Security Type Error Diagnosis for Higher-Order, Polymorphic Languages

Jeroen Weijers
Dept. of Comp. Science,
Universität Tübingen
Sand 13, 72076 Tübingen,
Germany
jeroen.weijers@uni-tuebingen.de

Jurriaan Hage
Dept. of Inf. and Comp. Sciences,
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht,
The Netherlands
J.Hage@uu.nl

Stefan Holdermans
Vector Fabrics
Paradijslaan 28, 5611 KN Eindhoven,
The Netherlands
stefan@vectorfabrics.com

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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Functional constructs, Type structure

General Terms Languages, Theory

Keywords type-based program analysis, security analysis, error feedback

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 has not always been the case. As the literature study of Heeren [10] 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 [24].

Consider the following expression (taken from [11]; this is not code anyone would care to write) for which we have provided explicit type and *security* annotations:

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

PEPM'13, January 21–22, 2013, Rome, Italy.

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 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: because the value of the condition decides whether the **then** or **else** part must be evaluated, so observing the value of the low confidentiality result reveals information about the highly confidential conditional. Therefore, the expression is not security type correct (although it *is* type correct) and it should be rejected. The problem can 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. [15] 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 analyze 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 themself suggest, their work has not been combined with heuristics to further prune the traces they provide (see [5, 8, 13] for work on such heuristics).

This paper offers the following contributions:

- We are the first to combine the type error slicing approach of Haack and Wells [7] with the heuristic approach of Heeren [10].
- 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 describe our approach in Section 2. In Section 3 we define the subject language, specify the more interesting parts of the security-annotated type system and indicate how the type system relates to the heuristics discussed in Section 4. Section 5 provides further examples as a first step in

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

Figure 1. Error message for FlowCaml.

validating our work. In Section 6 we discuss related work, and Section 7 concludes. For reasons of space we omit many details, in particular, example code for the FlowCaml system and SecLib library, parts of the security type system, and the complete inference algorithm and constraint solver that implement the security type system. These details can be obtained from [25].

2. 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 precision is to a large extent compensated for by the polyvariance of the analysis). Our source language is a higher-order polymorphic functional language, very much akin to FlowCaml, an implementation based on the Core ML language [19].

In a FlowCaml program the programmer describes the relations that must hold in the lattice of security levels, e.g., $!everyone < !root$. If, during analysis, the analyzer finds that $!root < !everyone$, then an error has occurred, and this is communicated to the programmer. An example error message of this kind is given in Figure 1. The message explains there is an illegal flow, and the location points to the line where the definition that contains the inconsistency begins. Although correct, the message does not explain how the flow was derived, which subexpressions were responsible, and what the programmer can do to fix the problem. For reasons of space, we omit the example program written in FlowCaml and further discussion (but see [25]).

As we explain below, our work essentially combines the approaches of Hage and Heeren [8, 10], and Haack and Wells [7]. 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.

Due to space restriction we refer the reader to Section 7 of Stuckey, Sulzmann and Wazny [23] for the algorithm to compute a minimal unsatisfiable set of constraints.

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 4, 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 [7].

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 [1]. 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.

3. Security type system

The language we use is based upon the Fun language of [18], a let-polymorphic call-by-value lambda calculus. We have extended Fun with a few special purpose security program constructs as well as some additional features to make the analysis and examples more interesting. We call this extended language sFun++.

In this paper we employ the following syntactic categories as given below. Most categories should speak for themselves. We note that the category **Sec** ranges over security levels, which we assume to form a lattice [3], 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 $\sqcup$.

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

The abstract syntax of our language is defined as:

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

A program p is a list of declarations, and each declaration binds an expression to an identifier. As in [18], **fn** $x \Rightarrow e_0$ defines a non-recursive function and **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 **let** $x = e_0$ **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 **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 expression **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).

3.1 The sFun++ type language

As usual, we specify the security type system as an annotated type system ([17], and Chapter 5 of [18]). Our security analysis is polyvariant, which means that we can quantify over annotation variables. The relations between annotation variables, and restrictions on them are expressed as constraints. These constraints may be added to types, in the style of qualified types [14].

We introduce the following new syntactic categories:

$$
\begin{array}{rcll}
\alpha & \in & \textbf{TyVar} & \textit{type variables} \\
\beta & \in & \textbf{AnnVar} & \textit{annotation variables} \\
\pi & \in & \textbf{Constr} & \textit{constraints} \\
l & \in & \textbf{Levels} & \textit{security levels} \\
\varphi & \in & \textbf{Ann} & \textit{security annotations} \\
\tau & \in & \textbf{Ty} & \textit{annotated types} \\
\rho & \in & \textbf{Qualified Types} & \textit{qualified types} \\
\sigma & \in & \textbf{TyScheme} & \textit{annotated type schemes} \\
C & \in & \textbf{Constraints} & \textit{constraint set} \\
\Gamma & \in & \textbf{TyEnv} & \textit{type environments}
\end{array}
$$

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.

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

We then define a three-layer type language:

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

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, and existentially only over annotation variables. The latter facility is used to deal with annotation type variables that may be constrained in some way, but that are not exposed as part of the type. Note that in contrast to FlowCaml [19] 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.

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

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 can not quantify 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 sub-effecting will counter this loss. Constraint sets C will be used to store a constraint environment in our typing judgment.

3.2 The security type system

In Figure 2 we give the non syntax directed type rules for our security analysis. For reasons of space, we provide only those of a core calculus and the ones that we shall need in the remainder of this paper. The full type system, as well as the syntax-directed variant and the inference algorithm can be found in [25].

The main judgment 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.

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 has not already conveyed (see [*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 $\bot$, so then the annotation of the application only depends on the annotation on the result of the body.

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 φ_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. To alleviate this problem, the rule for subeffecting can be used to selectively increase the level of protection for an expression. **1, example?** The rules for generalisation and instantiation of types and annotions are straightforward, and we omit the details.

Our type system specification follows those described in [1, 19]. We are confident therefore that — omitting the rule for declassification —, the full 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.

3.3 Towards an algorithm

The full type system can be transformed to an algorithm, following Chapter 5 of [18]. First, we turn the type system of Figure 2 into a type system that is completely syntax-directed, e.g., by performing generalisation of the let-definition before passing the computed type schemes into the body of the let. Then we implement a variant of algorithm W [2] that computes the types of variables during a tree traversal, doling out fresh annotation variables wherever necessary, and equating these variables during unification. The annotations themselves obtain their particular security level when we solve the security constraints that are collected during the tree traversal. 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 to establish over which annotation variables we should universally or existentially quantify. This explains the use of *simplify* in the definition of *gen* given in Figure 3. 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 are free in the environment Γ, while C' contains the remaining constraints. We quantify universally over the type and annotation variables that occur free in τ, and quantify existentially over the remaining free annotation variables (from C'). The constraints from C' can be stored in the type scheme, because they only involve annotation variables that we just quantified over, while the constraints in C'' are returned for further propagation.

Note that 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.

$$
\begin{array}{l}
\mathit{gen}\ \Gamma\ \varphi\ \tau\ C = \\
\quad (\forall \alpha_1 \ldots \alpha_n.\, \forall \beta_1 \ldots \beta_m.\, \exists \beta_{m+1} \ldots \beta_p.\, C' \Rightarrow \tau, C'')\ \textbf{where} \\
\quad\quad (C', C'') = \mathit{simplify}\ C \\
\quad\quad \{\alpha_1, \ldots, \alpha_n\} = \mathit{ftv}(\tau) - \mathit{ftv}(\Gamma) \\
\quad\quad \{\beta_1, \ldots, \beta_m\} = \mathit{fav}(\tau) - \mathit{fav}(\Gamma) - \mathit{fav}(\varphi) \\
\quad\quad \{\beta_{m+1}, \ldots, \beta_p\} = \mathit{fav}(C') - \mathit{fav}(\tau) - \mathit{fav}(\Gamma) - \mathit{fav}(\varphi)
\end{array}
$$

Figure 3. The generalisation function *gen*

4. 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 describe the heuristics themselves, and define the order in which they are applied, and why. In Section 5 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.

4.1 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 the source of an inconsistency [12]. We use this idea to point to a possible mistake in a value that is involved in a security error. 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 amount 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 Figure 4, the first five lines declare some values, where the first four are protected at level *Low* and the last value is protected at level *High*. The declaration *fifteen* on the sixth line computes the sum of the five values and passes the result to the *print* function. 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 the 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 (see Figure 5). The programmer can now decide whether the use of *five* in this place was incorrect or whether the use of *print* was at fault.

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

$$\frac{\Gamma\,[x \mapsto (\tau_x, \varphi_x)], C \vdash e_0 : {\tau_0}^{\varphi_0}}{\Gamma, C \vdash \textbf{fn}\ x \Rightarrow e_0 : {\tau_x}^{\varphi_x} \rightarrow^{\varphi} {\tau_0}^{\varphi_0}}\ [\textit{t-fn}] \qquad \frac{\Gamma, C \vdash e : \tau^{\varphi_1} \quad C \vdash \varphi_1 \sqsubseteq \varphi}{\Gamma, C \vdash e : \tau^{\varphi}}\ [\textit{t-sub}]$$

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

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

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

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

Figure 2. Non-syntax directed rules for security analysis

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

Figure 4. Example program: majority of Low values in faulty subexpression

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

Figure 5. Error generated by the majority heuristic in Figure 4

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 5 we discuss an example of this kind.

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 [10].

4.2 The propagation heuristic

Many of the generated constraints only propagate security levels. Consider for example the program in Figure 6. The highly secure $\mathit{secureVal}$ is passed through the identity function once and then twice in succession 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 $\mathit{secureVal}$ 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.

4.3 Heuristics for dependency analyses

Security analysis is an instance of a dependency analysis [1], 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

$$
\begin{array}{l}
secureVal :: \texttt{Int}^{High} \\
incr :: \forall\beta_1.\,\forall\beta_2.\,\beta_1 \sqsubseteq \beta_2 \Rightarrow \texttt{Int}^{\beta_1} \rightarrow \texttt{Int}^{\beta_2} \\
id :: \forall\alpha.\,\forall\beta_1.\,\forall\beta_2.\,\beta_1 \sqsubseteq \beta_2 \Rightarrow \alpha^{\beta_1} \rightarrow \alpha^{\beta_2} \\
print :: \forall\alpha.\,\alpha^{Low} \rightarrow \alpha^{Low} \\
print\ (incr\ (incr\ (id\ secureVal)))
\end{array}
$$

Figure 6. Propagating security levels through $incr$ and id

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

Figure 7. Secure information leaks through conditional

```
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.
```

Figure 8. Error generated by the if-heuristic for the program in Figure 7

$$
\begin{array}{l}
secureValue = \textbf{protect}\ True\ High \\
printSecure = printValue\ (\textbf{protect}\ secureValue\ Low)
\end{array}
$$

Figure 9. Example program: misuse of protect statement

programmer may believe that the security level of the condition does not contribute to that of the whole conditional, although the type rule in Figure 2 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 Figure 2. For most of the discrepancies we have found, we have implemented a corresponding heuristic. We shall consider all of these below, 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 Figure 7 we present a program where secure information is leaked through the conditional. The error presented in Figure 8 is generated by our heuristic. This is an example of a program that 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.

4.4 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** 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 Figure 9 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 error message presented in Figure 10. The error 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.

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 [9].

4.5 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

```
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
```

Figure 10. Error generated by sibling heuristic for the program in Figure 9

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 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 Figure 4 would, for example, result in the slice presented in Figure 11, which indeed only shows those point 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

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

Figure 11. Program slice error from the program in Figure 4

picks from the remaining constraint the one that occurs earliest in the program, and bases the error report on that constraint.

5. Further examples

Validation of our work is best performed on a large corpus of programs, following the example of [16] 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 [15], but that is a Java based system, a context in which we have no polyvariance and no higher-order functions (see Section 6 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 Figure 12 we present a security type correct login program written in sFun++. 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 Figure 13 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 given in Figure 15. In this case, it is the least trusted constraint heuristic that correctly blames the application.

A mistake that is easily made is that the programmer tries to declassify information through the **protect** statement. In Figure 14 the login function uses **protect** to declassify information. The error provided by our prototype is shown in Figure 16. This time one of the security specific heuristics discovers the mistake, and generates an error message that suggests to replace **protect** by **declassify**.

We now continue with an example of an invocation of **hd** that leads to an inconsistency in Figure 17. In this case, 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**; that the contents of the list has confidentiality Low is not important. The error message that this results in can be found in Figure 18. Strikingly, the error messages changes substantially (to the one in Figure 19) when we change the confidentiality level of the subexpression 1 in the definition of zl to $Medium$. This is because we now have two sources for the fact that **hd** zl is not of level Low. So now the application of log to its argument is blamed. Fortunately, 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.

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

Figure 12. An sFun++ login program

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

Figure 13. 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)
```

Figure 14. The login function with protection

```
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 at most level:
    Low
But the argument: "let userRecord ... == p)"
Is protected at a higher level.
```

Figure 15. sFun++ error given for program in Figure 13

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

Figure 16. sFun++ error given for program in Figure 14

```
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)
```

Figure 17. Another example

```
The structure of list: zl at (l 11, c 40)
applied to head in the expression:
  head 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 ....
```

Figure 18. sFun++ error given for program in Figure 17

```
Error in application:
(log if (id (id boolVal)) then (id lVal) else
    head zl) at (l 10, c 8)
An argument protected at 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)
```

Figure 19. sFun++ error given for program in Figure 17

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

Figure 20. An example to show the effect of highly confidential functions

```
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 ....
```

Figure 21. sFun++ error given for the program in Figure 20

$log = \textbf{fn}\ x \Rightarrow \textbf{protect}\ x\ Low$
$lVal = \textbf{protect}\ 2\ Low$
$fakeId = \textbf{fn}\ x \Rightarrow \textbf{let}\ y = x\ \textbf{in}\ (\textbf{protect}\ y\ High)$
$main = log\ ($
  $\textbf{if}\ True\ \textbf{then}$
    $\textbf{if}\ False\ \textbf{then}\ lVal + 2\ \textbf{else}\ 10$
  $\textbf{else if}\ fakeId\ False\ \textbf{then}\ lVal\ \textbf{else}\ lVal + 1)$

Figure 22. An example with a nested conditional

```
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 ....
```

Figure 23. sFun++ error given for program in Figure 22

Consider Figure 20. Here 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 is shown in Figure 21.

A final example is the program in Figure 22, 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 message in Figure 23 indeed picks out this particular subexpression to blame, but also refers to where the mistake shows up: in the call to log.

6. Related work

Sabelfeld and Myers provide a comprehensive overview of the field of security analysis [22], in particular the part of the field that derives from the work of Volpano, Smith and others [24]. 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 [11]. 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 sub-typing, making the analysis more accurate than ours. They support **protect**, but **declassify** is not available.

In [19] 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.

Security analysis has been shown to be an instance of dependency analysis, see, e.g., [1]. The authors mention three advantages for defining analyses as instances of their Dependency Core Calculus (DCC), to which we add a fourth that says that heuristics that address the peculiarities of a dependency-like analysis may also be transferable to other instances of DCC. Although the other instances of DCC discussed in [1] are optimising analyses, inconsistencies may again arise if dependency type signatures are allowed.

A lot of work has been done on providing feedback for type errors for polymorphic, higher-order functional languages. The PhD thesis of Heeren [10] contains a comprehensive overview of the field up to 2004. Our work combines that of [7] and [23] with that of [10]. In [10], 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 [8]. Type error diagnosis for parametric polymorphism in Java is addressed in [6].

Whereas the previous work intends to provide as precise type errors as possible, [7] take 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 size language [20].

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 constraint 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 [23]. Moreover, they advocate a third approach in which, instead of displaying a type error message, the programmer is assisted by an interactive type debugging session.

The intentions of the work in this paper are most closely related to that of [4] and [15]. 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 [4] 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. [15] 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; 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 4.3). 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 [21]. 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.

7. 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 affect 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.

Acknowledgments

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.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM.

[3] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990.

[4] Z. Deng and G. Smith. Type inference and informative error reporting for secure information flow. In *Proceedings of the 44th annual Southeast regional conference*, ACM-SE 44, pages 543–548, New York, NY, USA, 2006. ACM.

[5] N. el Boustani and J. Hage. Corrective hints for type incorrect Generic Java programs. In J. Gallagher and J. Voigtländer, editors, *Proceedings of the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM '10)*, pages 5–14. ACM Press, 2010.

[6] N. el Boustani and J. Hage. Improving type error messages for generic java. *Higher-Order and Symbolic Computation*, 24(1):3–39, 2012. 10.1007/s10990-011-9070-3.

[7] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

[8] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation of Functional Languages – IFL 2006*, volume 4449, pages 199 – 216, Heidelberg, 2007. Springer Verlag.

[9] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13. ACM Press, 2003.

[10] B. J. Heeren. Top quality type error messages (phd), September 2005.

[11] N. Heintze and 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*, pages 365–377, New York, NY, USA, 1998. ACM.

[12] G. F. Johnson and 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*, pages 44–57, New York, 1986. ACM.

[13] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pages 9–9, 2004.

[14] M. P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1994.

[15] Dave King, Trent Jaeger, Somesh Jha, and 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, pages 250–260, New York, NY, USA, 2008. ACM.

[16] B.S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *ACM SIGPLAN Notices*, volume 42, pages 425–434. ACM, 2007.

[17] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM.

[18] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, 1999.

[19] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[20] V. Rahli, J. B. Wells, and F. Kamareddine. A constraint system for a SML type error slicer. Technical Report HW-MACS-TR-0079, Herriot Watt University, Edinburgh, Scotland, Aug 2010.

[21] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, New York, NY, USA, 2008. ACM.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[23] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83, New York, NY, USA, 2003. ACM.

[24] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.

[25] J. Weijers. Feedback-oriented security analysis (msc thesis), 2010. `http://www.cs.uu.nl/people/jur/jweijers-msc.pdf`.