Science of Computer Programming 95 (2014) 200–218

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Security type error diagnosis for higher-order, polymorphic languages

Jeroen Weijers[a], Jurriaan Hage[b,*], Stefan Holdermans[c]

[a] Unaffiliated
[b] Dept. of Inf. and Comp. Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[c] Vector Fabrics, Vonderweg 22, 5616 RM Eindhoven, The Netherlands

HIGHLIGHTS

- Combines type error slicing with type error diagnosis heuristics.
- Source language is higher-order and parametrically polymorphic.
- Features tailored heuristics to deal with parametricity.
- Features tailored heuristics for dealing with dependency based analyses.
- A working implementation in Haskell is available.

ARTICLE INFO

Article history:
Received 20 May 2013
Received in revised form 21 March 2014
Accepted 23 March 2014
Available online 8 April 2014

Keywords:
Type-based program analysis
Security analysis
Error feedback

ABSTRACT

We combine the type error slicing and heuristics based approaches to type error diagnostic improvement within the context of type based security analysis on a let-polymorphic call by value lambda calculus extended with lists, pairs and the security specific constructs declassify and protect. We define and motivate four classes of heuristics that help diagnose inconsistencies among the constraints, and show their effect on a selection of security incorrect programs.

1. Introduction

We take for granted that a compiler for a strongly-typed language refuses to generate code for a program that is not type correct. We also expect compilers to provide a reasonable explanation of what is wrong, but this is not always the case. As the literature study of Heeren [13] documents, the problem of type error diagnosis for the Hindley–Milner type system, and several extensions thereof, has been extensively studied. Thus far, however, most effort in this area has been directed to the intrinsic type systems of functional programming languages, and not to other validation-oriented type based analyses such as security analysis [27].

Consider the following expression (taken from [14]; this is not code anyone would care to write) for which we have provided explicit type and *security* annotations:

$$\begin{array}{l}\textbf{if}\ (\mathit{True} :: \mathtt{Bool}^{\mathsf{H}})\ \textbf{then}\ (\mathit{True} :: \mathtt{Bool}^{\mathsf{L}})\\ \textbf{else}\ (\mathit{False} :: \mathtt{Bool}^{\mathsf{L}}) :: \mathtt{Bool}^{\mathsf{L}}\end{array}$$

* Corresponding author.
E-mail addresses: mail@jeroenweijers.eu (J. Weijers), J.Hage@uu.nl (J. Hage), stefan@vectorfabrics.com (S. Holdermans).

http://dx.doi.org/10.1016/j.scico.2014.03.011
0167-6423/© 2014 Elsevier B.V. All rights reserved.

```
flow ! everyone < !root
type ('α, 'β) pwdEntry =
  {
    userName : 'α string;
    password : 'β string
  }
val checkPwd :
  'α string → (!everyone, !root) pwdEntry → 'α bool
val pwdList : ((!everyone, !root) pwdEntry, !everyone) list
```

Fig. 1. Password.fmli.

```
type pwdEntry =
  {
    userName : string;
    password : string
  }
let checkPwd p uRec =
  if p = uRec.password then true else false
let pwdList =
  let jimmy : pwdEntry =
      { userName = "Jimmy"; password = "1234" } in
  ...
  [ jimmy; ... ]
```

Fig. 2. Password.ml.

In this expression, an annotation H means that the expression to which it is attached delivers values that are highly confidential. Expressions annotated with L deliver values of low confidentiality. The purpose of security analysis is to verify that data is never leaked to expressions that may also evaluate to strictly less confidential data. Intuitively, in a security correct program no value may inadvertently influence a value of a lesser security level. In the expression above, this is not the case: since the value of the condition decides whether the **then** or **else** part must be evaluated, observing the value of the low confidentiality result reveals information about the highly confidential condition. Therefore, the expression is not security type correct (although it *is* type correct) and it should be rejected. The problem can, for example, be fixed by changing the annotation on the condition to L, or by changing the annotation on the complete expression to H.

King et al. [18] observed that information-flow reporting techniques are inadequate to explain security type errors. In their work they assign information-flow blame, and provide traces of the Java programs they analyse to show how values of high confidentiality end up in locations that may only expose values of lower confidentiality. Their work, however, does not transfer easily to a functional setting: their analysis is (largely) context-insensitive (i.e., monovariant), the language they consider is first-order, there is no discussion of parametric polymorphism, and their trace-like explanation does not seem so natural for a higher-order functional language. Moreover, as the authors themselves suggest, their work has not been combined with heuristics to further prune the traces they provide (see, e.g., [6,9,13,15,16] for work on such heuristics). We reuse some of these heuristics in our work. See Section 6 for more details.

In this light, this paper offers the following contributions:

- We are the first to combine the type error slicing approach of Haack and Wells [8] with the heuristic approach of Heeren [13].
- We introduce and motivate a number of heuristics, divided into four essentially different categories, as described in Section 2. We provide a substantial number of example programs that show how our approach works.
- We provide a polyvariant prototype implementation of our work obtainable at `http://www.cs.uu.nl/wiki/bin/view/Hage/Downloads` for a polymorphic lambda-calculus extended with recursion, lists and tuples, and special security specific constructs, **declassify** and **protect**.

The paper is structured as follows. We further motivate our work in Section 2, and describe our approach in Section 3. In Section 4 we describe the security-annotated type system, followed by the corresponding inference algorithm in Section 5. In Section 6 we describe our heuristics, and Section 7 provides further illustrations of these heuristics as a first step in validating our work. In Section 8 we discuss related work, and Section 9 concludes.

2. Motivation

In this section, we further motivate our work by providing an example program written in FlowCaml [22] to illustrate the kind of diagnostics that is to be expected from this system.

The FlowCaml implementation of password verification is given in Figs. 1, 2, and 3. Fig. 1 defines the security lattice that the implementation uses: the line *flow* ! *everyone* < !*root* declares a flow relation that tells us that !*everyone* is of strictly lower confidentiality than !*root*. Comparing this with the example in the introduction, !*everyone* is like L, and !*root* plays the role of H. The fact that this source file has such a flow statement conveys to FlowCaml the fact that the code in the file must be checked not to violate the rules imposed by the security type system. Fig. 1 further declares the datatype *pwdEntry* that represents password entries. Here, the $ACC\,\alpha$ annotation represents the security level of the *userName*, and $ACC\,\beta$ the security level of the *password*. The function *checkPwd* checks a single password entry, expecting the *username* to have low confidentiality, and *password* to have high confidentiality. In Fig. 2 implementations in plain OCaml are given.

The *login* function in Fig. 3 takes a user name and a password as its arguments. It uses the former to retrieve the corresponding entry from *pwdList* (using the auxiliary function *getUserRecord*), and then uses *checkPwd* to verify that the right password was provided. The types of *login* and *getUserRecord* are inferred by FlowCaml. Note that the code provided in Fig. 3 may look like ordinary OCaml code, but as a consequence of the presence of the flow statement at the top, the compiler checks that it does not violate the rules of the security type system.

```
flow ! everyone < !root

let getUserRecord name =
  let rec lookUp l ls =
    match ls with
      [ ] → None
    | u :: us → if u.Password.userName = l
                then Some u
                else lookUp l us
  in
    lookUp name Password.pwdList

let login user pwd =
  let userRecord = getUserRecord user
    in match userRecord with
      None → false
      | Some u → Password.checkPwd pwd u
```

Fig. 3. Login.fml.

```
log = fn x ⇒ protect x Low

boolVal = protect True Low
lVal = protect 2 Low

zl = Cons (protect 1 Low) (protect Nil High)

id = fn x ⇒ let y = x in y

main = log (if id (id boolVal) then id lVal
                                else hd zl)
```

Fig. 4. An example to illustrate our work.

```
The structure of list: zl at (l 11, c 40)
applied to head in the expression:
  hd zl at (l 11, c 37)
is protected at level: High.
This causes the result of the head operation to be protected at level: High.
Instead a value protected at level: Low that was expected by: log ....
```

Fig. 5. Security type error given for program in Fig. 4.

In the absence of a declassification operator, a security analysis would demand that the output of *checkPwd* have security level !*root*, so in order to be able to convey the output of the comparison (a boolean) to everyone, the result of the comparison must be declassified. This is achieved within FlowCaml by writing the *checkPwd* function in plain OCaml in Fig. 2.

Consider now the situation that we had instead provided the type signature

$$'\alpha\ \textbf{string} \to (!everyone, !everyone)\ pwdEntry \to\ '\alpha\ \textbf{bool}$$

for *checkPwd*. Then one would expect the compiler to complain about passing an insecure *pwdEntry* into a function that expects otherwise in the definition of *login* in *Login.fml*. Instead the compiler returns the following message:

```
File "Login.fml", line 11, characters 0-144:
This expression generates the following information flow:
root < everyone
which is not legal.
```

telling us only that there is some illegal flow, and the location points to the line where the *login* function begins. The flow that is found indeed exists within the *login* function and is illegal, but it is not explained where the value protected at level *root* originates from, and where it is found to flow to a location for values protected at level *everyone*. In addition, there is no indication what the programmer can do to fix the problem. In this particular case it might not be very hard to find the cause of the problem, but in more complicated programs error messages like these are not very helpful.

Before we go on we provide a small but non-trivial example of what our system provides. In this example an invocation of **hd** leads to an inconsistency in Fig. 4: the *log* function expects a value of confidentiality *Low*, but the expression **hd** *zl* makes one of the possible arguments to *log* a value of confidentiality *High*, which can be traced back to the fact that the structure of the list has *High* confidentiality and this is inherited by the value returned by **hd**, because taking the head of the list reveals information about the structure of the list. The error message that results can be found in Fig. 5. Strikingly, the error message changes substantially (to the one in Fig. 6) when we change **protect** 1 *Low* in the definition of *zl* to **protect** 1 *Medium*. This is because we now have two reasons to think that **hd** *zl* is not of level *Low*. So now the application of *log* to its argument is blamed. Note that the message does still explain that the too high confidentiality arises from *zl*. Note that the applications of *id* are never blamed for anything, and that how *id* is exactly implemented has no bearing on what is derived for it. This is what we expect.

3. Approach

In this section we describe at a high level our approach and the heuristics that come into play. The type based security analysis we provide error diagnosis for is polyvariant and includes a rule for subeffecting, but not full subtyping. The rule for subeffecting is somewhat more restrictive than that of full subtyping, but this is not relevant for our work here (and note that the loss of precision is to a large extent compensated for by the polyvariance of the analysis). Our source language

```
Error in application:
(log if (id (id boolVal)) then (id lVal) else head zl) at (l 10, c
      8)
An argument protected at most level: Low is expected by: log ...
The argument provided:
  if (id (id boolVal)) then (id lVal) else head zl at (l 10, c 13)
is protected at level: High
Because of the following sub—expressions: zl at (l 11, c 40)
```

Fig. 6. Security type error given for program in Fig. 4.

```
minUnsat (D) = do
  M = { }
  while satisfiable M {
    C = M
    while satisfiable C {
      let e ∈ D − C
      C = C ∪ {e}
    }
    D = C
    M = M ∪ {e}
  }
  return M
```

Fig. 7. Computing a minimal unsatisfiable set of constraints.

is a higher-order polymorphic functional language, very much akin to FlowCaml, an implementation based on the Core ML language [22].

As we explain below, our work essentially combines the approaches of Hage and Heeren [9,13], and Haack and Wells [8]. Like Haack and Wells, we first compute a (security) type error slice when a security type error has been found. A security type error slice is a program slice (or fragment) that only contains those parts of the program that contribute to the error. The constraints that are needed to construct such a slice together form a minimal unsatisfiable set of constraints: remove any of its elements, and it becomes satisfiable. If a program contains multiple errors, then the next error will be revealed only after the first one has been corrected.

We now provide the details on how to compute a minimal unsatisfiable constraint set from the complete set of constraints. Haack and Wells [8], and Stuckey, Sulzmann and Wazny [26] both present an algorithm for computing such a set. We follow the algorithm presented by Stuckey, Sulzmann and Wazny (see Section 7 of [26]), as displayed here in Fig. 7.

The algorithm takes an unsatisfiable set of constraints D and constructs a minimal unsatisfiable set of constraints M iteratively. The algorithm consists of two nested while loops: the inner loop adds a constraint from the original set D minus C, to a copy, called C, of the unsatisfiable set collected thus far. When C becomes unsatisfiable in the inner loop, then the last added constraint is known to contribute to the error, and it is then added to the minimal unsatisfiable set M. This process continues until the set M becomes unsatisfiable.

Displaying the security type error slice is a first approximation for the type error, but in some cases there may be strong evidence that a smaller set of locations will do just as well, or that we can suggest a fix for the mistake. In Section 6, we present a number of heuristics that inspect the constraints in the minimal unsatisfiable set to determine whether certain constraints/locations should never be marked as the cause of a security type error or, just the opposite, a particular constraint should be blamed for the mistake. In the latter case, a very specific security type error message can be provided. Because a compiler cannot know what the intentions of the programmer are, we are taking a risk here. This risk can be mitigated by, for example, offering various security error messages and allowing the programmer to scan through these. Our implementation currently offers only one error message. Adding a facility such as we just described is only a matter of engineering.

There are two good reasons to start from a minimal unsatisfiable set of constraints. First, it is impossible to blame a constraint that cannot be responsible for the mistake (which may be considered a “soundness property” for the heuristics). Second, the heuristics need only look at a restricted set of constraints, which we may hope is much smaller than the complete set of constraints.

In the end, we will be left with a set of constraints that will receive the blame for the mistake. When a constraint is generated, meta information about the AST node where it was generated is added to the constraint. The collection of AST nodes associated with the constraints in the minimal unsatisfiable subset together form the program slice. This slice can in itself be presented as an error message, with some explanation on the nature of the error [8].

Because our analysis is polyvariant, simplification/solving of constraints will take place for every definition, i.e., at any point that generalisation is to take place. During this process of simplification, the error diagnosis process we have just sketched will be invoked whenever simplification results in an inconsistency.

Our heuristics can be divided into four categories:

- generic heuristics that borrow heavily from earlier work and apply just as well in the current setting, e.g., a heuristic that filters out constraints that equate the security level of a let-expression with that of the let-body.

- propagation heuristics that prevent blaming code that only propagates the security levels of their inputs. For example, blaming a function *inc* that increments an integer value (and that implicitly maintains the level of confidentiality of its input) cannot be sensibly blamed for an inconsistency. In practice, we expect most functions to be of this kind. Changes in levels of confidentiality are most likely to arise from implicit control-flow (see the example in the introduction), from security specific operations like **protect** and **declassify** and from values explicitly provided with security annotations.
- heuristics derived from the assumption that programmers may be used to dealing with the intrinsic type system, and will be unaware of the subtle differences that arise from the fact that a security type system is in fact a dependency analysis [2]. We systematically derive these heuristics from observable differences between the specification of security typing and the underlying intrinsic type system.
- heuristics that are specific to security analysis, in particular the operations of **declassify** and **protect**.

Although we have no evidence to support this, we believe that many of the heuristics and our approach can be reused in other settings besides security analysis. For example, the heuristics in the third category are also likely to apply to other dependency analyses.

4. The security type system

The language we use is based upon the Fun language, a simply typed, call-by-value lambda calculus (see Chapter 3 of [21]). We have extended Fun with a few special purpose security program constructs as well as constructs for dealing with lists and tuple in order to make the analysis and examples more interesting. We call this extended language sFun++. Before we can introduce the language below, we define the following syntactic categories

n	$\in$	**Nat**	*natural numbers* ,
b	$\in$	**Bool**	*booleans* ,
e	$\in$	**Exp**	*expressions* ,
f, x	$\in$	**Var**	*variables* ,
$u, \oplus$	$\in$	$\mathbf{Op}', \mathbf{Op}$	*unary and binary operators* ,
p	$\in$	**Prog**	*program* ,
d	$\in$	**Decl**	*declarations* ,
s	$\in$	**Sec**	*security levels*

Most categories should speak for themselves. We note that the category **Sec** ranges over security levels, which we assume to form a lattice [4], meaning that given a finite non-empty set S of security levels, there is a unique lowest security level that is at least as secure as each element of S. The join operator of this lattice is, as usual, denoted by $\sqcup$.

The abstract syntax of our language is defined as:

$$\begin{array}{rcl}
p & ::= & d^* \\
d & ::= & f = e_1 \\
e & ::= & n \mid b \mid x \mid \mathbf{fn}\, x \Rightarrow e_0 \mid \mathbf{fun}\, f\, x \Rightarrow e_0 \\
 & \mid & e_0\, e_1 \mid \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \\
 & \mid & \mathbf{let}\, x = e_0\, \mathbf{in}\, e_1 \mid e_1 \oplus e_2 \mid u\, e_1 \\
 & \mid & \mathbf{Cons}\, e_1\, e_2 \mid \mathbf{Nil} \mid (e_1, e_2) \\
 & \mid & \mathbf{fst}\, e_1 \mid \mathbf{snd}\, e_1 \mid \mathbf{null}\, e_1 \mid \mathbf{hd}\, e_1 \mid \mathbf{tl}\, e_1 \\
 & \mid & \mathbf{declassify}\, e_0\, s \mid \mathbf{protect}\, e_0\, s
\end{array}$$

A program p is a list of declarations, and each declaration binds an expression to an identifier. As in [21], $\mathbf{fn}\, x \Rightarrow e_0$ defines a non-recursive function and $\mathbf{fun}\, f\, x \Rightarrow e_0$ a recursive one. In the latter, the identifier f refers to the recursively defined function. Function application is left associative. Local definitions $\mathbf{let}\, x = e_0\, \mathbf{in}\, e_1$ are non-recursive. Top level declarations are syntactic sugar for a nested let. This means that a declaration can only use functions that are declared earlier in the program. For data types we have pairs (e_1, e_2), and lists are built from **Cons** and **Nil** as usual. Pairs are destructed by **fst** and **snd**, and **hd** and **tl** destruct lists. Finally, we can test for the empty list with the **null** predicate.

The two security constructs are **protect** and **declassify**. The expression $\mathbf{protect}\, e_0\, s$ increases the level of protection (security) of an expression e_0 to level s. It is important to note that this construct can only increase the security level of e_0, so level s has to be at least as secure as the level at which e_0 was previously protected. The **protect** construct is essential to the language: without it we would have no way of creating values at a high level of confidentiality, and there would never be a reason to consider a program type incorrect. With **protect** the programmer can explicitly mark data as confidential, and the type system will decide whether this confidentiality is ever breached.

The expression $\mathbf{declassify}\, e_0\, s$ does exactly the opposite: it decreases the security level of e_0 to level s. The presence of this construct implies that our analysis is not sound with respect to confidentiality. However, without some form of declassification it will be hard to write useful programs. For example, we cannot write a valid program that informs an unauthorised user that he or she entered an invalid password (assuming the password information is confidential).

4.1. The sFun++ type language

As usual, we specify the security type system as an annotated type system ([20], and Chapter 5 of [21]). Our security analysis is polyvariant, which means that we can generalise over annotation variables. The relations between annotation variables, and restrictions on them are expressed as constraints. These constraints are added to types in the style of qualified types [17].

We introduce the following new syntactic categories:

α	$\in$	**TyVar**	*type variables*
β	$\in$	**AnnVar**	*annotation variables*
π	$\in$	**Constr**	*constraints*
l	$\in$	**Levels**	*security levels*
φ	$\in$	**Ann**	*security annotations*
τ	$\in$	**Ty**	*annotated types*
ρ	$\in$	**QualifiedTypes**	*qualified types*
σ	$\in$	**TyScheme**	*annotated type schemes*
C	$\in$	**Constraints**	*constraint sets*
Γ	$\in$	**TyEnv**	*type environments*

The sets of annotation variables, **AnnVar**, and type variables, **TyVar** are assumed to be mutually disjoint. An annotation is either some security level l (taken from any given security lattice), or an annotation variable β. Constraints relate two security levels, where we take $\varphi_1 \sqsubseteq \varphi_2$ to mean that φ_2 is at least as secure as φ_1.

$$\begin{array}{lcl} \varphi & ::= & l \mid \beta \\ \pi & ::= & \varphi_1 \sqsubseteq \varphi_2 \end{array}$$

We then define a three-layer type language:

$$\begin{array}{lcl} \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{List}\,\tau^{\varphi} \\ & \mid & ({\tau_1}^{\varphi}, {\tau_2}^{\varphi}) \mid {\tau_1}^{\varphi} \rightarrow {\tau_2}^{\varphi} \mid \alpha \\ \rho & ::= & \pi \Rightarrow \rho \mid \tau \\ \sigma & ::= & \forall\alpha.\sigma \mid \forall\beta.\sigma \mid \rho \end{array}$$

The first layer, τ, consists of annotated types for the primitive types, lists, pairs and function types. We introduce type variables in order to be able to construct type schemes. Qualified types ρ consist of a type and a sequence of constraints that further restrict the type. Finally, type schemes allow us to quantify universally over type and annotation variables. Note that in contrast to FlowCaml [22] we do have annotations on pairs, although we do not really need them. For uniformity with the treatment of other datatypes that do need them, we have also included them here.

A type environment Γ is a mapping from variables x to a pair consisting of a type scheme σ and a top-level annotation for x.

$$\begin{array}{lcl} \Gamma & ::= & \emptyset \mid \Gamma\,[\,x \mapsto (\sigma, \varphi)\,] \\ C & ::= & \emptyset \mid \{\pi_1, \ldots, \pi_n\} \cup C \end{array}$$

The pair associated with a variable x in an environment Γ is written $\Gamma\,(x)$; it returns a pair associated with the rightmost occurrence of x. Note that top level annotations are not present in type schemes, but are stored separately in the type environment. As a result, we cannot generalise over these annotations, and therefore declarations will never be polyvariant in their top level annotation. Although one might think that this causes a loss of expressivity, the presence of a rule for subeffecting will counter this loss. Constraint sets C will be used to store a constraint environment in our typing judgement.

4.2. The security type rules

In Fig. 8 we give the non-syntax directed type rules for our security analysis, following the usual conventions for such (see [20], and Chapter 5 of [21]). The main judgement is $\Gamma, C \vdash e : \sigma^{\varphi}$, which reads "under the type environment Γ and constraint environment C, the expression e can have type σ and is protected at level φ". The constraint environment C contains the relations between security levels that define the security lattice proper, as well as the constraints that should hold for e.

We note that some of the rules in Fig. 8 may specify a family of rules. For example, in the rule [*t-true*], φ ranges over the elements of the chosen lattice, and thereby provides a single rule for each such element. In the case of [*t-list*], we have a rule for each combination of choices for φ_1 and φ_2. Also note that the rule *t-int* specifies an infinite family of rules: one for every combination of an integer n and an element of the security lattice φ. We shall now discuss the more interesting rules one by one.

As seen in rules [*t-Nil*] and [*t-Cons*], lists have separate security annotations for the elements and the structure of the list. Moreover, these annotations are independent: we can, say, protect the length of a list at a higher level than the actual contents, and vice versa. Obviously, for the Cons case, the annotation on the new element φ_1 should correspond to the annotation on the elements of the list, and the annotation on the argument list φ propagates to the result list. For rules [*t-null*], [*t-hd*] and [*t-tl*] we note that the functions **null**, **hd** and **tl** all reveal information about the structure of the list, and **hd** additionally reveals information about the contents of the list; hence the least upper bound in the consequent of [*t-hd*]. As mentioned before, for reasons of consistency, pairs also have a top level annotation, although we learn nothing from knowing the structure of a pair that the type hasn't already conveyed (see [*t-Pair*], [*t-fst*] and [*t-snd*]).

Function application ([*t-app*]) requires the annotation of the provided argument to be equal to the annotation of the expected argument. The annotation of the application result is the least upper bound of the security level of the result, and of the result of applying the function. The reason for the former, is that applying a function may reveal information about that function. Functions declared at top level are assigned the lowest security level $\perp$, so then the annotation of the application only depends on the annotation on the result of the body; the same reasoning applies to operators in the rules [*t-Bin-op*] and [*t-Un-op*].

For the rule [*t-if*], recall from the introduction that the security annotation on the condition should also propagate to the security level of the result of the conditional. Otherwise, the outcome may leak secure information accessed during the evaluation of the condition.

The influence of **protect** and **declassify** on the security levels are expressed in the rules [*t-protect*] and [*t-declass*], respectively. The expression **protect** $e_0\ \varphi_0$ is protected at level φ_0, under the condition that e_0 is at most as secure as φ_0. Declassification does exactly the opposite, lowering the level of security.

As evidenced by many of our rules, we typically insist that different subexpressions have exactly the same annotated type. In the rule [*t-if*], for example, the condition, the then part and the else part need to have exactly the same security level. This is by itself too restrictive, making reasonable programs unanalysable. Therefore, a rule for subeffecting, [*t-sub*], is introduced that may increase the level of protection for an expression.

The rules for annotation generalisation and instantiation, [*t-ann-gen*] and [*t-ann-ins*], are analogous to the standard rules [*t-gen*] and [*t-ins*]. The rule [*t-gen*] uses the function $ftv(x)$ to compute the free type variables in x, and [*t-ann-gen*] uses a similar function *fav* to compute the free annotation variables. The functions are straightforward, and we omit the details.

Qualification ([*t-qual*]) allows us to move a constraint from the constraint set into the (qualified) type, and resolution ([*t-res*]) allows us to do the opposite. In Fig. 9, rules for reasoning with constraints are provided. The rule [*c-in*] states that if π is in C then π holds. Transitivity is provided through the rule [*c-trans*], and reflexivity through [*c-reflex*]. The rules [*c-bot*] and [*c-top*] state that any φ is above (or on) $\perp$ and below (or on) $\top$.

Our type system specification follows those described in [2,22]. We are confident therefore that — omitting the rule for declassification —, the security type system satisfies a non-interference result. Intuitively, non-interference implies that replacing an expression of some confidentiality with any other (of the same level of confidentiality), does not change the values of any expression of lower confidentiality. Since these properties are well-known, and our focus in this paper is on deriving heuristics from the security type system, we forego the definition of semantics that we need to formally specify the non-interference result.

5. Security type inference

In this section an algorithmic version of the inference system from Section 4 is presented. We start by introducing substitutions and unification.

During type inferencing a variable is introduced whenever a type or annotation cannot be determined. Over time, we may learn more to further constrain what the variable represents. This we encode, as usual, in a substitution, θ. A substitution is a pair of finite mappings: one from type variables to types, and one from annotation variables to annotations. The sets of type and annotation variables are assumed to be disjoint. Substitutions are idempotent. This means that $\theta\ (\theta\ \tau)$ is equal to $\theta\ \tau$. Composition of substitutions is defined as follows: $(\theta_2.\theta_1)\ (\tau) \equiv \theta_2\ (\theta_1\ (\tau))$.

The empty substitution is represented by *Id*. Type variables that are not in the domain of the substitution are assumed to map to themselves. In order to describe how substitutions should be applied, we introduce a new syntactic category for all variables:

$$\zeta \quad \in \quad \mathbf{TyVar} \cup \mathbf{AnnVar} \qquad \textit{all variables}$$

Bound variables should not be substituted for. Therefore we introduce a new sort of substitution θ/ζ that prevents the bound variable ζ from being substituted by θ. Fig. 10 defines how substitutions are applied to the various syntactic categories employed in the paper: types, annotations, type schemes, qualified types, type environments, constraint environments, and constraints (in that order).

As usual, we employ a unification algorithm $\mathcal{U}$ to reconcile two given annotated types. The unification algorithm is defined both for annotated types and annotations. The result of unification is a substitution which has the property that

Typing judgements $\boxed{\Gamma, C \vdash e : \sigma^{\varphi}}$

$$\frac{}{\Gamma, C \vdash n : \texttt{Int}^{\varphi}}\,[\textit{t-int}] \qquad \frac{}{\Gamma, C \vdash \mathbf{true} : \texttt{Bool}^{\varphi}}\,[\textit{t-true}] \qquad \frac{}{\Gamma, C \vdash \mathbf{false} : \texttt{Bool}^{\varphi}}\,[\textit{t-false}]$$

$$\frac{}{\Gamma, C \vdash \mathbf{Nil} : (\texttt{List}\ \tau^{\varphi_2})^{\varphi_1}}\,[\textit{t-Nil}] \qquad \frac{\Gamma, C \vdash e_1 : \tau^{\varphi_1} \quad \Gamma, C \vdash e_2 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{Cons}\, e_1\, e_2 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}\,[\textit{t-Cons}]$$

$$\frac{\Gamma, C \vdash e_1 : \tau_1{}^{\varphi_1} \quad \Gamma, C \vdash e_2 : \tau_2{}^{\varphi_2}}{\Gamma, C \vdash (e_1, e_2) : (\tau_1{}^{\varphi_1}, \tau_2{}^{\varphi_2})^{\varphi}}\,[\textit{t-Pair}] \qquad \frac{\Gamma\,(x) = (\sigma, \varphi)}{\Gamma, C \vdash x : \sigma^{\varphi}}\,[\textit{t-var}]$$

$$\frac{\Gamma\,[\,x \mapsto (\tau_x, \varphi_x)\,], C \vdash e_0 : \tau_0{}^{\varphi_0}}{\Gamma, C \vdash \mathbf{fn}\, x \Rightarrow e_0 : \tau_x{}^{\varphi_x} \rightarrow^{\varphi} \tau_0{}^{\varphi_0}}\,[\textit{t-fn}] \qquad \frac{\Gamma\,[\,f \mapsto (\tau_x{}^{\varphi_x} \rightarrow \tau_0{}^{\varphi_0}, \varphi)\,]\,[\,x \mapsto (\tau_x, \varphi_x)\,], C \vdash e_0 : \tau_0{}^{\varphi_0}}{\Gamma, C \vdash \mathbf{fun}\, f\, x \Rightarrow e_0 : \tau_x{}^{\varphi_x} \rightarrow^{\varphi} \tau_0{}^{\varphi_0}}\,[\textit{t-fun}]$$

$$\frac{\Gamma, C \vdash e_1 : \tau_2{}^{\varphi_2} \rightarrow^{\varphi} \tau_0{}^{\varphi_0} \quad \Gamma, C \vdash e_2 : \tau_2{}^{\varphi_2}}{\Gamma, C \vdash e_1\, e_2 : \tau_0{}^{\varphi \sqcup \varphi_0}}\,[\textit{t-app}] \qquad \frac{\Gamma, C \vdash e_0 : \texttt{Bool}^{\varphi} \quad \Gamma, C \vdash e_1 : \tau^{\varphi} \quad \Gamma, C \vdash e_2 : \tau^{\varphi}}{\Gamma, C \vdash \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 : \tau^{\varphi}}\,[\textit{t-if}]$$

$$\frac{\Gamma, C \vdash e_1 : \sigma^{\varphi} \quad \Gamma\,[\,x \mapsto (\sigma, \varphi)\,], C \vdash e_2 : \tau_2{}^{\varphi_2}}{\Gamma, C \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau_2{}^{\varphi_2}}\,[\textit{t-let}]$$

$$\frac{\Gamma, C \vdash e_1 : \tau_{\oplus}^{1\,\varphi} \quad \Gamma, C \vdash e_2 : \tau_{\oplus}^{2\,\varphi}}{\Gamma, C \vdash e_1 \oplus e_2 : \tau_{\oplus}{}^{\varphi}}\,[\textit{t-Bin-op}] \qquad \frac{\Gamma, C \vdash e_1 : \tau_{\oplus}^{1\,\varphi}}{\Gamma, C \vdash u\, e_1 : \tau_{\oplus}{}^{\varphi}}\,[\textit{t-Un-op}]$$

$$\frac{\Gamma, C \vdash e_1 : (\tau_1{}^{\varphi_1}, \tau_2{}^{\varphi_2})^{\varphi}}{\Gamma, C \vdash \mathbf{fst}\, e_1 : \tau_1{}^{\varphi \sqcup \varphi_1}}\,[\textit{t-fst}] \qquad \frac{\Gamma, C \vdash e_1 : (\tau_1{}^{\varphi_1}, \tau_2{}^{\varphi_2})^{\varphi}}{\Gamma, C \vdash \mathbf{snd}\, e_1 : \tau_2{}^{\varphi \sqcup \varphi_2}}\,[\textit{t-snd}]$$

$$\frac{\Gamma, C \vdash e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{null}\, e_1 : \mathbf{bool}^{\varphi}}\,[\textit{t-null}] \qquad \frac{\Gamma, C \vdash e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{hd}\, e_1 : \tau^{\varphi_1 \sqcup \varphi}}\,[\textit{t-hd}] \qquad \frac{\Gamma, C \vdash e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}{\Gamma, C \vdash \mathbf{tl}\, e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}\,[\textit{t-tl}]$$

$$\frac{\Gamma, C \vdash e : \tau^{\varphi} \quad C \vdash \varphi_0 \sqsubseteq \varphi}{\Gamma, C \vdash \mathbf{declassify}\, e\, \varphi_0 : \tau^{\varphi_0}}\,[\textit{t-declass}] \qquad \frac{\Gamma, C \vdash e : \tau^{\varphi} \quad C \vdash \varphi \sqsubseteq \varphi_0}{\Gamma, C \vdash \mathbf{protect}\, e\, \varphi_0 : \tau^{\varphi_0}}\,[\textit{t-protect}]$$

$$\frac{\Gamma, C \vdash e : \tau^{\varphi_1} \quad C \vdash \varphi_1 \sqsubseteq \varphi}{\Gamma, C \vdash e : \tau^{\varphi}}\,[\textit{t-sub}]$$

$$\frac{\Gamma, C \vdash e : \sigma^{\varphi} \quad \beta \notin \mathit{fav}(\Gamma) \cup \mathit{fav}(C) \cup \mathit{fav}(\varphi)}{\Gamma, C \vdash e : (\forall \beta.\sigma)^{\varphi}}\,[\textit{t-ann-gen}] \qquad \frac{\Gamma, C \vdash e : (\forall \beta.\sigma)^{\varphi}}{\Gamma, C \vdash e : ([\,\beta \mapsto \varphi_1\,]\,\sigma)^{\varphi}}\,[\textit{t-ann-ins}]$$

$$\frac{\Gamma, C \vdash e : \sigma^{\varphi} \quad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma, C \vdash e : (\forall \alpha.\sigma)^{\varphi}}\,[\textit{t-gen}] \qquad \frac{\Gamma, C \vdash e : (\forall \alpha.\sigma)^{\varphi}}{\Gamma, C \vdash e : ([\,\alpha \mapsto \tau\,]\,\sigma)^{\varphi}}\,[\textit{t-ins}]$$

$$\frac{\Gamma, C \cup \{\,\pi\,\} \vdash e : \rho^{\varphi}}{\Gamma, C \vdash e : (\pi \Rightarrow \rho)^{\varphi}}\,[\textit{t-qual}] \qquad \frac{\Gamma, C \vdash e : (\pi \Rightarrow \rho)^{\varphi} \quad C \vdash \pi}{\Gamma, C \vdash e : \rho^{\varphi}}\,[\textit{t-res}]$$

Fig. 8. Non-syntax directed rules for security analysis.

Typing judgements $\boxed{C \vdash \pi}$

$$\frac{\pi \in C}{C \vdash \pi}\,[\textit{c-in}] \qquad \frac{C \vdash \varphi_1 \sqsubseteq \varphi_2 \quad C \vdash \varphi_2 \sqsubseteq \varphi_3}{C \vdash \varphi_1 \sqsubseteq \varphi_3}\,[\textit{c-trans}] \qquad \frac{}{C \vdash \varphi \sqsubseteq \varphi}\,[\textit{c-reflex}]$$

$$\frac{}{C \vdash \bot \sqsubseteq \varphi}\,[\textit{c-bot}] \qquad \frac{}{C \vdash \varphi \sqsubseteq \top}\,[\textit{c-top}]$$

Fig. 9. Rules for constraints.

when applied to its two arguments gives identical annotated types. In Fig. 11 we present unification over types. We carefully maintain the invariant that a substitution that is obtained by unifying two subterms of some term is always applied to the remaining subterms of that term before these are unified. At the end, the substitutions obtained from the various subterms must be composed to obtain the substitution that is to be returned as the result. Unification of any type with a type variable results in a substitution of that variable with the given type. The type is required not to contain the type variable (unless the type is the same type variable), otherwise unification fails. Unification also fails whenever two different top level type constructors are unified.

$$
\begin{array}{ll}
\theta\ \mathtt{Int} & = \mathtt{Int} \\
\theta\ \mathtt{Bool} & = \mathtt{Bool} \\
\theta\ ({\tau_1}^{\varphi_1}, {\tau_2}^{\varphi_2}) & = ((\theta\ \tau_1)^{(\theta\ \varphi_1)}, (\theta\ \tau_2)^{(\theta\ \varphi_2)}) \\
\theta\ ({\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2}) & = (\theta\ \tau_1)^{(\theta\ \varphi_1)} \to (\theta\ \tau_2)^{(\theta\ \varphi_2)} \\
\theta\ (\mathtt{List}\tau^{\varphi}) & = \mathtt{List}(\theta\ \tau)^{(\theta\ \varphi)} \\
\theta\ \alpha & = \theta(\alpha)
\end{array}
$$

$$
\begin{array}{l}
\theta\ \beta = \theta(\beta) \\
\theta\ l \ \ = l
\end{array}
$$

$$
\begin{array}{ll}
\theta\ (\forall\alpha.\sigma) & = \forall\alpha.((\theta/\alpha)\ \sigma) \\
\theta\ (\forall\beta.\sigma) & = \forall\beta.((\theta/\beta)\ \sigma) \\
\quad \textbf{where} & \\
\qquad (\theta/\zeta)\ \zeta_1 = \zeta_1 \quad \textbf{if}\ \zeta \equiv \zeta_1 & \\
\qquad\qquad\qquad \theta\ \zeta_1\ \textbf{otherwise} &
\end{array}
$$

$$\theta\ (C \Rightarrow \tau) = (\theta\ C \Rightarrow \theta\ \tau)$$

$$
\begin{array}{l}
\theta\ \Gamma\ [x \mapsto (\tau, \varphi)] = (\theta\ \Gamma)\ [x \mapsto (\theta\ \tau, \theta\ \varphi)] \\
\theta\ [\,] = [\,]
\end{array}
$$

$$
\begin{array}{l}
\theta\ C = \{\theta\ \pi \mid \pi\ \in\ C\} \\
\theta\ (\varphi_1 \sqsubseteq \varphi_2) = (\theta\ \varphi_1 \sqsubseteq \theta\ \varphi_2)
\end{array}
$$

Fig. 10. Substitution application on types, annotations, type schemes, qualified types, type environments, constraint environments, and constraints.

$$
\begin{array}{llll}
\mathit{unifyTy} :: \textbf{Type} \to & & \textbf{Type} \to \textbf{Substitution} & \\
\mathit{unifyTy} & \mathtt{Int} & \mathtt{Int} & = \mathit{Id} \\
\mathit{unifyTy} & \mathtt{Bool} & \mathtt{Bool} & = \mathit{Id} \\
\mathit{unifyTy} & \mathtt{List}{\tau_1}^{\varphi_1} & \mathtt{List}{\tau_2}^{\varphi_2} & = \\
\quad \textbf{let} & \theta_1 = \mathit{unify}\ \tau_1\ \tau_2 & & \\
 & \theta_2 = \mathit{unify}\ (\theta_1\ \varphi_1)\ (\theta_1\ \varphi_2) & & \\
\quad \textbf{in} & \theta_2.\theta_1 & & \\
\mathit{unifyTy} & ({\tau_{11}}^{\varphi_{11}}, {\tau_{12}}^{\varphi_{12}}) & ({\tau_{21}}^{\varphi_{21}}, {\tau_{22}}^{\varphi_{22}}) & = \\
\quad \textbf{let} & \theta_1 = \mathit{unifyTy}\ \tau_{11}\ \tau_{21} & & \\
 & \theta_2 = \mathit{unifyAnn}\ (\theta_1\ \varphi_{11})\ (\theta_1\ \varphi_{21}) & & \\
 & \theta_3 = \mathit{unifyTy}\ (\theta_2.\theta_1\ \tau_{12})\ (\theta_2.\theta_1\ \tau_{22}) & & \\
 & \theta_4 = \mathit{unifyAnn}\ (\theta_3.\theta_2.\theta_1\ \varphi_{12})\ (\theta_3.\theta_2.\theta_1\ \varphi_{22}) & & \\
\quad \textbf{in} & \theta_4.\theta_3.\theta_2.\theta_1 & & \\
\mathit{unifyTy} & {\tau_{11}}^{\varphi_{11}} \to {\tau_{12}}^{\varphi_{12}} & {\tau_{21}}^{\varphi_{21}} \to {\tau_{22}}^{\varphi_{22}} & = \\
\quad \textbf{let} & \theta_1 = \mathit{unifyTy}\ \tau_{11}\ \tau_{21} & & \\
 & \theta_2 = \mathit{unifyAnn}\ (\theta_1\ \varphi_{11})\ (\theta_1\ \varphi_{21}) & & \\
 & \theta_3 = \mathit{unifyTy}\ (\theta_2.\theta_1\ \tau_{12})\ (\theta_2.\theta_1\ \tau_{22}) & & \\
 & \theta_4 = \mathit{unifyAnn}\ (\theta_3.\theta_2.\theta_1\ \varphi_{12})\ (\theta_3.\theta_2.\theta_1\ \varphi_{22}) & & \\
\quad \textbf{in} & \theta_4.\theta_2.\theta_2.\theta_1 & & \\
\mathit{unifyTy} & \tau & \alpha & = \textbf{if}\ (\tau \equiv \alpha\ \vee\ \alpha \notin (\mathit{ftv}\ \tau)) \\
 & & & \quad \textbf{then}\ [\alpha \mapsto \tau\,] \\
 & & & \quad \textbf{else}\ \mathit{fail} \\
\mathit{unifyTy} & \alpha & \tau & = \textbf{if}\ (\tau \equiv \alpha\ \vee\ \alpha \notin (\mathit{ftv}\ \tau)) \\
 & & & \quad \textbf{then}\ [\alpha \mapsto \tau\,] \\
 & & & \quad \textbf{else}\ \mathit{fail} \\
\mathit{unifyTy} & \_ & \_ & = \mathit{fail}
\end{array}
$$

Fig. 11. Type unification.

Our inference algorithm will always assign an annotation variable to an expression, and any restrictions on its security level will be expressed by separate constraints. This is why the following simple definition of the unification of security annotations suffices:

$$
\begin{array}{llll}
\mathit{unifyAnn} :: \textbf{Annotation} \to & \textbf{Annotation} \to & \textbf{Substitution} & \\
\mathit{unifyAnn} \quad \beta_1 & \beta_2 & = \ [\beta_1 \mapsto \beta_2\,] & \\
\mathit{unifyAnn} \quad \_ & \_ & = \mathit{fail} &
\end{array}
$$

Based on the logical deduction system presented in Section 4 we can define the security type inference algorithm $\mathcal{W}_{sec}$. Because the given type system is not yet syntax-directed, we implicitly make it syntax-directed by incorporating the rule of subeffecting, generalisation and instantiation into the syntax-directed rules (the syntax-directed variant is explicitly provided in [28]). Generalisation takes place in the implementation of the let-rule, to achieve let-polyvariance, and instantiation is performed in the rule for variables (by always choosing fresh variables for universally bound type and annotation variables). The non-syntax-directed rule [*t-sub*] can be handled by generating fresh annotation variables in various places, and relating these by constraints. For example, in the case for [*t-if*] we generate a fresh annotation variable for the annotated type for the conditional, say β, and relate the annotation variable on the **then** part, say β_1, by the constraint $\beta_1 \sqsubseteq \beta$; the **else** part and the conditional can be treated similarly.

The algorithm infers principal type-schemes, security annotations and constraints on security annotations. The algorithm is based on algorithm $\mathcal{W}$ [3]. The algorithm is defined by structural induction on e and has type:

$$\mathcal{W}_{sec} :: (\textbf{TyEnv}, \textbf{Exp}) \to (\textbf{Ty}, \textbf{Ann}, \textbf{Subst}, \textbf{Constraints})$$

The algorithm takes a pair of a type environment Γ and an expression e as an argument. The type environment Γ maps program variables to a pair consisting of a type-scheme and an annotation. The result of the algorithm is a quadruple containing a type τ, an annotation φ, a substitution θ and a set of constraints C. Subeffecting is handled through the constraints and verified by our constraint solver described below. The main algorithm is listed in Figs. 13, 14 and 15, again following the conventions of Chapter 5 of [21]. Below, we discuss some of the more interesting cases.

For the alternatives that type natural numbers and booleans, a fresh annotation variable is generated without any constraints, because there is no information that insists on a specific security level at this point. For the same reason, the empty list obtains the type $\mathtt{List}\,\alpha^{\beta_1}$ where again β_1 is fresh.

For the alternatives for function abstraction and recursive function abstraction, we introduce fresh type and annotation variables and associate these with the bound variable. While inferring the type of the body, more information about the bound variable's actual type may be discovered. This information is returned by means of the substitution. In the case of recursive functions the type of the recursive function also needs to be given a type, before we infer the type of the body. Later we must ensure that this type corresponds to the type found for the function; this is handled by unification.

For variables, the type scheme and annotation of the variable are retrieved from the environment. The type scheme is instantiated with *inst* from Fig. 12. The resulting qualified type is then divided into a type, and a constraint environment.

```
$inst\ (\forall\alpha_1 \ldots \alpha_n.\forall\beta_1 \ldots \beta_m.\rho) = [\alpha_1 \mapsto \alpha_1{}'] \ldots [\alpha_n \mapsto \alpha_n{}'] [\beta_1 \mapsto \beta_1'] \ldots [\beta_m \mapsto \beta_m'] \rho$
    $\textbf{where } \alpha_1{}' \ldots \alpha_n{}' \textit{ are fresh}$
        $\beta_1' \ldots \beta_m' \textit{ are fresh}$

$gen\ \Gamma\ \varphi\ \tau\ C =$
  $(\forall\alpha_1 \ldots \alpha_n.\forall\beta_1 \ldots \beta_m.C'' \Rightarrow \tau, C')$ $\textbf{where}$
    $(C', C'') = simplify\ \Gamma\ \tau\ C$
    $\{\alpha_1, \ldots, \alpha_n\} = ftv(\tau) - ftv(\Gamma)$
    $\{\beta_1, \ldots, \beta_m\} = fav(\tau) - fav(\Gamma) - fav(\varphi)$
```

Fig. 12. The instantiation function *inst* and the generalisation function *gen*.

$\boxed{\mathcal{W}_{sec}\ (\Gamma, e) = (\tau, \varphi, \theta, C)}$

```
$\mathcal{W}_{sec}\ (\Gamma, n) = \textbf{let } \beta \textit{ be fresh } \textbf{in } (\texttt{Int}, \beta, id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, True) = \textbf{let } \beta \textit{ be fresh } \textbf{in } (\texttt{Bool}, \beta, id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, False) = \textbf{let } \beta \textit{ be fresh } \textbf{in } (\texttt{Bool}, \beta, id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{Nil}) = \textbf{let } \alpha, \beta, \beta_1 \textit{ be fresh } \textbf{in } (\texttt{List}\ \alpha^{\beta_1}, \beta, id, \emptyset)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{Cons}\ e_1\ e_2) =$
  $\textbf{let } \beta, \beta_1 \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\theta_3 = \mathcal{U}\ (\tau_2, \theta_2\ (\texttt{List}\ \tau_1{}^{\varphi_1}))$
    $\theta = \theta_3.\theta_2.\theta_1$
  $\textbf{in } (\theta\ (\texttt{List}\ \tau_1{}^{\beta_1}), \beta, \theta, \theta\ (C_1$
      $\cup\ C_2 \cup \{\varphi_1 \sqsubseteq \beta_1, \varphi_2 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, (e_1, e_2)) =$
  $\textbf{let } \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\theta = \theta_2.\theta_1$
  $\textbf{in } (\theta\ (\tau_1{}^{\varphi_1}, \tau_2{}^{\varphi_2}), \beta, \theta, \theta\ (C_1 \cup C_2))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{fn}\ x \Rightarrow e_0) =$
  $\textbf{let } \alpha_x, \beta_x, \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma\ [x \mapsto (\alpha_x, \beta_x)], e_0)$
  $\textbf{in } (\theta_1\ (\alpha_x{}^{\beta_x} \to \tau_1{}^{\varphi_1}), \beta, \theta_1, C_1)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{fun}\ f\ x \Rightarrow e_0) =$
  $\textbf{let } \alpha_x, \alpha_r, \beta_x, \beta_r, \beta \textit{ be fresh}$
    $(\tau_0, \varphi_0, \theta_0, C_0) =$
      $\mathcal{W}_{sec}\ (\Gamma\ [f \mapsto (\alpha_x{}^{\beta_x} \to \alpha_r{}^{\beta_r}, \beta)]\ [x \mapsto (\alpha_x, \beta_x)], e_0)$
    $\theta_1 = \mathcal{U}\ (\tau_0, \theta_0\ \alpha_r)$
    $\theta_2 = \mathcal{U}\ (\theta_1\ \varphi_0, (\theta_1.\theta_0)\ \beta_r)$
    $\theta = \theta_2.\theta_1.\theta_0$
  $\textbf{in } (\theta\ (\alpha_x{}^{\beta_x} \to \tau_0{}^{\varphi_0}), (\theta_2.\theta_1)\ \beta, \theta, \theta\ C_0)$
$\mathcal{W}_{sec}\ (\Gamma, x) =$
  $\textbf{let } \beta_1 \textit{ be fresh}$
    $(\sigma, \varphi) = \Gamma\ (x)$
    $C' \Rightarrow \tau = inst\ (\sigma)$
  $\textbf{in } (\tau, \beta_1, id, C' \cup \{\varphi \sqsubseteq \beta_1\})$
$\mathcal{W}_{sec}\ (\Gamma, e_1\ e_2) =$
  $\textbf{let } \alpha_r\ \beta_r\ \beta_x\ \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\theta_3 = \mathcal{U}\ (\theta_2\ \tau_1, \tau_2{}^{\beta_x} \to \alpha_r{}^{\beta_r})$
    $\theta = \theta_3.\theta_2.\theta_1$
    $C = C_1 \cup C_2 \cup \{\varphi_2 \sqsubseteq \beta_x, \varphi_1 \sqsubseteq \beta, \beta_r \sqsubseteq \beta\}$
  $\textbf{in } (\theta_3\ \alpha_r, \beta, \theta, \theta\ C)$
```

Fig. 13. Type and security analysis inference algorithm.

$\boxed{\mathcal{W}_{sec}\ (\Gamma, e) = (\tau, \varphi, \theta, C)}$

```
$\mathcal{W}_{sec}\ (\Gamma, \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) =$
  $\textbf{let } \beta \textit{ be fresh}$
    $(\tau_0, \varphi_0, \theta_0, C_0) = \mathcal{W}_{sec}\ (\Gamma, e_0)$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\theta_0\ \Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ ((\theta_1.\theta_0)\ \Gamma, e_2)$
    $\theta_3 = \mathcal{U}\ ((\theta_2.\theta_1)\ \tau_0, \texttt{Bool})$
    $\theta_4 = \mathcal{U}\ (\theta_3\ \tau_2, (\theta_3.\theta_2)\ \tau_1)$
    $\theta = \theta_4.\theta_3.\theta_2.\theta_1.\theta_0$
    $C = C_0 \cup C_1 \cup C_2$
      $\cup\ \{\varphi_0 \sqsubseteq \beta, \varphi_1 \sqsubseteq \beta, \varphi_2 \sqsubseteq \beta\}$
  $\textbf{in } (\theta\ \tau_2, \beta, \theta, \theta\ C)$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2) =$
  $\textbf{let } \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\sigma, C') = gen\ (\theta_1\ \Gamma, \varphi_1, \tau_1, C_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma\ [x \mapsto (\sigma, \varphi_1)], e_2)$
  $\textbf{in } (\tau_2, \beta, \theta_2.\theta_1, \theta_2\ (C' \cup C_2 \cup \{\varphi_2 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, e_1 \oplus e_2) =$
  $\textbf{let } \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $(\tau_2, \varphi_2, \theta_2, C_2) = \mathcal{W}_{sec}\ (\theta_1\ \Gamma, e_2)$
    $\tau_\oplus^1 \to \tau_\oplus^2 \to \tau_\oplus = \Gamma_\oplus\ (\oplus)$
    $\theta_3 = \mathcal{U}\ (\theta_2\ \tau_1, \tau_\oplus^1)$
    $\theta_4 = \mathcal{U}\ (\theta_3\ \tau_2, \tau_\oplus^2)$
    $\theta = \theta_4.\theta_3.\theta_2.\theta_1$
  $\textbf{in } (\tau_\oplus, \beta, \theta, \theta\ (C_1 \cup C_2 \cup \{\varphi_1 \sqsubseteq \beta, \varphi_2 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, u\ e_1) =$
  $\textbf{let } \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\tau_u^1 \to \tau_u = \Gamma_\oplus\ (u)$
    $\theta_2 = \mathcal{U}\ (\tau_1, \tau_u^1)$
  $\textbf{in } (\tau_u, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{fst}\ e_1) =$
  $\textbf{let } \alpha_1, \beta_1, \alpha_2, \beta_2, \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\theta_2 = \mathcal{U}\ (\tau_1, (\alpha_1{}^{\beta_1}, \alpha_2{}^{\beta_2}))$
  $\textbf{in } (\theta_2\ \alpha_1, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta, \beta_1 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{snd}\ e_1) =$
  $\textbf{let } \alpha_1, \beta_1, \alpha_2, \beta_2, \beta \textit{ be fresh}$
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\theta_2 = \mathcal{U}\ (\tau_1, (\alpha_1{}^{\beta_1}, \alpha_2{}^{\beta_2}))$
  $\textbf{in } (\theta_2\ \alpha_2, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta, \beta_2 \sqsubseteq \beta\}))$
```

Fig. 14. Type and security analysis inference algorithm (continued).

To maintain the invariant of never assigning a security level to an expression we introduce a fresh annotation variable and restrict it to be at least as secure as the security level from the type environment.

For function application, the type of e_2 has to match the type expected by the function. We first obtain types for e_1 and e_2, and then enforce that the former type is a function type. Only then can we unify the type found for e_2 with the argument type of this function type. We introduce fresh variables for the result type and annotation as well as for the argument annotation to implement subeffecting at this point.

The alternative for let-bindings deserves a bit more attention as this is the place where generalisation takes place. First the type of e_1 is inferred. To obtain a generalised type (and a set of remaining constraints), we apply the function *gen* given in Fig. 12. Because our language is let-polyvariant, we need to simplify/solve the constraints as we compute the annotated type scheme of a let-bound identifier. This is necessary to establish over which annotation variables we should quantify. This

$$\boxed{\mathcal{W}_{sec}\ (\Gamma, e) = (\tau, \varphi, \theta, C)}$$

```
$\mathcal{W}_{sec}\ (\Gamma, \textbf{null}\ e_1) =$
  let $\alpha_1, \beta_1, \beta$ be fresh
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\theta_2 = \mathcal{U}\ (\tau_1, \mathtt{List}\ {\alpha_1}^{\beta_1})$
  in $(\mathtt{Bool}, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{hd}\ e_1) =$
  let $\alpha_1, \beta_1, \beta$ be fresh
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\theta_2 = \mathcal{U}\ (\tau_1, \mathtt{List}\ {\alpha_1}^{\beta_1})$
  in $(\theta_2\ \alpha_1, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta, \beta_1 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{tl}\ e_1) =$
  let $\alpha_1, \beta_1, \beta$ be fresh
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e_1)$
    $\theta_2 = \mathcal{U}\ (\tau_1, \mathtt{List}\ {\alpha_1}^{\beta_1})$
  in $(\theta_2\ \tau_1, \beta, \theta_2.\theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\}))$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{declassify}\ e\ \varphi) =$
  let $\beta$ be fresh
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e)$
  in $(\tau_1, \beta, \theta_1, C_1 \cup \{\varphi \sqsubseteq \varphi_1, \varphi \sqsubseteq \beta\})$
$\mathcal{W}_{sec}\ (\Gamma, \textbf{protect}\ e\ \varphi) =$
  let $\beta$ be fresh
    $(\tau_1, \varphi_1, \theta_1, C_1) = \mathcal{W}_{sec}\ (\Gamma, e)$
  in $(\tau_1, \beta, \theta_1, C_1 \cup \{\varphi_1 \sqsubseteq \varphi, \varphi \sqsubseteq \beta\})$
```

Fig. 15. Type and security analysis inference algorithm (continued).

```
$range(C) =$ do
  $worklist$ :: Constraints
  $worklist \quad = \{\ \}$
  $A$ :: AnnVar $\rightarrow$ (Levels, Levels)
  $A \quad = [\ ]$
  $ub$ :: AnnVar $\rightarrow$ {Constr}
  $ub \quad = [\ ]$
  $lb$ :: AnnVar $\rightarrow$ {Constr}
  $lb \quad = [\ ]$
  for all $\pi$ in $C$ do $worklist := worklist \cup \{\pi\}$
  for all $\beta$ in $fav(C)$ do
    $A \ = A[\beta \mapsto (\bot, \top)]$
    $ub = ub[\beta \mapsto \{\ \}]$
    $lb \ = lb[\beta \mapsto \{\ \}]$
  for all $\pi @ (\beta_1, \beta_2)$ in $C$ do
    $ub[\beta_2] = ub[\beta_2] \cup \{\pi\}$
    $lb[\beta_1] \ = lb[\beta_1] \cup \{\pi\}$
  while $worklist \neq \{\ \}$ do
    let $C_1 \cup \{\pi\} = worklist$
    $worklist = C_1$
    case $\pi$ of
      $(\beta_1 \sqsubseteq \beta_2) \Rightarrow$
        if $\lceil A\ [\beta_1] \rceil \neq \lceil A\ [\beta_1] \rceil \sqcap \lceil A\ [\beta_2] \rceil$) then
          $worklist = worklist \cup ub[\beta_1]$
          $\lceil A\ [\beta_1] \rceil = \lceil A\ [\beta_1] \rceil \sqcap \lceil A\ [\beta_2] \rceil$
        if $\lfloor A\ [\beta_2] \rfloor \neq \lfloor A\ [\beta_2] \rfloor \sqcup \lfloor A\ [\beta_1] \rfloor$) then
          $worklist = worklist \cup lb[\beta_2]$
          $\lfloor A\ [\beta_2] \rfloor = \lfloor A\ [\beta_2] \rfloor \sqcup \lfloor A\ [\beta_1] \rfloor$
      $(l \sqsubseteq \beta) \Rightarrow$
        if $(\lfloor A\ [\beta] \rfloor \neq \lfloor A\ [\beta] \rfloor \sqcup l)$ then
          $worklist = worklist \cup lb[\beta]$
          $\lfloor A\ [\beta] \rfloor = \lfloor A\ [\beta] \rfloor \sqcup l$
      $(\beta \sqsubseteq l) \Rightarrow$
        if $(\lceil A\ [\beta] \rceil \neq \lceil A\ [\beta] \rceil \sqcap l)$ then
          $worklist = worklist \cup ub[\beta]$
          $\lceil A\ [\beta] \rceil = \lceil A\ [\beta] \rceil \sqcap l$
  return $A$
```

Fig. 16. Worklist algorithm for range computation.

explains the use of *simplify* in the definition of *gen*. The task of *simplify* is to decide whether the constraint set is consistent, and, if this is the case, remove trivially satisfied constraints, and return a partition (C', C'') of the remaining constraints. The latter contains constraints that involve annotation variables that we generalise over, and that we therefore need to store in the type scheme. We quantify universally over the type and annotation variables that occur free in τ. The other constraints, in C', are returned for further propagation. The function *simplify* is implemented by a worklist algorithm, and is discussed below. We can then proceed to infer the type of the body of the let e_2, after adding the information we have just found for e_1 to the type environment.

The alternatives for **declassify** $e\ \varphi$ and **protect** $e\ \varphi$ are relatively simple, but crucial for this particular analysis. In both cases, the type of the whole expression is the inferred type for e, and its security level β is at least φ. The security level inferred for e, φ_1, has to be at least as secure as φ in the case of declassify, and φ has to be at least as secure as φ_1 in the case of protect. These requirements are enforced by constraints.

To conclude this section, we discuss the details of the function *simplify* that is employed in the case of the let-rule. The function first determines the set of variables that should not be instantiated. It then computes the range of allowed security levels for each annotation variable, by means of the worklist algorithm *range* detailed in Fig. 16. Then it verifies whether the constraint set is satisfiable at all. The satisfiability check results in a substitution that maps the variables that may be instantiated to their lowest possible security level. Finally, the constraint set is partitioned, for reasons described earlier. We next describe these steps in more detail below.

Our security analysis is polyvariant. This that as we compute the type of some identifier, say x, we should establish not only its polymorphic type (which is what algorithm W by itself would do), but also to establish what the security annotations on its type should be, and, in particular, which of the security annotation variables that occur in the type scheme for x may be generalised over. Consider, for example, that we have analysed a function f for which we obtain a polymorphic type $\alpha \rightarrow \alpha \rightarrow \alpha$, and in which we have associated with the first α a security annotation variable β_1, with the second β_2 and the result β_3. The constraint set obtained from the definition may, for example, be $\beta_1 \sqsubseteq \beta_3$ and $\beta_2 \sqsubseteq \beta_3$, and these constraints need to be stored in the annotation type scheme to be verified at each instantiation. Since we have to perform this check at each instantiation, it makes sense to simplify these constraint as much as possible, so that constraints that arise from the definition of f and that either are trivially satisfied, or occur more than once, or known to be satisfiable whatever choice we make for β_1 and β_2 at instantiation, can be safely omitted from the annotated type

```
$simplify(\Gamma, \tau, C) =$ do
  $active = pseudoActive\ (fav\ (\Gamma) \cup fav\ (\tau), C)$
  $ranges = range\ C$
  $\theta = satisfiable\ (active, ranges)$
  $c' = \theta\ C$
  return $partition(\Gamma, c')$
```

Fig. 17. Simplification algorithm.

```
$pseudoActive\ (active, C) =$ do
  $vars = \{\beta_2 \mid (\beta_1 \sqsubseteq \beta_2) \in C \wedge \beta_1 \in active\} \cup active$
  if ($vars$ neq $active$) then
    $pseudoActive\ (vars, C)$
  else
    return $vars$
```

Fig. 18. Determine set of pseudo active variables.

```
$satisfiable\ (active, analysis) =$ do
  $substitution = Id$

  for all $[\beta \mapsto (l, l')]$ in $analysis$ do
    if $(l \sqsubseteq l')$ then
      if $(\beta \notin active \vee l \equiv l')$ then
        $substitution = [\beta \mapsto l].substitution$
      else skip
    else error
  return substitution
```

Fig. 19. Checking satisfiability.

```
$partition\ (\Gamma, C) =$ do
  $active = fav(\Gamma)$
  $prop = \{(\beta_1 \sqsubseteq \beta_2) \in C \mid (\beta_1 \in active \vee \beta_2 \in active)\}$
  $qual = \{\pi \in C \mid \pi \notin prop\}$
  return $(prop, qual)$
```

Fig. 20. Partitioning of constraints.

scheme. For example, consider that the constraints found for f are not $\beta_1 \sqsubseteq \beta_3$ and $\beta_2 \sqsubseteq \beta_3$, but $\beta_1 \sqsubseteq \beta_4$, $\beta_4 \sqsubseteq \beta_3$ and $\beta_2 \sqsubseteq \beta_3$. The annotation variable β_4 does not occur in the type of f, it is an intermediary introduced by the algorithm (for whatever reason). A crucial thing to realise here is that whatever we choose for β_1 and β_3 that satisfies $\beta_1 \sqsubseteq \beta_3$, we can always default β_4 to the chosen value of β_1, so that also the original two constraints $\beta_1 \sqsubseteq \beta_4$ and $\beta_4 \sqsubseteq \beta_3$ are satisfied. This means that when it comes to storing constraints in the type scheme for f, we can forget about β_4 and the constraints imposed on it, as long we remember to store the implied constraint $\beta_1 \sqsubseteq \beta_3$ in the annotated type signature. This process of simplification, and while simplifying deciding which constraints must be stored in the type scheme, and which do not, is the goal of the function *simplify*. To decide which constraints go where in this partitioning, we need to examine the constraints in some detail anyway, which is why we have integrated this with the process of simplification (see Fig. 17).

We call the annotation variables that occur free in the type τ that is being generalised, and the annotation variables that occur free in the type environment Γ the *active variables*, i.e. $fav\ (\tau) \cup fav\ (\Gamma)$. This set of active variables is, however, only a first approximation: some variables do not occur in the type τ, but do depend on the active variables. For example, in the constraint $\beta_1 \sqsubseteq \beta_2$ the variable β_2 is restricted to be at least as secure as β_1. If β_1 is an active variable it will not be instantiated, and therefore it is not clear what the lowest possible security level for β_2 is. We therefore delay instantiation of β_2 until β_1 is known. Variables such as β_2 are known as *pseudo active variables*. In Fig. 18 we present a function that computes the set of all pseudo active variables (which includes also the active variables), given the set of all active variables and a set of constraints.

Any variable that is not pseudo active may be instantiated, if there exists a non-empty range of security levels for each such variable. If, for any variable, it is not possible to choose a security level such that all constraints are satisfied, then the program contains an error. In that case the type inference process will explain where the problem arises and what caused it (this is discussed in Section 6). The lowest and highest allowed security level for each variable is computed by a work list algorithm. We present this algorithm in Fig. 16. The algorithm is based on the work list algorithm presented in Chapter 3 of [21]. It has been extended to deal with the fact that some variables are pseudo active and may therefore not be touched, and computes two arrays *ub* and *lb*. Compared to [21], we use two instead of one array as we compute both a lower bound and an upper bound for each variable.

The array *ub* is used to propagate more precise security upper bounds downward. The set $ub\,[\beta]$ contains all constraints in which β is the upper bound. The *lb* array does exactly the opposite. The result for all variables that occur in the constraint set C is initialised with the range $(\bot, \top)$. After initialisation the algorithm will compute the actual ranges by taking constraints from the work list and updating the bounds of variables accordingly. A variable β that had its lower bound changed will add all constraints in the set $lb\,[\beta]$ to the work list. A variable β that had its upper bound changed adds all constraints from the set $ub\,[\beta]$ to the work list. In the algorithm we use the following notational conventions: $\lfloor A\ [\beta] \rfloor$ denotes the current lower bound of the range associated with β, and $\lceil A\ [\beta] \rceil$ returns the upper bound of the range associated with β. The least upper bound operator is written as $\sqcup$ and greatest lower bound operator as $\sqcap$.

The outcome of the work list algorithm is a mapping of annotation variables to a range of security levels. The function *satisfiable* (presented in Fig. 19) processes the results of the work list algorithm. If any variable has an empty range (the upper bound is then strictly under the lower bound), then an error has been found and error diagnosis starts. Otherwise, we compute the substitution that replaces all non-pseudo active variable by the lower bound of the range.

Partitioning the constraint set is the final step of the simplification process. The constraint set is partitioned into two sets, for reasons explained earlier. The partition function is given in Fig. 20.

6. Heuristics

This section discusses the various heuristics we have developed, organized into four different categories: generic heuristics, propagation heuristics, dependency analysis specific heuristics, and security specific heuristics. We motivate and de-

$one = \textbf{protect}\ 1\ Low$
$two = \textbf{protect}\ 2\ Low$
$three = \textbf{protect}\ 3\ Low$
$four = \textbf{protect}\ 4\ Low$
$five = \textbf{protect}\ 5\ High$
$fifteen = print\ (one + two + three + four + five)$

Fig. 21. Example program: majority of Low values in faulty subexpression.

$secureVal :: \mathtt{Int}^{High}$
$incr :: \forall\beta_1.\forall\beta_2.\beta_1 \sqsubseteq \beta_2 \Rightarrow \mathtt{Int}^{\beta_1} \to \mathtt{Int}^{\beta_2}$
$id :: \forall\alpha.\forall\beta_1.\forall\beta_2.\beta_1 \sqsubseteq \beta_2 \Rightarrow \alpha^{\beta_1} \to \alpha^{\beta_2}$
$print :: \forall\alpha.\alpha^{Low} \to \alpha^{Low}$

$print\ (incr\ (incr\ (id\ secureVal)))$

Fig. 22. Propagating security levels through *incr* and *id*.

scribe the heuristics themselves, and define the order in which they are applied, and why. In Section 7 we provide additional examples. We note that whatever the heuristics do, they will *never* remove all constraints from the current set. This would imply that no constraint can be blamed for the inconsistency.

We note that having had to refashion some of our code to fit the columns of the paper, the location information may not be correct in all cases.

6.1. Generic heuristics

The heuristics in this section are generally applicable heuristics that have also been employed in other work on heuristics-based type error diagnosis.

Majority heuristic

Johnson and Walz introduced the idea to look at the amount of evidence for a constraint to be the source of an inconsistency [15]. The majority heuristic retrieves all constraints from an expression that is used as an argument but is considered to be too secure. Then it computes for each security level the number of constraints that imply that the expression should at least have that security level. If the number of constraints that testify that the expression should have a lower security level is substantially larger than the amount of constraints demanding a higher security level, then the subexpressions where the latter were generated might be the actual cause of the inconsistency. As a result, a mention of those subexpressions will be added to the type error message, stating that they caused the security level to be so high. Note, that we do *not* propose that these expressions are at fault.

In Fig. 21, the first five lines declare four values protected at level *Low* and one protected at level *High*. The declaration *fifteen* on the sixth line computes the sum of these values and passes the result to *print*. The latter expects a value that is protected at level *Low*, but the sum of the five values is protected at level *High*. The only value that causes the sum to be protected at level *High* is *five*, all four other values are protected at level *Low*. The heuristic blames the application of *print*. As there is very little evidence stating that the sum should be protected at level *High* the heuristic also explains what caused the expression to be protected at this level:

```
Error in application:
  "(print ((((one + two) + three) + four) + five))" at: (line 6, column 12)
Expected an argument protected at most level: Low
The argument is protected at level: High
Because of the following subexpression(s): "five" at: (line 6, column 46)
```

The least trusted constraint

All constraints are assigned a certain amount of trust based on the AST-node they are generated at. We believe that there is a good reason to have more trust in certain programming constructs than others, because some constructs are more often the cause of an inconsistency or are less intuitive. Constraints that result from instantiating the type of a program variable defined elsewhere receive a higher trust value than constraints that were generated for the expression itself. The constraints that are generated at these sites belong to the declaration of that particular variable and were found to be consistent when generalising the type of that program variable. Constraints that are generated at application sites receive the least amount of trust, because this construct is considered most likely to introduce inconsistencies. In Section 7 we discuss an example of this kind. This heuristic was introduced by Heeren in Section 8.4 of [13].

Irrefutable constraints

At some nodes of the abstract syntax tree we generate a constraint for reasons of uniformity with the rest of the type system. We know that such constraints can never be wrong, and therefore should never be blamed. The irrefutable constraints heuristic removes these constraints from the current set of constraints. This is achieved by setting the trust for such constraints to infinity. Our type system, for example, generates a constraint at let bindings stating that the let binding is at least as protected as the body of the let binding. This constraint is, for obvious reasons, always true, and should never be blamed. This heuristic was introduced by Heeren in Section 8.3 of [13].

6.2. The propagation heuristic

Many of the generated constraints only propagate security levels. Consider for example the program in Fig. 22. The highly secure *secureVal* is passed through the identity function once and then twice in succession through *incr* before being passed to the print function. Both functions, *incr* and *id*, are polyvariant and propagate the security annotation from their argument to their result. An algorithm will generate explicit constraints to, step-by-step, propagate the security level of *secureVal* to the *print* function. Since the *print* function expects a value of low confidentiality, the program is inconsistent. When we want to assign blame, it makes no sense to blame the application of functions like *id* and *incr*, because they do not affect the security levels of their argument. What we want then is that the constraints responsible for the propagation of security levels are never blamed for an inconsistency. Therefore we have devised a heuristic that will remove such constraints from the current constraint set. Of course, an error message could suggest to replace a function like *id* by a function that does change the security properties (like **declassify** in this particular case). But since there is no way that we can decide which function should be replaced and with what it should be replaced, this can only serve to confuse the programmer. We will thus have to accept that the sequence of calls to security agnostic functions is correct, which is attained by deleting all propagation constraints from the minimal unsatisfiable constraint set. Note that the usefulness of this heuristic stems from the polyvariance of the analysis, and is independent of the fact that the underlying language is polymorphic or monomorphic.

6.3. Heuristics for dependency analyses

Security analysis is an instance of a dependency analysis [2], which makes some of the type rules that govern security annotations slightly and subtly different from those of the intrinsic type system that the programmer will most likely be used to. In this section we describe a few heuristics that are constructed with this in mind. For example, the heuristic for the conditional considers that a programmer may believe that the security level of the condition does not contribute to that of the whole conditional, although the type rule in Fig. 8 says otherwise. The heuristic tries to discover whether such a misunderstanding explains an inconsistency among the constraints.

We have systematically compared the constraints on security annotations and the constraints on the underlying types in each of the rules of Fig. 8. For the discrepancies we have found, we have implemented a corresponding heuristic, except for the cases of **fst** and **snd** (the reason is given below). We list all discrepancies in the following paragraphs, but in the interest of conciseness, we only provide details and examples for the heuristic for the conditional.

The first discrepancy is that the security level of an application may be influenced by the security level on the *function itself*. Specifically, in the rule [*t-app*], the φ on the arrow of the type of e_1 contributes to the security level of the application $e_1\ e_2$. In the underlying type system, only the result type of the function type contributes to the type of $e_1\ e_2$. We note that functions are usually created at the lowest security level, but it is still possible to increase the security level by protecting it at a higher level.

The second discrepancy can be found in the rule [*t-hd*]. In the underlying type system, only the element type of the list determines the type of the result of **hd**, but since successfully applying **hd** to a list also provides some information about the structure of its argument list (it is not empty), the security level φ attached to the structure of the list also contributes to the security level of the result of **hd**. We note that in the case of **fst** and **snd** the same reasoning applies, but that the annotations on pairs are there for reasons of uniformity of presentation only. However, it is possible to explicitly increase the security level on a pair by using protect, which will then indeed influence the security level on the result of **fst** and **snd**. It is easy to add such heuristics to our prototype, but currently these have not been implemented.

The final and arguably most striking discrepancy arises for the conditional statement. As explained in the introduction, the outcome of a conditional may reveal information about the value of the condition, and therefore if the condition is highly secure, the result of the conditional will be highly secure, even if the then and else parts themselves are not as secure. This fact can be gleaned from the rule [*t-if*], in which the annotation on e_0 contributes to the security annotation on the conditional, but that only the type τ of the then and else parts determine the type of the conditional.

The heuristic determines whether the security level of the condition is the reason that the whole expression is protected at a level higher than expected. If so it will report this to the programmer and will also explain why the conditional expression has a high security level. In Fig. 23 we present a program where secure information is leaked through the conditional. The following error message is generated by our heuristic:

```
The  conditional: hVal of the if statement:
  if hVal then lVal else mVal    at (l 9, c 14)
uses a value: hVal protected at level: High.
This causes the whole if expression:
  if hVal then lVal else mVal    at (l 9, c 14)
to be protected at level: High.
Instead a value protected at level: Medium was expected by: log.
```

Note that the example uses a lattice consisting of values Low, Medium and High, with the expected relations $Low \sqsubseteq Medium$ and $Medium \sqsubseteq High$. The message describes why the conditional is protected at level *High*, and that this is inconsistent with the security level expected by the function *log*.

6.4. Security specific heuristics

The sFun++ language has two constructs to explicitly change the security level, **declassify** and **protect**. It is not unlikely that a programmer, at first, will confuse the two. For example, he may try to declassify an expression *e* by using the **protect**

$log = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Medium$
$hVal = \textbf{protect}\ True\ High$
$mVal = \textbf{protect}\ 1\ Medium$
$lVal = \textbf{protect}\ 2\ Low$
$error = log\ (\textbf{if}\ hVal\ \textbf{then}\ lVal\ \textbf{else}\ mVal)$

Fig. 23. Secure information leaks through conditional.

$secureValue = \textbf{protect}\ True\ High$
$printSecure = printValue\ (\textbf{protect}\ secureValue\ Low)$

Fig. 24. Example program: misuse of protect statement.

statement, and provide a security level lower than the level inferred for *e*, or vice versa. In this section we describe in more detail the situation that a **protect** may need to be replaced by **declassify**. The inverse situation follows dually, and we will not discuss it further. Both have been implemented in the prototype.

In Fig. 24 we present an expression that contains a mistake. The value *secureValue* has type $\texttt{Bool}^{High}$ and the function *printValue* prints a value that is protected at level *Low*. The programmer of the program tried to declassify *secureValue* by protecting it at level *Low*. Our heuristic will generate the following error message

```
You try to protect the expression: "protect secureValue Low" at (line 3, column 27) to level: Low
But the expression you are protecting is protected at level: High.
Try declassify to assign the expression the level: Low
```

The message suggests to replace **protect** with **declassify**, and this indeed results in a security correct program.

It may seem that this heuristic can always be applied whenever we have an inconsistency involving **protect**, but that is not the case: it is essential that the level explicitly provided is *lower* than the inferred level, because that provides the additional hint to the system that **declassify** was intended. For example, the same heuristic will not suggest to replace **protect** 1 *Medium* by **declassify** 1 *Low* for the error in the code in Fig. 23.

The heuristic requires that there are currently only two constraints in the constraint set, each containing one non-variable annotation. It will then consider the nodes where the constraints originate from, and if this includes a **protect** node, and the explicitly provided level is lower than the inferred security level of the argument to **protect**, then the heuristic will be applicable and generate a suitable error message.

We note that these heuristics are instances of the sibling heuristic introduced by Heeren et al. [12].

6.5. How heuristics are applied

In our prototype heuristics are applied in sequence, and in a particular order. One can easily come up with various common-sense reasons why a given heuristic should be tried after another. For example, it makes sense to first filter out irrefutable constraints, and to keep the less-focused heuristics to later. As a rule, we prefer to try the heuristics that pinpoint a particular often-made mistake early on. In general, however, the "best" order very much depends on programmer preference (although we have not investigated this, we can imagine that this can be attained by some adaptive algorithm). This is why we made it easy to change the order in which heuristics are considered in our prototype (to be precise, the identifier *heuristics* defined in module *Heuristics* / *Heuristics.hs*). For the examples in this paper, we have used the following ordering that corresponds to the ordering in our downloadable prototype:

1. remove irrefutable constraints,
2. select constraints with heuristics for dependency analyses (if, head, application) (we omitted heuristics for fst and snd, because they are not so likely to be useful)
3. remove propagation constraints,
4. security specific heuristics,
5. select constraint based on majority heuristic,
6. select least trusted constraint,
7. pick among the remainder, the constraint that is "earliest" in the program (first come, first blamed)

The motivation for this particular order is as follows: we first delete all irrefutable constraints, because we know that blaming such a constraint will seem very silly. It makes sense to remove propagation constraints at this point, because blaming any of these will not make sense either. In our implementation, however, it is much easier to delete such constraints *after* we know that the dependency-analysis based heuristics are known not to apply. Having removed the propagation constraints, we move on to the last of the specific heuristics, that specifically targets **declassify** or **protect** invocations in the program. The remaining heuristics are more generic, starting with the heuristic that targets specific constraints to blame, followed by weaker heuristics that filter out constraints that are less likely to be to blame.

On the occasion that after applying heuristics 1 through 6 we have not yet found a single cause of the error, it is still possible to present a program slice. Since some of the heuristics will have filtered out various constraints, such a slice is typically much smaller than the original minimal unsatisfiable set of constraints. The program presented in Fig. 21 would, for example, result in the following slice

```
Sort of error: use of secure value as less secure argument, endpoints Low vs. High
  print ((..) + five)
```

```
passwordFile =
  Cons (1, protect 31415 High)
    (Cons (2, protect 27182 High) Nil)
findUser =
  fun f user ⇒
    fn l ⇒ if (null l) then Nil
      else
        let r = hd l
        in if ((fst r) ≡ user)
          then Cons r Nil
          else f user (tl l)
```

Fig. 25. Auxiliary code to go with the login function.

```
login =
  fn u ⇒
    fn p ⇒ print (declassify
      (let userRecord = (findUser u passwordFile)
        in (if (null userRecord)
          then False
          else ((snd (hd userRecord)) ≡ p))) Low)
```

Fig. 26. The correct login function.

```
login =
  fn u ⇒
    fn p ⇒ print
      (let userRecord = (findUser u passwordFile)
        in (if (null userRecord)
          then False
          else ((snd (hd userRecord)) ≡ p)))
```

Fig. 27. The login function without declassification.

```
login =
  fn u ⇒
    fn p ⇒ print (protect
      (let userRecord = (findUser u passwordFile)
        in (if (null userRecord)
          then False
          else ((snd (hd userRecord)) ≡ p))) Low)
```

Fig. 28. The login function with protection.

which indeed only shows those parts of the program that contribute to the inconsistency. The explanation of the slice, on the first line, is determined by the kind of AST-node that forms the root of the slice. The endpoints refer to the expected and the provided security levels. Thus, when we are not able to find the cause of the inconsistency, presenting the program slice may still provide a useful error message. Our prototype implementation cannot present program slices, but it can list all program points that contribute to an inconsistency. We believe that it is possible to construct a type error slice (those parts of the program that contribute to the error leaving out those that do not) from this information. In our implementation we resort to a catch-all heuristic (no. 7) that picks from the remaining constraint the one that occurs earliest in the program, and bases the error report on that constraint.

7. Further examples

Validation of our work is best performed on a large corpus of programs, following the example of [19] for such a study on type error diagnosis for ML. Unfortunately security facilities have not found their way yet to mainstream functional languages, which is why security incorrect programs are hard to come by. An exception may be the Jif system [18], but that is a Java based system, a context in which we have no polyvariance and no higher-order functions (see Section 8 for a more detailed discussion). We therefore resort to discussing a number of examples, and variations thereof. Our implementation (see the introduction) provides additional example programs.

In Figs. 25 and 26 we present a security type correct login program written in sFun++, similar to the FlowCaml program presented in Section 2. In the program user names and passwords are simple integers, as our language does not support characters and strings. A password file is an (unprotected) list of pairs, consisting of an unprotected user name and a protected password. The function *findUser* retrieves the user name from the list. Because we do not have a proper *Maybe* type, we return either a singleton list when the user is found, or the empty list when the requested user name is not known. The function *login* receives a user name and a password as arguments; it then looks up the user record in the password file. If a user record is found it compares the password inside with the given password. If no user was found it returns *False*. The result of the password comparison is protected at security level *High*, the *print* function expects a value that is protected no higher than *Low*. It is therefore declassified before it is printed.

In Fig. 27 we present a variation of the login function where declassification is forgotten. This means that a highly secure boolean value is passed to the *print* function. The corresponding error message is the following:

```
Error in application:
"(print let userRecord = ((findUser u) passwordFile) in if null userRecord
 then False else (snd head userRecord == p))" at: (line 12, column 25)
The function: "print"
Expected an argument protected at most level: Low
But the argument: "let userRecord ... == p)"
Is protected at a higher level.
```

In this case, it is the least trusted constraint heuristic that correctly blames the application.

A novice programmer may at first confuse **declassify** and **protect**. In Fig. 28 the login function uses **protect** to declassify information. The error provided by our prototype is the following:

```
You try to protect the expression: "let userRecord = ((findUser u) passwordFile)
in if null userRecord then False else (snd head userRecord == p)" at: "(line 12, column 32) to level: Low
But the expression you are protecting is protected at level: High
```

```
log = fn x ⇒ protect x Low
fLow = fn x ⇒ protect x Low
fHigh = protect fLow High
main = log (fLow 2 + fHigh 3)
```

Fig. 29. An example to show the effect of highly confidential functions.

```
log = fn x ⇒ protect x Low
lVal = protect 2 Low
fakeId = fn x ⇒ let y = x in (protect y High)
main = log (
  if True then
    if False then lVal + 2 else 10
  else if fakeId False then lVal else lVal + 1)
```

Fig. 30. An example with a nested conditional.

```
Try declassify to assign the expression to the level: Low
```

This time one of the security specific heuristics discovers the mistake, and generates an error message that suggests to replace **protect** by **declassify**.

In Fig. 29 (on the next page) we have implemented a function and protected the function (and not its return value) at level *High*; the default for functions is *Low*. Because of the rule for application, a return value of this function will be the join of the confidentiality of the body and the function itself, which will in this case always be *High*. This leads to an inconsistency, because *log* expects a value with *Low* confidentiality. The error report now becomes

```
The function: fHigh at (l 6, c 22)
in the application:
  (fHigh 3) at (l 6, c 22)
is protected at level: High.
This causes the result of the application to be protected at level: High.
Instead a value protected at level: Low that was expected by: log ....
```

A final example is the program in Fig. 30, where we provide a nested **if**-statement in which the **if** in the **else** part leads to an inconsistency, because the function *fakeId* returns a value of *High* confidentiality. Note that the following message indeed picks out this particular subexpression to blame, but also refers to where the mistake shows up: in the call to *log*:

```
The  conditional: (fakeId False) at (l 8, c 12) of the if statement:
if (fakeId False) then lVal else (lVal) + (1) at (l 8, c 9)
uses a value: '(fakeId False) at (l 8, c 12)' protected at level: High.
This causes the whole if expression:
if (fakeId False) then lVal else (lVal) + (1) at (l 8, c 9)
to be protected at level: High.
Instead a value protected at level: Low was expected by: log ....
```

8. Related work

Sabelfeld and Myers provide a comprehensive overview of the field of security analysis [25], in particular the part of the field that derives from the work of Volpano, Smith and others [27]. Our work deals only with the issue of program confidentiality. In early work all data was labelled with a security level and checked dynamically; we follow the static approach that aims to reject programs at compile time.

Heintze and Riecke present a small statically typed, lambda calculus based, language for security analysis called the Slam calculus [14]. The call by value language employs two kinds of security annotations, one for direct readers and one for indirect readers. In the Slam calculus all values are explicitly annotated with both direct and indirect reader permissions. The type system they present features subtyping, making the analysis more accurate than ours. They support **protect**, but **declassify** is not available.

In [22] a security analysis for a lightweight version of ML (called Core ML) is presented. Their subject language forms the basis for the language used in this paper, although we have omitted some advanced features such as exceptions and references. Core ML does not have the **protect** construct; it is not needed in the presence of a subtyping rule. Instead, a higher than necessary security level for a value can be set explicitly by the programmer; if, then, the inferred security level is higher than the explicit annotation, the program will be rejected. The Core ML specification is polyvariant, and the underlying language is polymorphic, as it is in this paper. This allows the programmer to write security agnostic functions.

Security analysis has been shown to be an instance of dependency analysis [2]. This work was later generalised to a polymorphic setting [1]. According to the authors, the three advantages for defining analyses as instances of their Dependency Core Calculus (DCC) are that it allows one to find out in which sense instances of dependency analyses differ, the mapping from dependency analysis to the instance can be used to verify that the expected dependencies do indeed arise, and the encoding of dependency analyses into DCC yield simple proofs of non-interference for these analyses. A fourth advantage may be that heuristics that address the peculiarities of dependency-like analyses may also be transferable to other instances of DCC. Since the other instances of DCC discussed in [2] are optimising analyses, this may not seem to be of any use. However, when programmer provided explicit annotated type signatures may be provided, inconsistencies may again arise.

A lot of work has been done on diagnosing type errors for polymorphic, higher-order functional languages. The PhD thesis of Heeren [13] contains a comprehensive overview of the field up to 2004. Our work combines that of [8] and [26] with that of [13]. In [13], a framework for type inferencing and type error reporting is presented. As do many authors, the Hindley–Milner type system is specified in a constraint-based manner. The way we employ heuristics, and some of the heuristics themselves are described in [9]. Type error diagnosis for parametric polymorphism in Java is addressed in [6,7].

Whereas the previous work intends to provide as precise type errors as possible, [8] takes a different approach. When a type error occurs, a program slice is presented to the user. The slice contains all positions in the program that may contribute to the error; positions outside the slice are known not to contribute to the error. Their work is based on an algorithm for finding a minimal, inconsistent constraint set, from which a slice can be computed. Recent work shows that the method scales to a full sized language [23].

In this paper we combine the two approaches: when we find an inconsistency, we first restrict ourselves to the constraints originating from the associated program slice. Then we apply our heuristics to these constraints to see if they can come up with a more specific error message for the error at hand. Our algorithm for computing the slice is taken from [26]. Moreover, the authors of this paper advocate yet a third approach in which, instead of displaying a type error message, the programmer is assisted by an interactive type debugger. During this session the programmer will be asked to supply type information, in order to find out where the source of the inconsistency lies. A distinct advantage of this system is that it becomes much easier to discover mistakes in definitions that were found to be type consistent, but that, as judged from their use later in the program, were found to be at odds with the intentions of the programmer.

The intentions of the work in this paper are most closely related to that of [5] and [18]. Both papers investigate security type error diagnosis, and in contrast to our work, both do so in an imperative setting. In particular, the work of Deng and Smith [5] works for a simple imperative language extended with arrays, and develops a tailormade inference algorithm that additionally provides a recursive trace of the security levels of all the variables involved in the inconsistency. This is precise, but also verbose. The work by King et al. [18] is much more mature, and has been implemented in the Jif information-flow compiler. Their work applies to Java, and by employing an algorithm similar to the algorithm to compute the minimal unsatisfiable subset and using the fact that the analysis is implemented as a dataflow analysis, the system can provide an execution trace leading up to the inconsistency, but restricted to the security aspect only. In contrast to our work, the analysis they provide is context-insensitive, does not deal with higher-order functions, and does not seem to be able to deal with parametric polymorphism. A limited form of context-sensitivity is attained by essentially duplicating the analyses of pieces of code for use in a secure and a non-secure context; moreover the choice to use either one of these versions has to be made explicitly by the programmer. This decreases the usability of the system and it still remains to be seen how well this extends to larger lattices. Moreover, there is an ad-hoc extension in order to deal with implicit control-flow, for which they need to introduce additional security variables (which amounts to analysing an SSA form of the program). As a result, the type error diagnosis is not always easy to map back onto the original program. Our work can handle implicit flows, in fact explicit flows and implicit flows cannot be distinguished in a higher-order setting. Moreover, some of our heuristics in particular address issues arising from implicit control-flow (Section 6.3). The authors suggest adding heuristics to their work as future work, which is one of our contributions.

A totally different approach to security typing, is to employ an embedded domain-specific language, SecLib [24]. The library uses type classes and monads to enforce security. All security levels and flows are checked by the Haskell type system, illegal flows are reported as regular type errors. All functions have explicit types, as the library heavily relies on type classes for which the programmer has to restrict the instances available. Furthermore the library requires some language extensions to be enabled (all of them are related to type classes). The compiler cannot infer the correct type for all expressions, so we have to help it by providing explicit types. Although beneficial from the viewpoint of language engineering, the fact that we embed the security types into normal types, implies that type errors and security errors cannot easily be distinguished: normal type errors will also reveal security type annotations, and vice versa. Also, one may prefer that security errors will only be reported for type correct programs.

In our security type system we employ qualifiers for storing constraints in types. The theory of qualified types was developed in [17]. Another more well-known example of qualified types are the type classes of Haskell, where qualifiers can be used to impose restrictions on let-polymorphic types, e.g., the equality operator has the type $\forall \alpha . Eq\, \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \texttt{Bool}$, i.e., it is only well-typed for those types α that are instances of the Eq type class. In this paper, the predicates we store in types are only meant to relate the security annotations of the various parts that make up a type. In [11], the same technique was used to exploit qualified types to model polyvariance, but then in the context of usage analysis.

9. Conclusion and future work

In this paper we combine a heuristics-based approach to type error diagnosis with a type error slice approach, in order to improve security type error diagnosis. We first compute a minimal unsatisfiable set of constraints, that allows us to determine the locations in a program that contribute to an error. Since these slices can be large, we try to provide more specific messages by applying a number of heuristics to this minimal unsatisfiable subset. We have presented heuristics divided into four categories: generic, propagation, dependency analysis specific and security analysis specific heuristics.

Obvious directions for future work are: extending the source language, extending the set of heuristics, and to consider how the order in which heuristics are applied affects type error diagnosis. Although we have provided a rationale for our heuristics, a full scale experimental validation still needs to be executed. A problem is the absence of a benchmark collection of security incorrect programs. The minimal unsatisfiable subsets give a certain guarantee of “soundness”, but that does not mean that our messages often reflect the diagnosis that an expert may come up with. A possible approach is to consider what has changed with respect to previous compiles.

Acknowledgements

This work was supported by the NWO project on "Scriptable Compilers" (612.063.406). The authors would like to thank Sean Leather and anonymous referees for their helpful comments. The present article is a revised version of a paper [29] that was presented at the 2013 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation in Rome, Italy. The authors wish to express their gratitude to the anonymous reviewers of POPL 2012, PEPM 2013 and the current journal version, for their insightful comments and helpful suggestions. Finally, the second author thanks the chairs of the VODCA 2008 workshop, in particular Alessandro Aldini, for encouraging this research into error diagnosis for security analyses as a follow-up to [10].

References

[1] M. Abadi, Access control in a core calculus of dependency, Electron. Notes Theor. Comput. Sci. 172 (2007) 5–31.
[2] M. Abadi, A. Banerjee, N. Heintze, J.G. Riecke, A core calculus of dependency, in: POPL '99: Proceedings of the 26th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 1999, pp. 147–160.
[3] L. Damas, R. Milner, Principal type-schemes for functional programs, in: POPL '82: Proceedings of the 9th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 1982, pp. 207–212.
[4] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, Cambridge University Press, 1990.
[5] Z. Deng, G. Smith, Type inference and informative error reporting for secure information flow, in: Proceedings of the 44th Annual Southeast Regional Conference, ACM-SE 44, ACM, New York, NY, USA, 2006, pp. 543–548.
[6] N. el Boustani, J. Hage, Corrective hints for type incorrect Generic Java programs, in: J. Gallagher, J. Voigtländer (Eds.), Proceedings of the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM '10), ACM Press, 2010, pp. 5–14.
[7] N. el Boustani, J. Hage, Improving type error messages for generic java, High.-Order Symb. Comput. 24 (1) (2012) 3–39, http://dx.doi.org/10.1007/s10990-011-9070-3.
[8] C. Haack, J.B. Wells, Type error slicing in implicitly typed higher-order languages, Sci. Comput. Program. 50 (1–3) (2004) 189–224.
[9] J. Hage, B. Heeren, Heuristics for type error discovery and recovery, in: Z. Horváth, V. Zsók, A. Butterfield (Eds.), Implementation of Functional Languages – IFL 2006, vol. 4449, Springer Verlag, Heidelberg, 2007, pp. 199–216.
[10] J. Hage, B. Heeren, Strategies for solving constraints in type and effect systems, Electron. Notes Theor. Comput. Sci. 236 (2009) 163–183.
[11] J. Hage, S. Holdermans, A. Middelkoop, A generic usage analysis with subeffect qualifiers, in: ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA, 2007, pp. 235–246.
[12] B. Heeren, J. Hage, S.D. Swierstra, Scripting the type inference process, in: Eighth ACM Sigplan International Conference on Functional Programming, ACM Press, 2003, pp. 3–13.
[13] B.J. Heeren, Top quality type error messages (PhD), September 2005.
[14] N. Heintze, J.G. Riecke, The slam calculus: programming with secrecy and integrity, in: POPL '98: Proceedings of the 25th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 1998, pp. 365–377.
[15] G.F. Johnson, J.A. Walz, A maximum-flow approach to anomaly isolation in unification-based incremental type inference, in: POPL '86: Proceedings of the 13th ACM Symposium on Principles of Programming Languages, ACM, New York, 1986, pp. 44–57.
[16] R. Johnson, D. Wagner, Finding user/kernel pointer bugs with type inference, in: Proceedings of the 13th Conference on USENIX Security Symposium – Volume 13, 2004, pp. 119–134.
[17] M.P. Jones, Qualified Types: Theory and Practice, Cambridge University Press, New York, NY, USA, 1994.
[18] Dave King, Trent Jaeger, Somesh Jha, Sanjit A. Seshia, Effective blame for information-flow violations, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, ACM, New York, NY, USA, 2008, pp. 250–260.
[19] B.S. Lerner, M. Flower, D. Grossman, C. Chambers, Searching for type-error messages, in: ACM SIGPLAN Notices, vol. 42, ACM, 2007, pp. 425–434.
[20] J.M. Lucassen, D.K. Gifford, Polymorphic effect systems, in: POPL '88: Proceedings of the 15th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 1988, pp. 47–57.
[21] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, Berlin, 1999.
[22] F. Pottier, V. Simonet, Information flow inference for ML, ACM Trans. Program. Lang. Syst. 25 (1) (2003) 117–158.
[23] V. Rahli, J.B. Wells, F. Kamareddine, A constraint system for a SML type error slicer, Technical Report HW-MACS-TR-0079, Herriot Watt University, Edinburgh, Scotland, Aug 2010.
[24] A. Russo, K. Claessen, J. Hughes, A library for light-weight information-flow security in Haskell, in: Haskell '08: Proceedings of the first ACM SIGPLAN Symposium on Haskell, ACM, New York, NY, USA, 2008, pp. 13–24.
[25] A. Sabelfeld, A.C. Myers, Language-based information-flow security, IEEE J. Sel. Areas Commun. 21 (2003) 2003.
[26] P.J. Stuckey, M. Sulzmann, J. Wazny, Interactive type debugging in Haskell, in: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell, ACM, New York, NY, USA, 2003, pp. 72–83.
[27] D. Volpano, C. Irvine, G. Smith, A sound type system for secure flow analysis, J. Comput. Secur. 4 (2–3) (1996) 167–187.
[28] J. Weijers, Feedback-oriented security analysis, MSc thesis, http://www.cs.uu.nl/people/jur/jweijers-msc.pdf, 2010.
[29] J. Weijers, J. Hage, S. Holdermans, Security type error diagnosis for higher-order, polymorphic languages, in: Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM '13, ACM, New York, NY, USA, 2013, pp. 3–12.