_2$ with $\text{rank}(R_i) = i$ and take $\text{cls} = R_1(\cdot)$. Next define $\rho_1$ by $\rho_1(R_1) = \emptyset$, $\rho_1(R_2) = \{\cdot\}$, define $\rho_2$ by $\rho_2(R_1) = \{\cdot\}$, $\rho_2(R_2) = \emptyset$, and define $\rho_3$ by $\rho_3(R_1) = \{\cdot\}$, $\rho_3(R_2) = \emptyset$. We then have $F_{\text{cls}}(\rho_1) = \rho_3$ and $F_{\text{cls}}(\rho_2) = \rho_2$ and $\rho_1 \sqsubseteq \rho_2$ but $\rho_3 \not\sqsubseteq \rho_2$. This shows that $\rho_1 \sqsubseteq \rho_2$ need not imply that $F_{\text{cls}}(\rho_1) \sqsubseteq F_{\text{cls}}(\rho_2)$.

That $F_{\text{cls}}$ need not be $\sqsubseteq$-monotonic can be shown in a similar way by taking $\text{cls} = \neg R_1(\cdot) \Rightarrow R_2(\cdot)$ and defining $\rho_i(R_j) = \emptyset$ except $\rho_2(R_1) = \rho_3(R_2) = \{\cdot\}$. We then have $F_{\text{cls}}(\rho_1) = \rho_3$ and $F_{\text{cls}}(\rho_2) = \rho_2$ and $\rho_1 \sqsubseteq \rho_2$ but $\rho_3 \not\sqsubseteq \rho_2$. This shows that $\rho_1 \sqsubseteq \rho_2$ need not imply that $F_{\text{cls}}(\rho_1) \sqsubseteq F_{\text{cls}}(\rho_2)$. \hfill $\Box$

3 Safety and Security Constraints on ALFP

As a motivating example consider a simple functional language

$$e^l ::= \texttt{tt}^l \mid \texttt{ff}^l \mid x^l \mid (\lambda x.e^0_1)^l \mid (e^l_1 \, e^l_2)^l \mid (\text{if } e^0_1 \text{ then } e^l_1 \text{ else } e^l_2)^l$$

where $e^l$ ranges over labelled expressions. A control flow analysis keeps track of which values (truth values and lambda abstractions) reach which points in the program. Labels need not be unique (and indeed are unlikely to be so after reduction), so we shall take care to ensure that the specification of the control flow analysis does not make that assumption. We axiomatise the analysis using these predicates:

$$C(l, v) \quad \text{indicates the set of values arising at some subexpression labelled } l \text{ may contain the value } v,$$

$$R(x, v) \quad \text{indicates that in the environment the variable } x \text{ may be bound to the value } v,$$

$$P(l, v) \quad \text{indicates that the value } v \text{ may be an actual parameter to some } \lambda\text{-abstraction whose body is labelled } l,$$

$$B(v) \quad \text{indicates that } v \text{ is a boolean value used in the program},$$

$$A(l) \quad \text{indicates that } l \text{ labels the body of some } \lambda\text{-abstraction mentioned in the program}.$$

For a given program $e^l$ we then generate clauses

$$(A, B, C, P, R) \vdash e^l$$

as shown in Figure 1. Here the variables (like $u$, $v$ and $w$) range over the universe $\mathcal{U}$ that consists of the basic values $\texttt{tt}$ and $\texttt{ff}$ and all labels. In the clause for an application $(e^l_1 \, e^l_2)^l$, a typical value of $u$ will be some label $l_0$ denoting a $\lambda$-abstraction. The overall clause generated for the entire program is the conjunction of all the
\[(A, B, C, P, R) \models \text{tt} \iff C(l, \text{tt}) \land B(\text{tt})\]

\[(A, B, C, P, R) \models \text{ff} \iff C(l, \text{ff}) \land B(\text{ff})\]

\[(A, B, C, P, R) \models x \iff \forall v : R(x, v) \Rightarrow C(l, v)\]

\[(A, B, C, P, R) \models (\lambda x. e_0^l)^l \iff (A, B, C, P, R) \models e_0^l \quad \land \quad C(l, l_0) \land A(l_0) \quad \land \quad \forall v : P(l_0, v) \Rightarrow R(x, v)\]

\[(A, B, C, P, R) \models (e_1^l e_2^l)^l \iff (A, B, C, P, R) \models e_1^l \quad \land \quad (A, B, C, P, R) \models e_2^l \quad \land \quad \forall u : C(l_1, u) \Rightarrow (\forall v : C(l_2, v) \Rightarrow P(u, v)) \quad \land \quad (\forall w : C(u, w) \Rightarrow C(l, w)))\]

\[(A, B, C, P, R) \models (\text{if } e_0^l \text{ then } e_1^l \text{ else } e_2^l)^l \iff (A, B, C, P, R) \models e_0^l \quad \land \quad (A, B, C, P, R) \models e_1^l \quad \land \quad (A, B, C, P, R) \models e_2^l \quad \land \quad \forall v : (C(l_1, v) \lor C(l_2, v)) \Rightarrow C(l, v)\]

Fig. 1. Flow logic for a functional language — without safety constraints.

clauses above. It is clearly a stratifiable clause w.r.t. a rank function given by
\(\text{rank}(B) = \text{rank}(A) = 1\) and \(\text{rank}(C) = \text{rank}(R) = \text{rank}(P) = 2\).

**Example 3.1** In order to validate the correct behaviour of the program it would be prudent to impose constraints ensuring that only functions are applied to arguments and that only booleans are used to discriminate between branches of conditionals. Such constraints can be checked by evaluating the following formulae on the least solution to the clause generated above:

for an application \((e_1^l e_2^l)^l\) check that \(\forall u : C(l_1, u) \Rightarrow A(u)\), and

for a conditional \((\text{if } e_0^l \text{ then } e_1^l \text{ else } e_2^l)^l\) check that \(\forall u : C(l_0, u) \Rightarrow B(u)\).

It would be preferable if the constraints could be integrated with the specification of the clause generation.

However, we cannot merely add the above formulae verbatim to the specification in Figure 1, because rather than giving rise to strong constraints that may evaluate
to \textit{false}, they would give rise to \textit{soft} constraints that merely add new “spurious elements” to the predicates \(A\) and \(B\). Indeed, the incorrect program \((\texttt{tt}^{1}\texttt{e}^{2})\) would merely add \(A(\texttt{tt})\) to the predicate \(A\), rather than report a violation. Similarly, the incorrect program \((\lambda x.\texttt{e}^{1})\texttt{e}^{2}\) would merely add \(B(l)\) to the predicate \(B\), rather than report a violation.

It is worth pointing out, that the least solution as produced by \(S\), given Figure 1 as it stands, ensures that no such “spurious elements” are part of the least solution (simply because they are not explicitly demanded to be so by the clause constructed above).

\textbf{Constrained ALFP}

We therefore extend the syntax of ALFP by allowing explicit occurrences of constraints. We distinguish between an assertion \(R(x_1, \cdots, x_n)\) and a constraint by writing the latter as \(R!(x_1, \cdots, x_n)\).

\textbf{Definition 3.2} The set of constrained ALFP clauses, \(cl\), are given by

\[
cl ::= R!(x_1, \cdots, x_n) \mid R(x_1, \cdots, x_n) \mid 1 \mid cl_1 \land cl_2 \mid \forall x : cl \mid \text{pre} \Rightarrow cl
\]

whereas constrained preconditions and formulae are as in Definition 2.1.

A constrained formula \(cls\) is stratified w.r.t. \(\text{rank}\) whenever it has the form \(cls = cl_1, \cdots, cl_k\), and the function \(\text{rank} : \mathcal{R} \rightarrow \{0, \cdots, k\}\) satisfies the following properties for all \(i = 1, \cdots, k\):

(i) \(\text{rank}(R) < i\) for every constraint \(R!\) in \(cl_i\);

(ii) \(\text{rank}(R) \geq i\) for every assertion \(R\) in \(cl_i\);

(iii) \(\text{rank}(R) \leq i\) for every positive query \(R\) in \(cl_i\); and

(iv) \(\text{rank}(R) < i\) for every negative predicate \(\neg R\) in \(cl_i\).

A constrained formula \(cls\) is stratified if there exists a ranking function \(\text{rank}\) such that \(cls\) is stratified w.r.t. \(\text{rank}\).

In the definition of stratified we have taken the view (to become even clearer when discussing constraint violations below) that a predicate must not be used as a constraint until it has been fully asserted.

The \textit{optimal ranking} function is constructed much as before: \(\text{rank}(R) = k\) if there are no (positive or negative) queries to \(R\) nor constraints on \(R\) in \(cls\), otherwise \(\text{rank}(R) = 0\) if there are no assertions to \(R\) in \(cl\), and \(\text{rank}(R) = i\) if \(cl_i\) is the rightmost clause containing an assertion to \(R\).

The considerations of \textit{stratifiability} apply mutatis mutandis; as before there is a normal edge from \(R\) to \(S\) if the clause contains a subclause \(\cdots R \cdots \Rightarrow \cdots S \cdots\) and there is a fat edge from \(R\) to \(S\) if the clause contains a subclause \(\cdots \neg R \cdots \Rightarrow \cdots S \cdots\). (In both cases \(S\) denotes an assertion rather than a constraint.) Much as before a constrained clause \(cl\) is stratifiable if and only if there is no loop containing a fat edge. However, when constructing the stratified formula \(cls\) we may have to
introduce a new rank $k + 1$ and construct it as $cls = cl_1, \cdots, cl_{k+1}$ where each $cl_i$ is obtained from $cl$ by replacing assertions of rank different from $i$ with the clause $1$ and furthermore replacing constraints of rank different from $i - 1$ with the clause $1$.

As an example, the clause $R(a) \Rightarrow (R(b) \land R!(c))$ becomes $R(a) \Rightarrow (R(b) \land 1), R(a) \Rightarrow (1 \land R!(c))$.

**Constraint Validation**

We shall deal with the semantics of constrained ALFP in a syntactic manner, by defining two ways in which to translate a constrained formula into a formula of ALFP.

The function $\text{ignore}$ simply replaces $\cdots R!(\overline{x}) \cdots$ by $\cdots 1 \cdots$ and hence ignores the constraints imposed (see Table 2 for the details):

$$\text{ignore}(\cdots R!(\overline{x}) \cdots) = \cdots 1 \cdots$$

It is useful for extracting the constraint-free part of the formula for which the least solution is desired. If $cls$ is a stratified and constrained formula then clearly $\text{ignore}(cls)$ is a stratified formula of ALFP.

Similarly, the function $\text{enforce}$ replaces $\cdots R!(\overline{x}) \cdots$ by $\cdots R(\overline{x}) \cdots$ and hence ignores the distinction between constraints and assertions (see Table 2 for the details):

$$\text{enforce}(\cdots R!(\overline{x}) \cdots) = \cdots R(\overline{x}) \cdots$$

It is useful for extracting a formula that can be used to check whether or not the constraints are fulfilled. However, even if $cls$ is a stratified and constrained formula, the formula $\text{enforce}(cls)$ need not be a stratified formula of ALFP; as an example consider $1, \neg R(a) \Rightarrow R!(b)$ where $R$ has rank 1.

<table>
<thead>
<tr>
<th></th>
<th>IGNORE (cls)</th>
<th>ENFORCE (cls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R!(x)$</td>
<td>1</td>
<td>$R(x)$</td>
</tr>
<tr>
<td>$R(x)$</td>
<td>$R(x)$</td>
<td>$R(x)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$cl_1 \land cl_2$</td>
<td>$\text{ignore}(cl_1) \land \text{ignore}(cl_2)$</td>
<td>$\text{enforce}(cl_1) \land \text{enforce}(cl_2)$</td>
</tr>
<tr>
<td>$\forall x : cl$</td>
<td>$\forall x : \text{ignore}(cl)$</td>
<td>$\forall x : \text{enforce}(cl)$</td>
</tr>
<tr>
<td>$\text{pre} \Rightarrow cl$</td>
<td>$\text{pre} \Rightarrow \text{ignore}(cl)$</td>
<td>$\text{pre} \Rightarrow \text{enforce}(cl)$</td>
</tr>
<tr>
<td>$cl_1, \cdots, cl_k$</td>
<td>$\text{ignore}(cl_1), \cdots, \text{ignore}(cl_k)$</td>
<td>$\text{enforce}(cl_1), \cdots, \text{enforce}(cl_k)$</td>
</tr>
</tbody>
</table>

Table 2

Ignoring and enforcing the constraints in clauses and formulae.
Given a closed, stratified and constrained formula \( \text{cls} \), we may define the validation function \( \mathcal{V}(\text{cls}) = (\rho, b) \) as follows:

\[
\mathcal{V}(\text{cls}) = (\rho, b) \quad \text{where} \quad \begin{cases} 
\rho = S(\text{ignore}(\text{cls})) \\
b = \begin{cases} 
\text{true} & \text{if } (\rho, \sigma_0) \models \text{ENFORCE}(\text{cls}) \\
\text{false} & \text{otherwise}
\end{cases}
\end{cases}
\]

Here \( \mathcal{V}(\text{cls}) = (\rho, b) \) means that \( \rho \) is the least solution when ignoring the constraints and \( b \) indicates whether or not the constraints are validated.

**Example 3.3** Returning to Figure 1 and Example 3.1 we can now write the clauses to be generated for application and conditionals as in Figure 2: In the case of
\[
\begin{array}{|c|c|}
\hline
\text{OBserve} & \text{OBserve} \\
\hline
R!(x) & \neg R(x) \Rightarrow R^E(x) \\
R(x) & R(x) \\
1 & 1 \\
cl_1 \land cl_2 & \text{OBserve}(cl_1) \land \text{OBserve}(cl_2) \\
\forall x : cl & \forall x : \text{OBserve}(cl) \\
pre \Rightarrow cl & pre \Rightarrow \text{OBserve}(cl) \\
cl_1, \ldots, cl_k & \text{OBserve}(cl_1), \ldots, \text{OBserve}(cl_k) \\
\hline
\end{array}
\]

Table 3
Observing the constraints in clauses and formulae.

\((e_1^{l_1} e_2^{l_2})^l\) this amounts to adding the safety constraint \(\forall u : C(l_1, u) \Rightarrow A!(u)\) and in the case of \(\text{if } e_0^{l_0} \text{ then } e_1^{l_1} \text{ else } e_2^{l_2}\) this amounts to adding the safety constraint \(\forall u : C(l_0, u) \Rightarrow B!(u)\).

The resulting clauses are stratifiable w.r.t. the rank function used before. Furthermore, \textsc{ignore} translates the clauses generated into clauses that are logically equivalent to the ones of Figure 1, whereas \textsc{enforce} translates the clauses generated into clauses that are logically equivalent to the conjunction of the ones of Figure 1 together with the constraints imposed in Example 3.1.

Considering once again the incorrect program \((tt^{l_1} e_2^{l_2})\), we now do report a violation as \(A!(tt)\) fails. Similarly, the incorrect program \((\text{if } (\lambda x.e')^{l_0} \text{ then } e_1^{l_1} \text{ else } e_2^{l_2}\) also gives rise to a violation as \(B!(l)\) fails.

\section{Constraint Violation}

There is an alternative approach to calculating \(\mathcal{V}(cls) = (\rho, b)\) that does not require an extension of the logic with constraints of the form \(R!(x_1, \ldots, x_n)\) (as opposed to assertions of the form \(R!(x_1, \ldots, x_n)\)). We sketch it here, because it has been used in the literature [3] and because it could be used as an alternative implementation strategy; however, the resulting specifications less intuitive than what has been suggested here. The alternative approach makes use of auxiliary predicates, i.e., observation predicates of the form \(R^E(x_1, \ldots, x_n)\), for directly recording the violations. The intention then is that \(R^E\) holds on those elements where \(R!\) would have failed.

To describe the alternative approach in the current setting, we define a function \textsc{observe} that translates \(\cdots R!(\bar{x}) \cdots \) to \(\cdots (\neg R(\bar{x}) \Rightarrow R^E(\bar{x})) \cdots\) where we assume that to each “ordinary” predicate \(R\) there potentially is an “observation” predicate $R^E$.
\(R^E\) (see Table 3 for the details):

\[
\text{OBserve}(\cdots R!(x) \cdots) = \cdots \neg R(x) \Rightarrow R^E(x) \cdots
\]

We shall write \(\mathcal{R}^o\) for the set of ordinary predicates and \(\mathcal{R}^E\) for the set of observation predicates and assume that \(\mathcal{R}\) is the disjoint union of these two sets. Furthermore, we extend the given ranking function \(\text{rank}\) by setting \(\text{rank}(R^E) = k + 1\) for all observation predicates \(R^E\) (regardless of the rank of \(R\)).

We can now prove the formal equivalence of the two approaches. We do so by showing that \(\mathcal{V}\) can be precisely characterised in terms of the emptiness of the observation predicates:

**Proposition 3.4** Writing \(\cdots |_{\mathcal{R}^o}\) for the restriction to ordinary predicates only we have:

\[
\mathcal{V}(\text{cls}) = (\rho, b) \quad \text{where} \quad \begin{cases} 
\rho = S(\text{OBserve}(\text{cls})) |_{\mathcal{R}^o} \\
b = \bigwedge_{R^E \in \mathcal{R}^E} (\rho(R^E) = \emptyset)
\end{cases}
\]

**Proof.** Straightforward by the above reasoning. \(\square\)

## 4 Heuristics for Enforcing Constraints

As a fairly substantial motivating example consider a simplified presentation of the Bell-LaPadula mandatory access control policy for enforcing confidentiality \([2,7]\). The basic entities are subjects (e.g. programs or users), objects (e.g. files), operations (read and write) and security levels (high and low).

The actual operations are specified by statements of the form \(\text{read}(s,o)\) for indicating that the subject \(s\) is initiating a read-operation on the object \(o\) and similarly \(\text{write}(s,o)\) for indicating that the subject \(s\) is initiating a write-operation on the object \(o\). We do not formalise the semantics of this language which should be seen as expressing the access requests that need to be mediated by an operating system.

The discretionary part of the access control policy is syntactically specified by statements of the form \(\text{readable}(o : s_1, \cdots, s_n)\) for indicating that the object \(o\) may be read by any one of the subjects \(s_1, \cdots, s_n\) and by \(\text{writable}(o : s_1, \cdots, s_n)\) for indicating that the object \(o\) may be written by any one of the subjects \(s_1, \cdots, s_n\).

The mandatory part of the access control policy is syntactically specified by statements of the form \(\text{subject}(s : \phi)\) for indicating that the subject \(s\) is allowed to operate at security level \(\phi\) (being one of \(H\) or \(L\)) and by \(\text{object}(o : \phi)\) for indicating that the object \(o\) may be accessed at security level \(\phi\) (being one of \(H\) or \(L\)).

For the purposes of specifying the security policy we shall view the semantics as operating over configurations of the form \((S, O, M, B)\). Here \(S(s,\phi)\) records that the subject \(s\) has been previously allowed to operate at security level \(\phi\); in the classical presentation \([7]\) it aims at capturing \(f_C(s) = f_S(s) = \phi\). Similarly, \(O(s,\phi)\) records that the object \(o\) has been previously allowed to be manipulated at security level
φ; in the classical presentation [7] it aims at capturing $f_O(o) = \phi$. Furthermore, $M(s, o, r)$ captures that the object $o$ has previously been recorded as readable by subject $s$, and $M(s, o, w)$ captures that the object $o$ has previously been recorded as writable by subject $s$; this is as in the classical presentation [7]. Finally, $B(s, o, r)$ indicates that in the current state the subject $s$ has initiated reading the object $o$, and $B(s, o, w)$ indicates that in the current state the subject $s$ has initiated writing the object $o$; also this is as in [7].

We shall develop a simple flow-insensitive analysis for keeping track of these operations and for enforcing the Bell-LaPadula mandatory access control policy (called \textit{mac}). Since the analysis is flow-insensitive it may operate over an “abstract state” $(S, O, M, B)$ as explained above. The clauses can then be generated as shown in Figure 3.

The clauses for reading and writing make use of the policy \textit{mac} also defined in Figure 3. The first line of \textit{mac} considers a situation where a subject $s$ is simultaneously writing an object $o$ and reading an object $o'$. The second line enforces that these operations have indeed been previously allowed as indicated by the access control matrix $M$. The remaining lines enforce that the security classification of $o$ dominates those of $s, o'$ and that the security classification of $o'$ is dominated by those of $s, o$ (relying once more on $f_C = f_S$ in the classical presentation of [7]).

The clause generated is clearly stratifiable. A simple choice of a ranking function is to take $\text{rank}(S) = 1$, $\text{rank}(O) = 1$, $\text{rank}(M) = 1$ and $\text{rank}(B) = 1$. A more interesting choice (as we shall argue shortly) is to take $\text{rank}(S) = 1$, $\text{rank}(O) = 2$, $\text{rank}(M) = 3$ and $\text{rank}(B) = 4$.

\textbf{Example 4.1} To be a bit more concrete consider a program involving one subject \texttt{sub} and two objects \texttt{ob1} and \texttt{ob2}:

\begin{verbatim}
/* declarations */
subject(sub:H);
object(ob1:L); readable(ob1:sub); writable(ob1:sub);
object(ob2:L); readable(ob2:sub); writable(ob2:sub);

/* access operations */
read(sub,ob2); write(sub,ob1);
\end{verbatim}

Here there is a violation of the mandatory part of the access control policy: when \texttt{sub} reads \texttt{ob2} and writes \texttt{ob1} the security level of \texttt{sub} (which is $H$) must be dominated by that of \texttt{ob1} (which is $L$).

Assuming that the program is intended to be legitimate we must modify the security annotations such that \textit{mac} holds. Intuitively, there are two ways to do so: one is to downgrade \texttt{sub} to $L$, the other is to upgrade \texttt{ob1} to $H$. From a security policy point of view it is usually preferred to upgrade the objects rather than downgrading the subjects (see [7] for a discussion). In the present case this means that we prefer to add “spurious elements” to relations like $O$ rather than relations like $S$.

Hence we would like a general heuristics that, based on the rank-information automatically suggests remedial actions. We shall decide to go for an approach
\[(S, O, M, B) \models \text{read}(s, o) \iff B(s, o, r) \land \text{mac} \]
\[(S, O, M, B) \models \text{write}(s, o) \iff B(s, o, w) \land \text{mac} \]
\[(S, O, M, B) \models \text{readable}(o : s_1, \ldots, s_n) \iff M(o, s_1, r) \land \cdots \land M(o, s_n, r) \]
\[(S, O, M, B) \models \text{writable}(o : s_1, \ldots, s_n) \iff M(o, s_1, w) \land \cdots \land M(o, s_n, w) \]
\[(S, O, M, B) \models \text{subject}(s : \phi) \iff S(s, \phi) \]
\[(S, O, M, B) \models \text{object}(o : \phi) \iff O(o, \phi) \]
\[(S, O, M, B) \models S_1; S_2 \iff (S, O, M, B) \models S_1 \land (S, O, M, B) \models S_2 \]

\[
mac = \begin{cases} 
\forall s, o, o' : B(s, o, w) \land B(s, o', r) & \Rightarrow M!(s, o, w) \land M!(s, o', r) \land \\
(S(s, H) \land \neg S(s, L)) \Rightarrow O!(o, H) \land \\
(O(o, L) \land \neg O(o, H)) \Rightarrow (S!(s, L) \land O!(o', L)) \land \\
(O(o', H) \land \neg O(o', L)) \Rightarrow (O!(o, H) \land S!(s, H)) \land \\
(S(s, L) \land \neg S(s, H)) \Rightarrow O!(o', L) 
\end{cases}
\]

Fig. 3. Flow logic for mandatory access control — with security constraints.

where we prefer to remedy the values of higher-rank relations rather than lower-rank relations; in operational terms this means restricting how far the Succinct Solver needs to backtrack and corresponds to its overall mode of operation as described in Section 2. In the present case this suggests taking $\text{rank}(S) = 1$, $\text{rank}(O) = 2$, $\text{rank}(M) = 3$ and $\text{rank}(B) = 4$.

\section*{Acceptable Heuristics}

So far we have been content with an optimal algorithm $S$ for solving an analysis problem expressed by a closed and stratified formula, and an optimal algorithm $V$ for solving and validating the constraints as expressed by a closed and stratified constrained formula.

Turning to the construction of a heuristic algorithm $H$ we shall shortly formulate a notion of optimality and show that in general there does not exist an optimal algorithm. Hence we shall consider candidate functions $H$ of the form $H(\text{cls}) = (\rho, \varrho)$ and define when we consider them to be acceptable. Henceforth, we shall write $\text{cls} \in \mathcal{F}[\text{rank}]$ to express that $\text{cls}$ is a closed constrained formula that is stratified w.r.t. $\text{rank}$.

The first part of the development amounts to allowing $\varrho$ to be freely chosen but to demand that $\rho$ is constructed in an optimal manner from $\text{cls}$ and $\varrho$ and to show
that this is always possible.

**Definition 4.2** A function of the form $H(cls) = (\rho, \varrho)$ is a heuristics provided that $\rho$ is least such that $\rho \supseteq \varrho$ and $(\rho, \sigma_0) \models \text{IGNORE}(cls)$ whenever $cls \in F[\text{rank}]$.

**Proposition 4.3** If $cls \in F[\text{rank}]$ and $\varrho$ is given, then there always exists a least $\rho$ such that $\rho \supseteq \varrho$ and $(\rho, \sigma_0) \models \text{ignore}(cls)$ whenever $cls \in F[\text{rank}]$.

**Proof.** The proof amounts to showing that $\rho = \bigcap \{ \rho' \in \Delta \mid (\rho', \sigma_0) \models \text{ignore}(cls) \land \rho' \supseteq \varrho \}$ always exists and fulfils the demands. Many strategies of proof can be used, but for the purposes of this presentation we restrict ourselves to the case $\forall R \in \mathcal{R} : \text{rank}(R) > 0$ where we can give a simple “syntactic” proof. Given $cls = cl_1, \cdots, cl_k$ we define the formula $cls@\varrho = cl_1', \cdots, cl_k'$ by setting 

$$cl_i' = cl_i \land \bigwedge_{a \in \varrho(R), \text{rank}(R)=i} R(a)$$

Clearly $(\rho, \sigma_0) \models \text{ignore}(cls)@\varrho$ is equivalent to $(\rho, \sigma_0) \models \text{ignore}(cls) \land \rho \supseteq \varrho$ and hence the above formula for $\rho$ amounts to $\rho = S(\text{ignore}(cls)@\varrho)$.

The second part of the development amounts to ensuring that $\varrho$ contains all the “spurious elements” that need to be admitted in order to fulfil the constraints.

**Definition 4.4** A heuristics in the sense of Definition 4.2 of the form $H(cls) = (\rho, \varrho)$ is acceptable provided that $(\rho, \sigma_0) \models \text{ENFORCE}(cls)$ whenever $cls \in F[\text{rank}]$.

**Fact 4.5** An acceptable heuristics exists.

**Proof.** Take $H(cls) = (\top, \top)$. To be able to choose between acceptable heuristics we shall define a partial order for comparing them. We base it on the lexicographic order (rather than the subset-order) in order to capture the intentions expressed towards the end of Example 4.1.

**Definition 4.6** A heuristics $H_1$ is better than a heuristics $H_2$, and equivalently $H_2$ is worse than $H_1$, provided that for all $cls \in F[\text{rank}]$: if $H_1(cls) = (\rho_1, \varrho_1)$ and $H_2(cls) = (\rho_2, \varrho_2)$ then $\rho_1 \supseteq \varrho_2$.

We prefer this definition to the alternative where we instead compare the resulting solutions, as in $\rho_1 \supseteq \rho_2$, because of its focus on the “spurious elements” that need to be added. Clearly the heuristics indicated in the proof of Fact 4.5 is worse than all others.

It would be natural to try to find the best acceptable heuristics. Unfortunately, this is not possible, i.e. we do not have the analogue of a Moore Family result for acceptable heuristics.
Proposition 4.7 There exists no best acceptable heuristics.

Proof. It suffices to find a stratified constrained formula $cls$ for which no acceptable heuristics $H$ can give a best result. For this consider the formula

$$1, \ ((\neg R(a) \land \neg R(b)) \Rightarrow R!(a)) \land ((\neg R(a) \land \neg R(b)) \Rightarrow R!(b))$$

where $\text{rank}(R) = 1$ and the universe is $U = \{a, b\}$. A heuristics $H$ must produce one of the following pairs $(\rho_i, \varrho_i)$:

(i) $\varrho_1(R) = \emptyset$ and $\rho_1(R) = \emptyset$;

(ii) $\varrho_2(R) = \{a\}$ and $\rho_2(R) = \{a\}$;

(iii) $\varrho_3(R) = \{b\}$ and $\rho_3(R) = \{b\}$;

(iv) $\varrho_4(R) = \{a, b\}$ and $\rho_4(R) = \{a, b\}$.

Of these 2–4 are acceptable and 2–3 are acceptable and minimal. Since there are two incompatible minimal choices no optimal choice of an acceptable heuristics can exist.

The implication of this result is that there may be two distinct heuristics that are both minimal (and hence neither is better than the other). In the setting of Example 4.1, this suggests that there may be incomparable ways of making an “insecure” system “secure”, neither of which is better than the other.

A Good Acceptable Heuristics

In order to get a handle of the “arbitrary choices” that Proposition 4.7 opens up for, we shall consider a class of parameterised iterative heuristic algorithms of the form $H[\text{choose, take}]$. Here $\text{choose}$ is a function intended to select an index from a set of indices, and $\text{take}$ is a function intended to select part of a partial solution; it will turn out that our preferred candidate has $\text{choose} = \max$ and $\text{take} = \lambda \varrho. \varrho$ (i.e. the identity).

The definition is given in Table 4. It accepts as input a closed, constrained and stratified formula $cls$ w.r.t. a ranking function $\text{rank}$ (that without loss of generality is assumed to use non-zero ranks only) and produces the pair $(\rho, \varrho)$. It operates in an iterative manner, “backpropagating” any violations to constraints. We use the function $\text{observe}(\cdots R!(x) \cdots) = \cdots \neg R(x) \Rightarrow R^E(x) \cdots$ of Table 3 and we write $\mathcal{R}_i^E$ for the set of ordinary predicates of rank $i$ and similarly $\mathcal{R}_j^E$ for the set of error predicates corresponding to ordinary predicates of rank $j$ and finally we use $\cdots |_{\mathcal{R}'}$ to denote the restriction to a set $\mathcal{R}'$ of predicates.

We shall briefly consider three algorithms. One is $H[\max, \lambda \varrho. \varrho]$ that selects the maximum index for which a violation of the constraints have been observed and then selects the entire error-component corresponding to this index. Another is $H[\min, \lambda \varrho. \varrho]$ that selects the minimum index for which a violation of the constraints have been observed and then selects the entire error-component corresponding to this index.
A potential third algorithm is $\mathcal{H}[\text{first, first}]$ that somewhat informally chooses the first index and tuple for which an error is observed. While the first two algorithms have been precisely defined and are evaluation-order independent, the third algorithm is somewhat informally defined and is clearly evaluation-order dependent (in terms of the operation of the Succinct Solver [11] as surveyed in Section 2).

The correct operation of the algorithm is guaranteed by:

**Proposition 4.8** $\mathcal{H}[\text{choose, take}]$ is an acceptable heuristics if $\forall I : \text{choose}(I) \in I$ and $\forall \tilde{\rho}_i: \tilde{\rho}_i \neq \perp \Rightarrow \text{take} (\tilde{\rho}_i) \neq \perp$.

**Proof.** The assumptions suffice for proving that $\mathcal{H}[\text{choose, take}]$ always terminates because in each iteration of the loop the tuple $((\varrho_1, \cdots, \varrho_i, \perp, \cdots, \perp), i)$ will be strictly increasing w.r.t. the lexicographic order defined using $\sqsubseteq$ for the first component (i.e., $(\varrho_1, \cdots, \varrho_i, \perp, \cdots, \perp)$) and $\leq$ for the second component (i.e., $i$). □

This shows that all three algorithms are acceptable heuristics. As our preferred choice we discount $\mathcal{H}[\text{first, first}]$ because it is evaluation-order dependent and hence hard to characterise; indeed it may appear non-deterministic in case of even simple logically equivalent formula rearrangements.

Finally we prefer $\mathcal{H}[\text{max, } \lambda \varrho, \varrho]$ over $\mathcal{H}[\text{min, } \lambda \varrho, \varrho]$ as it turns out to be the preferred candidate in our motivating example (see Example 4.9 below). However, there are examples (see Example 4.10 below) where different preferences are in order.

**Example 4.9** Returning to Example 4.1, $\mathcal{H}[\text{max, } \lambda \varrho, \varrho]$ suggests the remedial action of upgrading $\text{ob1}$ to $H$ by producing $\varrho(O) = \{(\text{ob1}, H)\}$. This is preferable to $\mathcal{H}[\text{min, } \lambda \varrho, \varrho]$ that suggests the remedial action of downgrading $\text{sub}$ to $L$ by producing $\varrho(S) = \{(\text{sub}, L)\}$. □

**Example 4.10** For an example of a formula where $\mathcal{H}[\text{min, } \lambda \varrho, \varrho]$ performs preferably to $\mathcal{H}[\text{max, } \lambda \varrho, \varrho]$ consider

$$R(a), \quad S(a), \quad T(a), \quad (\neg S(b) \land \neg T(b)) \Rightarrow (S!(b) \land T!(b)), \quad T(b) \Rightarrow R!(b)$$

where $\text{rank}(R) = 1$, $\text{rank}(S) = 2$ and $\text{rank}(T) = 3$. Here $\mathcal{H}[\text{min, } \lambda \varrho, \varrho]$ produces $\varrho(R) = \emptyset$, $\varrho(S) = \{b\}$, $\varrho(T) = \emptyset$ and $\rho(R) = \{a\}$, $\rho(S) = \{a, b\}$, $\rho(T) = \{a\}$, whereas $\mathcal{H}[\text{max, } \lambda \varrho, \varrho]$ produces $\varrho(R) = \{b\}$, $\varrho(S) = \emptyset$, $\varrho(T) = \{b\}$ and $\rho(R) = \{a, b\}$, $\rho(S) = \{a\}$, $\rho(T) = \{a, b\}$. □

## 5 Conclusion

We have extended the flow logic approach to static analysis: instead of merely specifying admissible solutions to analysis problems we additionally specify constraints to be enforced on the admissible solutions. Our use of a simple syntactic distinction between soft assertions, $R(\vec{x})$, and strong constraints, $R!(\vec{x})$, have resulted in very readable specifications, as was illustrated on a simple “typing example” for the $\lambda$-calculus.
Our main contribution is the development of a heuristics that facilitates loosening some of the constraints in order for a security policy to hold for selected programs. The motivating example for this development has been the Bell-LaPadula mandatory access control policy. Our strategy for loosening constraints has been to keep the logical formulae unchanged but to admit more elements in the constraining predicates; we have generally referred to these as “spurious elements”. We have studied and proposed heuristics for iteratively recomputing least solutions to analysis problems in such a way that the final solution adheres to the constraints posed. Also we have shown that a heuristics is all that can be hoped for.

It is worth pointing out that this inherently iterative procedure can still be formulated in a logical setting; this distinguishes our approach from the rather ad-hoc approach of [4, Algorithm 10]. The general mechanism facilitating our development has been to use the rank-information to express the order of preference for adding the “spurious elements”. We believe there to be a fair amount of flexibility in the choice of a ranking function $\text{rank}$ for turning a stratifiable clause into a formula that is stratified wrt. $\text{rank}$; generally it should be possible to assign low ranks to predicates recording simple observations from the program whereas the predicates carrying the actual control flow information may be so interdependent that they all need to get the same rank.

Further work is needed for determining the extent to which this approach is applicable to other security features; possibilities include restricting the behaviour of subjects inside the Trusted Computing Base and identifying the need for enlarging the Trusted Computing Base.

References


