Feedback-Oriented Security Analysis

Jeroen Weijers

MSc Thesis

April 1, 2010

INF/SCR-09-48

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily Supervisor:
dr. Jurriaan Hage
Second Supervisor:
drs. Stefan Holdermans

CONTENTS

ABSTRACT

Security analysis can provide a static guarantee that all information flows in a program are secure. We discuss two existing implementations and the feedback they give when an inconsistency is found. A programming language is presented with primitive constructs for protection and declassification. For this language we define the security analysis in a constraint based type and effect system. Our type system is both polymorphic and polyvariant, and has sub-effecting. The set of constraints that were generated by the security analysis are processed by heuristics when an inconsistency is found. We present several heuristics that generate an error message describing the cause of the inconsistency and if possible suggest a fix. We implemented the system and present some error messages that were generated by it.

ACKNOWLEDGEMENTS

With this thesis I complete my master, which I started in 2007, at the Center for Software Technology led by Doaitse Swierstra at Utrecht University. The lecturers of the courses I attended during this master all very enthusiastically taught me about their work. I am very impressed by the expertise and commitment of all the members of the group. I thank them all very much for teaching, challenging and inspiring me.

I would not have been able to write this thesis without the help of my supervisors Jurriaan Hage and Stefan Holdermans. I am very grateful to them for supervising me, and giving me the chance to participate in their research.

My primary supervisor Jurriaan Hage has been a great supervisor and advisor. Whenever I had a question he was willing to help me and made sure I fully understood the case. I would like to thank him very much for the enlightening discussions that we often had and for being a never ending source of new ideas. Also I would like to express my appreciation for him proof-reading this thesis and help me prepare for my thesis defence.

My secondary supervisor Stefan Holdermans was always available for answering questions, even after he started at his new job. I am very impressed by his technical expertise and very grateful for his willingness to help me. I would like to thank him very much for sharing his insights and giving valuable advice during the spontaneous and the weekly progress discussions.

Special thanks go to Bastiaan Heeren from the Open Universiteit Nederland for explaining to me his work on TOP, and Arie Middelkoop for answering my questions regarding AG and Haskell.

I would like to thank my friends Hugo Kuijf and Rogier Saarloos whom I both met at the very start of my study and with whom I had a great time during the studies. I would also like to thanks Tom Lokhorst for the enlightening discussions I had with him while working on this thesis. And also thanks to the other ‘lab-inhabitants’ who helped me and most of all contributed to having a fun time during my thesis project.

Lastly I would like to thank my family and friends for supporting me during the studies, encouraging me and believing in me. I could not have completed this study without them.

1. INTRODUCTION

In this thesis we discuss security analysis [Sabelfeld and Myers, 2003] and more importantly how to provide feedback to a programmer whenever he or she by accident or deliberately tries to leak secure information. In this chapter we introduce security analysis (Section 1.1). We also discuss what the difficulties are of providing feedback over security analysis (Section 1.2). In the remainder of this chapter we give an overview of our contributions, in Section 1.3, and the structure of this thesis 1.4.

1.1 Security Analysis

Writing a program that handles secure data and is used by all sorts of users (including some less authorised users) is a difficult task. The programmer has to make sure no sensitive data leaks to parts of the program that are visible to low authorised users. He also has to take into account that some control flow structures might indirectly reveal sensitive information. Figure 1.1 shows a program with security annotations that should be rejected. The boolean values in the program are explicitly typed, the conditional boolean has type $\texttt{Bool}^{\mathsf{H}}$ where the H means that the boolean is secure. The type of the booleans in the branches $\texttt{Bool}^{\mathsf{L}}$ means that the booleans are non secure. The whole expression was assigned the type $\texttt{Bool}^{\mathsf{L}}$. This is however wrong, the whole expression reveals the original value of the secure boolean. These sorts of programs should be rejected by the compiler because the branches of the **if** construction reveal the value of the high secure condition as low values.

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

Fig. 1.1: Illegal information flow (example from [Heintze and Riecke, 1998])

In order to guarantee that no information is leaked, an analysis that statically determines that no secure data is leaked in the program can be employed. This analysis, called security analysis [Volpano et al., 1996], requires the programmer to take into account the security levels of the program inputs, and specify the levels of secrecy he expects or needs at the outputs of the program (matching the level of users that will look at this output). Such analyses already exist [Sabelfeld and Myers, 2003] [Pottier and Simonet, 2003], but unfortunately little work has been done on providing feedback when inconsistencies are encountered by such an analysis. This might be one of the reasons that this sort of analysis is still not common in general programming languages.

1.2 Problem of providing feedback

Modern compilers perform all kinds of analyses. We distinguish between three kinds of analyses, namely optimising, comprehending and validating analyses. Optimising analyses are typically applied silently, meaning that they do not give any feedback to the programmer. This feedback is not necessary because it optimises the generated code without changing the semantics of the program. It does not inspect the program for any abnormalities. The second kind of analyses, comprehending analyses, is aimed at gathering information about the program; it might present this information to the programmer but it will not tell the programmer whether the program is correct. In fact the two analyses mentioned before are usually only applied to programs that are valid. The third kind of analyses, validating analyses, verifies that a certain property holds for the program. If the property does not hold the program is rejected, and this is usually reported to the programmer by providing a small diagnoses on the problem. An example of such a validating analysis is type checking and inferencing for functional programming languages [Damas and Milner, 1982]. Type error messages for functional languages had for a long time fallen short in explaining the cause of the problem. This lead to a lot of research to improve the quality of type error messages [Heeren, 2005], [Haack and Wells, 2004], [Stuckey et al., 2003]. For other validating analyses much less work has been done on providing good error messages. Security analysis is such an analysis.

A major difficulty of providing proper error message is the presence of polymorphism/polyvariance. Polymorphism is a feature that makes the language more useful as it allows the programmer to write only one version of a function that works for all types. Consider for example the identity function for integers, defined in Figure 1.2. If we now would like to have an identity function for boolean values we would need to write another version, defined in Figure 1.3.

$$
\begin{aligned}
&\mathit{intId} :: \texttt{Int} \rightarrow \texttt{Int} \\
&\mathit{intId}\ x = x
\end{aligned}
$$

Fig. 1.2: Identity function for `Int`'s

$$
\begin{aligned}
&\mathit{boolId} :: \texttt{Bool} \rightarrow \texttt{Bool} \\
&\mathit{boolId}\ x = x
\end{aligned}
$$

Fig. 1.3: Identity function for `Bool`'s

We see that the two versions of the identity function only differ in their types, but the implementations are the same. With polymorphism we only need to write one version of the identity function that will then work for any type, as is presented in Figure 1.4.

Polyvariance does the same for the property that is checked by the analysis as polymorphism does for types; this property will be referred to as annotations hereinafter. So the programmer would not have to write a version for each security context for functions that do not require any particular security level. Unfortunately polyvariance also leads to propagation of inconsistencies. A value that gets a wrong security level can be

$$id :: \alpha \to \alpha$$
$$id\ x = x$$

Fig. 1.4: Polymorphic identity function

propagated through a lot of polyvariant function before it leads to problems. For type systems this appears to be not much of a problem as the type errors in most program do not seem to propagate. This is due to explicit types compartmentalising the program so that type errors are found locally. For security analysis we do not have this possibility as most security annotations are not explicit in the program. Most functions are completely polyvariant and the programmer should not have to add the security annotations to the type signature.

1.3 Contribution

The Security analysis presented in this thesis is based on the system presented by Pottier and Simonet [Pottier and Simonet, 2003]. We defined the analysis on a small language called sFun++ and introduce a construct for explicit declassification. We present a constraint based inference algorithm for security analysis, based on the work of Pottier and Simonet. We then use these constraints to generate error messages separately from type error messages. We combine the idea of heuristics from Heeren [Heeren, 2005] and type error slicing from Haack and Wells [Haack and Wells, 2004] to find the causes of the problem and present a clear and helpful error message.

1.4 Organisation of the thesis

In this thesis we take a stepwise approach in building up the security analysis and conclude with the issue of providing feedback. We will first discuss our motivation to research providing better feedback, then we discuss some related work, upon which we will build our analysis. The thesis is organised as follows:

Chapter	Contents
Chapter 2: Motivation and Examples	Motivation for providing better feedback. Examples of two already existing security systems, the flowCaml language [Pottier and Simonet, 2003] and a Haskell library for security [Russo et al., 2008].
Chapter 3: Related work	Work upon which we build, broad overview of security analysis, type based security analysis, type error messages and qualfied types.
Chapter 4: Language and Type System	Introduction of the Fun language [Nielson et al., 1999], and our extension to sFun++. Introduction of the basic language, monomorphic type system, polymorphic type system and syntax directed type system.
Chapter 5: Security Type System	The security type system. Extensions of the basic type language, explanation of security lattices, non syntax directed type and effect system and syntax directed type and effect system.
Chapter 6: Inference Algorithm	Security inference algorithm. Introduction of substitutions, the unification algorithm, the security inference algorithm based on algorithm $\mathscr{W}$ [Damas and Milner, 1982], and constraint solving.
Chapter 7: Error Messages	Providing error messages. Computing the minimal unsatisfiable constraint set. Various heuristics to determine the cause of the error and explain the error.
Chapter 8: Conclusions	Conclusions and future work.

2. MOTIVATION AND EXAMPLES

Quite some work has been done to provide confidentiality in programming languages (see Chapter 3). In this chapter we explain the need for quality error message for security analysis. We do so by explaining two implementations of a typical use of security analysis. The implementations use a different approach of performing security analysis. The example is a login program. In login programs the password file is usually protected very well. We want to ensure that the password data does not leak to users who are not authorised to see the data. These systems usually use a file where user data is stored and accessing this file leads to a lot of code dedicated to file access. As we are merely interested in the error messages that are provided by the compiler when secure information is leaked in the program we chose not to obfuscate the program with file access code but instead just provide the data in a variable.

2.1 FlowCaml example

Our first implementation is written in FlowCaml, which is an OCaml dialect with a security extension. During the login procedure the password provided by the user is compared with the password that is stored in the system. The stored password is a secure value of which the system is never allowed to leak any information (not the password itself, but also not the length of the password). However it is necessary to tell the user whether the provided password matches the required password, which means leaking some secure information to a potentially untrusted person. The only information that is leaked is whether the given password matches, so just a boolean value. In order to do so declassification is required. FlowCaml does not have a construct to perform declassification, any part that requires declassification has to be written in OCaml. This of course doesn't mean that an interface without security types is exposed from the module. The header that specifies the module interface is written in FlowCaml. The *checkPwd* function presented in Figure 2.1 is a function that takes a password of any security level that is below the root level, a user record and then returns a boolean protected at the same security level as the provided password. In the actual implementation of this function presented in Figure 2.2 there are no security levels and therefore we can leak the result of the comparison. In the Password module we also defined the datatype for the user records, in the header file we declare the security levels so that the FlowCaml part has to handle the password part of the record at security level root. In the Figures 2.3 and 2.4 we present the Login module. This module only exposes one function *login*, which takes a username and a password and then returns a boolean indicating whether a valid combination was provided. The code provided in Figure 2.4 looks mostly like normal OCaml code, but it is however checked not to violate any of the security types provided.

When we would have provided the signature in Figure 2.5 for checkPwd one would expect the compiler to complain about the application of a secure user record to this function in the definition of *login* in Login.fml. So it would explain how some illegal flow is derived. But instead it gives the error found in Figure 2.6.

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

Fig. 2.1: Password.fmli

```
type pwdEntry =
  {userName : string;
   password : string
  }
let pwdList =
  let jeroen : pwdEntry = {userName = "Jeroen";password = "1234"} in
  let stefan : pwdEntry = {userName = "Stefan";password = "5678"} in
  let jur : pwdEntry = {userName = "Jur";password = "9123"} in
    [jeroen;stefan;jur]
let checkPwd p uRec = if p = uRec.password then true else false
```

Fig. 2.2: Password.fml

```
flow !everyone < !root
val login : 'α string → 'α string → 'α bool
  with !everyone < 'α
```

Fig. 2.3: Login.fmli

Which tells the programmer that there is some illegal flow, and the location points of 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 how the flow was derived or what the programmer should 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 message like these are not helpful at all.

```
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. 2.4: Login.fml

```
val checkPwd : 'α string → (!everyone, !everyone) pwdEntry → 'α bool
```

Fig. 2.5: Different signature for *checkPwd*

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

Fig. 2.6: Error given when the signature from Figure 2.5 is used

2.2 Haskell Library example

The second example is a similar program but now written in Haskell using the library presented in [Russo et al., 2008]. For our example we defined a Lattice with three security levels (although only one is used explicitly), and we only used the non IO security monad. 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. In the code fragment in Figure 2.7 the username lookup and password verification functions are presented. Any declassification is done in a different library where we have access to a special function called *reveal* that removes all security information from a value.

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 (see the *verifyPassword* function for example in Figure 2.7).

```
 {-# LANGUAGE FlexibleContexts #-}
module Spwd
(
   getSpwdName,
      verifyPassword
)
where
import SecLibTypes
import Sec
import SpwdTypes
import DeclCombinators (hatch, showResult, combine)
import Data.Maybe (fromJust)
verifyPassword :: Less α R ⇒ Sec R Spwd → Sec α Pwd → Sec α Bool
verifyPassword record provided = let known = fromJust $ ((hatch (λ(_,p) → p) record)
   :: Maybe (Sec R Pwd))
   in fromJust $ showResult $ ((combine (≡) known provided)
      :: (Sec R Bool))
getSpwdName :: Sec α Usr → (Maybe (Sec R Spwd))
getSpwdName nam = findUser nam passWordFile
findUser :: Sec α Usr → [(Usr, Sec R Spwd)] → Maybe (Sec R Spwd)
findUser usr ((u,r) : xs) = let match = public $ fromJust $ showResult (matchUser usr u)
   in case match of
      True → Just r
      False → findUser usr xs
findUser _ [] = Nothing
matchUser :: Sec α Usr → Usr → Sec α Bool
matchUser usr u = fromJust $ (hatch ((≡) u) usr)
passWordFile :: [(Usr, Sec R Spwd)]
passWordFile = [("Jeroen", sec ("Jeroen", "1234")), ("Stefan", sec ("Stefan", "5678"))]
```

Fig. 2.7: Spwd.hs

Having to write explicit types at so many places makes writing a program much harder and regular type errors and security errors become harder to read. For example consider the version of *verifyPassword* in Figure 2.8, the correct signature of the function combine is as in Figure 2.9.

This function would normally be considered type correct, but instead of the expected **Sec** α `Bool` it returns **Sec** *R* `Bool`, with the wrong security type. GHC provides the error found in Figure 2.10.

This error provides quite a lot of information, and a large fragment of code. It accurately blames the expression where *combine* is used, but explains the illegal flow of *R* below α in terms of type classes (which they are) but not in terms of secure information flow. When we make a mistake that would normally be a type error we also get a message that then contains information on the security types, making it harder to read.

$$
\begin{aligned}
&\mathit{verifyPassword} :: \mathit{Less}\ \alpha\ R \Rightarrow \mathbf{Sec}\ R\ \mathit{Spwd} \to \mathbf{Sec}\ \alpha\ \mathit{Pwd} \to \mathbf{Sec}\ \alpha\ \texttt{Bool} \\
&\mathit{verifyPassword}\ \mathit{record}\ \mathit{provided} = \mathbf{let}\ \mathit{known} = \mathit{fromJust}\ \$\ ((\mathit{hatch}\ (\lambda(\_, p) \to p)\ \mathit{record}) \\
&\quad :: \mathit{Maybe}\ (\mathbf{Sec}\ R\ \mathit{Pwd})) \\
&\quad \mathbf{in}\ \mathit{combine}\ (\equiv)\ \mathit{known}\ \mathit{provided}
\end{aligned}
$$

Fig. 2.8: Wrong version of *verifyPassword*

$$
\mathit{combine} :: (\mathit{Less}\ s\ s'', \mathit{Less}\ s'\ s'') \Rightarrow (\alpha \to \beta \to C) \to \mathbf{Sec}\ s\ \alpha \to \mathbf{Sec}\ s'\ \beta \to \mathbf{Sec}\ s''\ C
$$

Fig. 2.9: Type signature of the combine function

```
Spwd.hs:17:13:
    Could not deduce (Less R a) from the context (Less a R)
      arising from a use of ‘combine’ at Spwd.hs:17:13-39
    Possible fix:
      add (Less R a) to the context of
        the type signature for ‘verifyPassword’
      or add an instance declaration for (Less R a)
    In the expression: combine (==) known provided
    In the expression:
        let
          known = fromJust
                $ ((hatch (\ (_, p) -> p) record)
          :: Maybe (Sec R Pwd))
        in combine (==) known provided
    In the definition of ‘verifyPassword’:
        verifyPassword record provided
                       = let
                           known = fromJust
                                 $ ((hatch (\ (_, p) -> ...) record)
                     :: Maybe (Sec R Pwd))
                           in combine (==) known provided
```

Fig. 2.10: Error that GHC gives when the version of *verifyPassword* from Figure 2.8 is used.

```
Spwd.hs:31:19:
    Couldn't match expected type 'Sec a Bool'
           against inferred type 'Maybe (Sec s' Bool)'
    In the expression: (hatch ((==) u) usr)
    In the definition of 'matchUser':
        matchUser usr u = (hatch ((==) u) usr)
```

Fig. 2.11: Error that GHC gives when *fromJust* is forgotten in *matchUser*.

In this particular program the type errors never really get complicated because of the simple nature of the program. In Figure 2.11 the result of a small type error is presented, it is obvious that the problem lies in the inferred *Maybe* but we also see that they differ in the security type variable which is not a problem in this case.

2.3 Discussion

Both implementations have their strengths and weaknesses. The FlowCaml approach makes programming quite easy, but can provide rather useless error messages. The Haskell library doesn't require a new language to be learned, and gives quite accurate error messages because it uses the Haskell type system. Using this library however results in much more difficult programming, as the programmer has to provide quite a lot of type signatures. And also the type error messages get less readable because all security types are in there as well (and vice versa). In this thesis we present a solution to the problem of inaccurate error messages, as in the FlowCaml example. That particular problem could have been solved by attaching program points to the information flow information. In larger programs that make use of polymorphic library functions the place where the error is found is possibly not the place where the actual problem is. In these cases the source might be found by tracing information flows and using heuristics to identify the cause(s) of the problem. The problem of obfuscated error messages is due to the fact that security levels and types are mixed, and so are the errors. This is solved by making the type system and analysis two separate processes, who each provide error messages for themselves. The system we present in this thesis does indeed provide more accurate error messages than FlowCaml, in a situation similar to the one presented in Section 2.1 our system gives the error message presented in Figure 2.12. Our sFun++ language does not have explicit types, and derives all types including the security types. The system presented in this thesis does not give any error messages that are a combination of security annotations and types. As a result our error messages are more readable than the error messages given by the Haskell library presented in Section 2.2.

```
Error in application:
"(checkPwd 1) password" at (line 13, column 9)
The function: "(checkPwd 1)"
Expected an argument protected at at most level: Low
But the given argument: "password"
Is protected at a higher level.
```

Fig. 2.12: sFunn++ error in case similar to the example from Section 2.1

3. RELATED WORK

In this chapter we discuss some previous work related to security analysis and type error messages. We begin by providing a broad overview of the work done in the field of security analysis and to point out some important principles (Section 3.1). We then discuss some work on security analysis that is more closely related to ours in Section 3.2. The main purpose of our work is not improving security analysis but providing better feedback for inconsistencies found during security analysis. A similar analysis where providing feedback was already thoroughly researched is type analysis. Although difficulties are different when providing feedback for this analysis a lot of work done in the field of providing type error message is also applicable to security analysis. We discuss the state of the art of providing type error messages in Section 3.3. The security analysis we present works on top of the type system and security annotations are part of the types. Restrictions on annotation variables are similar to restrictions on type variables. In Haskell [**?**] restrictions on type variables are formulated in terms of type classes, which are a specific form of qualified types [Jones, 1994]. In our type language restrictions on annotations are also added in the form of qualified types. We will discuss the theory of qualified types in Section 3.4.

3.1 A broad overview of work on security analysis

Sabelfeld and Myers provide an overview of work done in the field of security analysis [Sabelfeld and Myers, 2003]. A security analysis is performed to ensure program confidentiality, meaning that no secure information may leak (partially) through a non secure output. With security analysis we shall also always refer to an analysis to ensure confidentiality unless mentioned otherwise.

In earlier work on confidentiality all data was labelled with a security level. Runtime computations are augmented with these security levels and violations would be reported at run-time. This approach is somewhat time consuming. A different approach is static checking of the security restrictions while performing type checking. This approach will reject a program if one of its possible execution paths violates confidentiality. We discuss this approach more thoroughly in Section 3.2.

In Section 3 of [Sabelfeld and Myers, 2003] Sabelfeld and Myers discuss a small imperative language and its security-type system. They point out that any expression may be typed as highly secure. There is no risk in treating something to be more secure than it actually is; it will most certainly not contribute to leaking secure information. An expression can only be typed low secure if it doesn't contain any high values. Typing an expression as low while it contains highly secure values may lead to (partially) leaking secure information. In the imperative language statements are called commands, and they are executed under a certain context that describes the security state. For an explanation of the rules we refer to their paper.

3.2 Type based Security analysis

Heintze, Nevin and Riecke present a small statically typed, lambda calculus based, language for security analysis called the Slam calculus [Heintze and Riecke, 1998]. It is a call by value language. It uses two kinds of security annotations, one for direct readers and one for indirect readers. A variable that has a High secure direct annotation and a Low secure indirect annotation can be used by someone with low permissions to make decisions with. 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. There is also a construct **protect** available that allows the programmer to increase the security level of a computed value. The **protect** can not be abused for decreasing the security level as it is semantically defined to make the new security level the join of the current security level and the provided security level. In [Pottier and Simonet, 2002] an implementation of security analysis for a lightweight version of ML (called Core ML) is presented. This work forms the basis for the language used in section 2.1. The presented language features more complicated constructs as exceptions and references which both are not in our language. Core ML does not have a construct similar to **protect** like the Slam calculus has. The language construct for increasing the security level in Slam calculus is not needed in the presence of sub-typing. Furthermore it might complicate matters when using a more complicated lattice, that branches along the way. Ensuring some higher security level for a value can be done by explicitly providing this level in the type signature. If this is possible you can increase the security level; if the original security level was higher the program will not type check. In contrast, **protect** chooses the highest of the two without rejecting the program. The Core ML implementation of security analysis also features polyvariance (polymorphism at annotation level). This gives the possibility of writing only one version of a function and use it in both secure and less secure contexts. In [Abadi et al., 1999] it is proven that some analyses are forms of dependency analysis (security analysis, slicing analysis and binding time analysis for example). They present a calculus called the Dependency Core Calculus for which the type system can easily be instantiated to any dependency analysis. In [Abadi, 2007] polymorphism is added to the analysis.

Other papers focus on security analysis in imperative languages, [Volpano et al., 1996], [Volpano and Smith, 1997] and [Volpano and Smith, 2000] for example. Although these analysis do not seem to be of immediate need as we perform the analysis on a functional language, the languages presented in these papers also describe an expression language for which the rules can also be applied to expressions in functional languages.

3.3 Providing type error messages

As mentioned earlier a lot of work has been done on providing feedback for type errors. In [Haack and Wells, 2004], [Stuckey et al., 2003] and [Heeren, 2005] some strategies are discussed. Although the errors encountered during security analysis are different from type errors,we can still use a lot of knowledge from this area.

In [Heeren, 2005], a framework for type inferencing and type error reporting is presented. It discusses constraint-based type inferencing and how to solve these constraints. The inferred constraints are ordered in a tree shaped similarly to the AST. The tree is built with combinators, these combinators determine the order that the constraints are solved in. Before solving the constraint tree is flattened to a list. How the tree is flattened depends on the tree walk that determines what branches are flattened first for each combinator. The

constraint tree can also spread constraints (from their definition site, to usage sites). The use of combinators and flattening trees is also discussed in [Hage and Heeren, 2009]. When inconsistencies are encountered, heuristics are used to determine what location in the AST is blamed for this inconsistency. The result of this, an error message, is then presented to the user. The heuristics try to find the error location in multiple ways, some heuristics look into the number of error paths one constraint is in, if this number is high this makes the constraint a good candidate to be the error. Other heuristics try to find the error by trying to fix the problem through altering the AST. If successful, the fix is suggested to the user. The use of heuristics to discover errors is also discussed in [Hage and Heeren, 2006].

A whole different approach is presented in [Haack and Wells, 2004]. In the case of a type error a program slice is presented to the user. The slice contains all places in the program where changes can be made in order to fix the type error (all other places are irrelevant). It does so by assigning labels to program points with which type constraints are associated. When an error is found a minimal unsolvable constraint set is computed and based on that a program slice is created. The authors present this method as an alternative to what the TOP framework does and an improvement over regular type errors provided by compilers. We however, believe that it can also be used together with TOP, where this approach is used to narrow down the possible candidates which might cause the problem. Then the TOP approach can be used to more precisely pinpoint the probable error location if there is evidence that suggests this. If there is no such evidence this then doesn't succeed presenting the program slice with an explanation is still a good error message.

The Chameleon Type Debugger takes another approach to provide feedback on type errors [Stuckey et al., 2003]. The Chameleon Type Debugger allows the programmer to perform interactive type debugging, so that the programmer can ask for an explanation why something is wrong. Chameleon can also explain why some class constraint should be present. The explanation consists of a program fragment where the parts that lead to a conflict are underlined. It uses a constraint based type system to find the error and compute a minimal unsatisfiable subset and from this create the error message. In [Stuckey et al., 2004] some improvements over the first paper are considered, providing more accurate results. This approach can also be used complementary to the approach of Heeren and, Haack and Wells. Both Haack and Stuckey compute a minimal unsolvable constraint set, but where Haack just presents this to the programmer as a program slice, the Chameleon type error debugger allows the programmer to ask why certain parts are typed as reported.

3.4 Using Qualified Types

Qualified types are a way of expressing properties over types, it is a form of predicate logic on types. Security properties can be considered as properties of types and thus be expressed as a form of qualified types. In [Jones, 1994] qualified types are discussed. Qualified types restrict polymorphism by adding predicates to types. An example of this are the Haskell type classes. A type class can be declared with which a predicate is created. Types can be made a member of a type class by defining an instance of the type class for that particular type. Consider the following type:

$$(\equiv) :: \forall \alpha . \mathit{Eq}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \texttt{Bool}$$

The equality operator exists within the *Eq* class, its defined for all types α that satisfy the predicate *Eq* α. A type can satisfy this predicate when an instance of the class is provided for that type. During type inferencing

at sites where the $(\equiv)$ operator is used it must be proven that the type that it is instantiated to satisfies the predicate. Such proof is constructed from the type class environment, that contains information on all known instances.

Jones provides a mathematical background on qualified types, motivated by type classes. Type classes are however just a form of qualified types and other properties can also be expressed using qualified types. In [Hage et al., 2007] it is shown that usage analysis can also be expressed using qualified types, indicating that qualified types can also be used for polyvariant program analysis. In [Heeren, 2005] qualified types are also treated and it is shown how they are incorporated in the TOP framework for type classes.

4. LANGUAGE AND TYPE SYSTEM

In this chapter we introduce our source language sFun++, which is build on the Fun language presented in Chapter 3 of [Nielson et al., 1999]. sFun++ contains additional programming constructs that are necessary for security analysis; pairs and lists are also added. Then we will present the underlying type system in Section 4.3, upon which we build our security analysis in Chapter 5.

4.1 The Fun Language

The language that we will use in this thesis, is based upon the Fun language as presented in Chapter 3 of [Nielson et al., 1999]. This language is a simply typed, call-by-value (also called strict) lambda calculus. In order to make the language suitable for security analysis we extend the language in section 4.2. In this section we present the original language.

We use the following syntactic categories

$$\begin{array}{rcll} n & \in & \textbf{Nat} & \textit{natural numbers} \ , \\ b & \in & \textbf{Bool} & \textit{booleans} \ , \\ e & \in & \textbf{Exp} & \textit{expressions} \ , \\ f,x & \in & \textbf{Var} & \textit{variables} \ , \\ \oplus & \in & \textbf{Op} & \textit{binary operators} \ , \end{array}$$

The abstract syntax of the Fun language is defined as:

$$\begin{array}{rcl} e & ::= & n \mid b \mid x \mid \textbf{fn}\, x \Rightarrow e_0 \mid \textbf{fun}\, f\, x \Rightarrow e_0 \\ & \mid & e_0\, e_1 \mid \textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \\ & \mid & \textbf{let}\, x = e_0\, \textbf{in}\, e_1 \mid e_1 \oplus e_2 \end{array}$$

We have two separate constructs for function definition, $\textbf{fn}\, x \Rightarrow e_0$ defines a non-recursive function where x is a bound variable, and e_0 is the expression representing the body of the function. Recursive functions are defined by $\textbf{fun}\, f\, x \Rightarrow e_0$, where all occurrences of f in e_0 refer to $\textbf{fun}\, f\, x \Rightarrow e_0$. We do not provide syntactic sugar for functions with multiple arguments, such functions are defined by nesting **fn** constructs (for example: $\textbf{fn}\, x \Rightarrow \textbf{fn}\, y \Rightarrow e$). Function application is represented by $e_0\, e_1$, as always function application is left associative. Due to the strict semantics of our language e_1 is first fully evaluated before being passed to e_0. Non-recursive local definitions are introduced by: $\textbf{let}\, x = e_0\, \textbf{in}\, e_1$, which means that x refers to the value of e_0 in e_1.

4.2 The extended source language: sFun++

In order to deal with security analysis we extend our language with some specials purpose program constructs as well as with some more common constructs such as declarations and unary operators. We also add two sorts of value containers to our language namely pairs and lists, and some special purpose constructs to deal with them.

We add the following syntactic categories:

$$\begin{array}{lcll} p & \in & \textbf{Prog} & \textit{Program} \ , \\ d & \in & \textbf{Decl} & \textit{Declarations} \ , \\ u & \in & \textbf{Op'} & \textit{Unary operators} \ , \\ s & \in & \textbf{Sec} & \textit{Security levels} \end{array}$$

The **Prog** category ranges over programs, a program is a list of declarations. The category **Decl** is added to express declarations in a module. The **Op'** category adds unary operators. The last category **Sec** ranges over security levels, the security levels for a program form a lattice. We describe the lattice in chapter 5.

We extend our abstract syntax to:

$$\begin{array}{lcl} p & ::= & d^* \\ d & ::= & f = e_1 \\ e & ::= & n \mid b \mid x \\ & \mid & \textbf{fn}\, x \Rightarrow e_0 \mid \textbf{fun}\, f\, x \Rightarrow e_0 \\ & \mid & e_0\, e_1 \mid \textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \\ & \mid & \textbf{let}\, x = e_0\, \textbf{in}\, e_1 \mid e_1 \oplus e_2 \mid u\, e_1 \\ & \mid & \textbf{fst}\, e_1 \mid \textbf{snd}\, e_1 \\ & \mid & \textbf{null}\, e_1 \mid \textbf{hd}\, e_1 \mid \textbf{tl}\, e_1 \\ & \mid & \textbf{declassify}\, e_0\, s \mid \textbf{protect}\, e_0\, s \\ & \mid & \textbf{Cons}\, e_1\, e_2 \mid \textbf{Nil} \mid (e_1, e_2) \\ \oplus & ::= & + \mid - \mid * \mid \div \mid \% \mid \wedge \\ & \mid & \& \mid \mid \\ & \mid & < \mid > \mid =< \mid \Rightarrow \mid \equiv \mid \neq \\ u & ::= & ! \end{array}$$

The 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 module. We added two datatypes as terms namely pairs defined as (e_1, e_2), and lists which are defined by **Cons** e_1 e_2 and **Nil**. In **Cons** e_1 e_2, e_2 evaluates to a list and e_1 has to something that can be contained by that list. **Nil** represents the empty list. Elements of the two datatypes can be extracted with their destructors: **fst** and **snd** return the first and the second component of a tuple respectively; The expression **hd** (**Cons** e_1 e_2) delivers e_1 as a result, **hd Nil** is undefined. **tl** (**Cons** e_1 e_2) delivers e2, and **tl Nil** is undefined. The expression **null** (**Cons** e_1 e_2) evaluates to **true** and **null Nil** evaluates to **false**.

The two most important additions for security analysis, are the constructs **protect** $e_0\ s$ and **declassify** $e_0\ s$. The expression **protect** $e_0\ s$ is used to increase the level of protection of an expression e_0 to the 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 s' at which e_0 was already protected. The construct can never decrease the security level of an expression, a program that uses **protect** to decrease the security level is rejected.

The expression **declassify** $e_0\ s$ does exactly the opposite of **protect** $e_0\ s$; 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. We however belief that a security system without a means of (controlled) explicit declassification is useless. In other words without declassification it would not be possible to tell an unauthorised user that he or she entered the wrong password. The analysis has to ensure that there are no other ways to decrease the level of confidentiality other than **declassify** $e_0\ s$.

4.3 Underlying Type System

In this section we present the underlying type system for our language. We first introduce a monomorphic type system (section 4.3.2), then we will add polymorphism (section 4.3.3) and finally we present the changes needed to make the whole system syntax directed (section 4.3.4). We implemented a type inference system based on algorithm $\mathscr{W}$ which, for the interested reader, we included in Appendix A.

4.3.1 Monomorphic Type language

We define a small type language for our type system. We start out with a monomorphic type language with which we can type programs of our language.

The monomorphic type language is build out of the following two syntactical categories:

$$\begin{array}{lcll} \tau & \in & \textbf{Ty} & \textit{types} \\ \Gamma & \in & \textbf{TyEnv} & \textit{type environments} \end{array}$$

And our type language is defined as:

$$\begin{array}{lcl} \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{List}\,\tau \mid (\tau_1, \tau_2) \mid \tau_1 \to \tau_2 \\ \Gamma & ::= & \emptyset \mid \Gamma\,[x \mapsto \tau] \end{array}$$

A type environment Γ is a mapping from variables to a type τ. The type associated with a variable x in an environment Γ is written $\Gamma\,(x)$; It returns the type associated to the right most occurrence of x. Our types τ range over integers, booleans, lists, pairs and functions.

Typing judgements $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash n : \texttt{Int}}\ [t\text{-}int] \qquad \frac{}{\Gamma \vdash \textbf{true} : \texttt{Bool}}\ [t\text{-}true] \qquad \frac{}{\Gamma \vdash \textbf{false} : \texttt{Bool}}\ [t\text{-}false]$$

$$\frac{}{\Gamma \vdash \textbf{Nil} : \texttt{List}\ \tau}\ [t\text{-}Nil] \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{List}\ \tau}{\Gamma \vdash \textbf{Cons}\ e_1\ e_2 : \texttt{List}\ \tau}\ [t\text{-}Cons]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}\ [t\text{-}Pair] \qquad \frac{\Gamma\,(x) = \tau}{\Gamma \vdash x : \tau}\ [t\text{-}var]$$

$$\frac{\Gamma\,[x \mapsto \tau_x] \vdash e_0 : \tau_0}{\Gamma \vdash \textbf{fn}\ x \Rightarrow e_0 : \tau_x \to \tau_0}\ [t\text{-}fn] \qquad \frac{\Gamma\,[f \mapsto \tau_x \to \tau_0]\,[x \mapsto \tau_x] \vdash e_0 : \tau_0}{\Gamma \vdash \textbf{fun}\ f\ x \Rightarrow e_0 : \tau_x \to \tau_0}\ [t\text{-}fun]$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_0 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_0}\ [t\text{-}app] \qquad \frac{\Gamma \vdash e_0 : \texttt{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau}\ [t\text{-}if]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma\,[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ [t\text{-}let] \qquad \frac{\Gamma \vdash e_1 : \tau^1_\oplus \quad \Gamma \vdash e_2 : \tau^2_\oplus}{\Gamma \vdash e_1 \oplus e_2 : \tau_\oplus}\ [t\text{-}Bin\text{-}op] \qquad \frac{\Gamma \vdash e_1 : \tau^1_\oplus}{\Gamma \vdash u\ e_1 : \tau_\oplus}\ [t\text{-}Un\text{-}op]$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \tau_2)}{\Gamma \vdash \textbf{fst}\ e_1 : \tau_1}\ [t\text{-}fst] \qquad \frac{\Gamma \vdash e_1 : (\tau_1, \tau_2)}{\Gamma \vdash \textbf{snd}\ e_1 : \tau_2}\ [t\text{-}snd]$$

$$\frac{\Gamma \vdash e_1 : \texttt{List}\ \tau}{\Gamma \vdash \textbf{null}\ e_1 : \texttt{Bool}}\ [t\text{-}null] \qquad \frac{\Gamma \vdash e_1 : \texttt{List}\ \tau}{\Gamma \vdash \textbf{hd}\ e_1 : \tau}\ [t\text{-}hd] \qquad \frac{\Gamma \vdash e_1 : \texttt{List}\ \tau}{\Gamma \vdash \textbf{tl}\ e_1 : \texttt{List}\ \tau}\ [t\text{-}tl]$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{declassify}\ e\ s : \tau}\ [t\text{-}declass] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{protect}\ e\ s : \tau}\ [t\text{-}protect]$$

Fig. 4.1: Underlying monomorphic type system

4.3.2 Monomorphic type system

The typing judgements in our inference rules have the form $\Gamma \vdash e_0 : \tau$, which is read as given the type environment Γ expression e_0 can be assigned type τ. The typing rules for the monomorphic type system are presented in Figure 4.1. The rules are explained below.

The inference rules for [*t-int*] and [*t-true*] and [*t-false*] are straightforward, they type an element from n and b as `Int` and `Bool` respectively. The rule [*t-Nil*] types the empty list constructor as a List of any chosen type. The rule [*t-Cons*] proves that a list has type `List` τ if e_1 is of type τ and e_2 is of type `List` τ. The rule to type pairs, [*t-Pair*] if the elements e_1 and e_2 are typed τ_1 and τ_2 then the pair has type (τ_1, τ_2). Types for variables are contained in the type environment, therefore the typing rule [*t-var*] types a variable x to have the type that is stored in the environment for x. The rule [*t-fn*] assigns **fn** $x \Rightarrow e_0$ the type $\tau_x \to \tau_0$ if e_0 has type τ_0 when we extend the type environment with a mapping from variable x to its type τ_x. The rule for recursive function [*t-fun*] is similar but the type environment also has to be extended with a mapping from its recursive argument f to the type of the function itself. Function application, [*t-app*], requires that the type of the given

argument is equal to the argument type of the function. The rules for binary operators, [*t-Bin-op*] represents a whole family of rules, for each operator $\oplus$ a rule exists. Every binary operator has a type $\tau^1_\oplus \to \tau^2_\oplus \to \tau_\oplus$. Unary operators are typed by the rule, [*t-Un-op*]. This rule also represent a family of rules for each unary operator u, the type of such an operator is $\tau^1_\oplus \to \tau_\oplus$. Conditional expressions, **if then else**, are typed by the [*t-if*] rule. The [*t-if*] rule requires the expressions e_1 and e_2 to have equal types, and the conditional expression, e_0, to have the type `Bool`. Let bindings, type by [*t-let*], types the expression e_1 to have type τ_1, e_2 is typed τ_2 under an extended type environment Γ where x is associated with the type of e_1. The rules [*t-fst*], [*t-snd*], [*t-null*], [*t-hd*] and [*t-tl*] are all fairly straightforward. The rules for protecting and declassification, [*t-declass*], [*t-protect*] currently do nothing more than passing through the type of the expression that they are applied to.

4.3.3 Adding polymorphism

The system presented in the previous section doesn't allow us to write functions that can be applied to arguments of any type. A function such as *id* would have to be written for booleans, integers, lists of integers etc. All different versions of such a function would be identical, except for their type. In order to increase the usefulness of our language, and prevent the programmer from having to write versions of the identity function for each type, we make the type system polymorphic. Polymorphism allows us to write functions that operate on any type. The addition of polymorphism to our type system requires an extension of the syntactic categories and type language, as well as changes to the type rules.

We extend the syntactic categories with type variables and type schemes:

α	$\in$	**TyVar**	*type variables*
σ	$\in$	**TyScheme**	*type schemes*

Our type language is extended to:

$$\begin{array}{lll} \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{List}\,\tau \mid (\tau_1, \tau_2) \mid \tau_1 \to \tau_2 \mid \alpha \\ \sigma & ::= & \forall \alpha.\, \sigma \mid \tau \\ \Gamma & ::= & \emptyset \mid \Gamma\,[x \mapsto \sigma] \end{array}$$

Our types τ can now contain type variables. A type variable stands for an arbitrary type. A type scheme σ represents a type with quantification over type variables. Our type environment Γ now maps variables to type schemes. The function *ftv* (σ) returns the set of all variables that occur free (variables that are not quantified over) in σ. The definition of *ftv* (τ) is defined recursively and presented in Figure 4.2. In Figure 4.3 we define *ftv* (σ), the set of free type variables in $\forall \alpha.\, \sigma$ is equal to the free type variables in sig minus type variable α. In Figure 4.4 we present the rules that perform generalisation and instantiation, these two rules convert a type into a type scheme and vice versa. The generalisation rule quantifies over a variable, provided that the variable does not occur free in Γ. The instantiation rule removes a quantification of a type scheme, and replaces the variable that was quantified over by replacing all occurrences of the variable that was quantified over with a fresh variable.

In order to deal with polymorphic types and the redefined type environment Γ, we need to change two rules. The type environment contains type signatures, therefore the lookup of a variable in the type environment will give us a type scheme in the rule [*t-var-poly*]. Let bindings add the type associated to a type variable to the type environment, in the rule [*t-let-poly*] we now add type schemes to the type environment.

$$
\begin{array}{ll}
ftv\,(\texttt{Int}) & = \emptyset \\
ftv\,(\texttt{Bool}) & = \emptyset \\
ftv\,(\texttt{List}\ \tau) & = ftv\,(\tau) \\
ftv\,((\tau_1, \tau_2)) & = ftv\,(\tau_1) \cup ftv\,(\tau_2) \\
ftv\,(\tau_1 \to \tau_2) & = ftv\,(\tau_1) \cup ftv\,(\tau_2) \\
ftv\,(\alpha) & = \{\alpha\}
\end{array}
$$

Fig. 4.2: Free type variables in types

$$ftv\,(\forall \alpha.\, \sigma) = ftv\,(\sigma) \setminus \{\alpha\}$$

Fig. 4.3: Free type variables in type schemes

Typing judgements $\boxed{\Gamma \vdash e : \sigma}$

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

Fig. 4.4: Generalisation and Instantiation rules

Typing judgements $\boxed{\Gamma \vdash e : \sigma}$

$$\frac{\Gamma\,(x) = \sigma}{\Gamma \vdash x : \sigma}\ [t\text{-}var\text{-}poly] \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma\,[x \mapsto \sigma] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}\ [t\text{-}let\text{-}poly]$$

Fig. 4.5: Polymorphic var and let rules

4.3.4 Syntax directed Type System

The extension to our type system presented in the previous section doesn't allow us to straightforwardly infer the type for a given program. In this section we will present new rules that allow us to deterministically infer types guided by the structure of the program. We call such a type system a syntax directed type system. Such a system is desirable as it allows us to rewrite it to an inference algorithm easily.

We merge the four rules presented in the previous section into two rules, that can replace the rules [*t-var*] and [*t-let*] in our monomorphic type system (Figure 4.1). The type system uses regular types in all rules, however a variable that is typed through the environment would in the non syntax directed system be assigned a type scheme. This mismatch can be made up for by fully instantiating the type scheme from the type environment. We therefore merge the rule [*t-var-poly*] with [*t-inst*]. The new rule, [*t-var-poly-syntax*], instantiates the type schemes associated with the variable directly, resulting in a type. In the presented monomorphic type system the only places where a variable with its associated type are added to the environment are the rules [*t-fn*], [*t-fun*] and [*t-let*]. All of these three places would thus seem a good place to generalise over a type. However

a type is a valid type scheme, it is a type scheme without any quantifications, so it is not necessary to do so. In fact generalising over the types that are added to the type scheme in [*t-fn*] and [*t-fun*] would be wrong, a given argument will not change it's type within a functions body. A recursive function also cannot change the type it was instantiated for when making a recursive call. At a let binding however it does make sense to generalise the type of expression e_1, the variable x can be used in different contexts within e_2. Therefore the rules [*t-let-poly*] and [*t-gen*] are merged as well. The new rule [*t-let-poly-syntax*] generalises the inferred type for e_1 and inserts the resulting type scheme into the type environment. We present the rules [*t-var-poly-syntax*] and [*t-let-poly-syntax*] in Figure 4.6. In Figure 4.7 we present the *inst* and *gen* functions used by the type rules. The *inst* function fully instantiates a type scheme. The *gen* function quantifies over all variables in τ that do not occur free in Γ.

Typing judgements $\boxed{\Gamma \vdash e : \sigma}$

$$\frac{\Gamma\,(x) = \sigma}{\Gamma \vdash x : \mathit{inst}\,(\sigma)}\ [\textit{t-var-poly-syntax}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma\,[x \mapsto \mathit{gen}\,(\tau_1)] \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ [\textit{t-let-poly-syntax}]$$

Fig. 4.6: Syntax directed polymorphic var and let rules

$$\begin{aligned}
&\mathit{inst} :: \textbf{TyScheme} \to \textbf{Ty}\\
&\mathit{inst}\,(\forall \alpha_1 \ldots \alpha_n.\,\tau) = [\alpha_1 \mapsto \alpha_1{}']\ldots[\alpha_n \mapsto \alpha_n{}']\ \tau\\
&\quad \textbf{where}\ \alpha_1{}' \ldots \alpha_n{}'\ \textit{are fresh}\\
&\mathit{gen} :: \textbf{TyEnv} \to \textbf{Ty} \to \textbf{TyScheme}\\
&\mathit{gen}\,(\Gamma, \tau) = \forall \alpha_1 \ldots \alpha_n.\,\tau\\
&\quad \textbf{where}\ \{\alpha_1 \ldots \alpha_n\} = \mathit{ftv}\,(\tau) - \mathit{ftv}\,(\Gamma)
\end{aligned}$$

Fig. 4.7: Instantiation and Generalisation functions

5. SECURITY TYPE SYSTEM

In this chapter we will introduce our security analysis. We build upon the type system presented in Chapter 4. Our security type system requires some new constructs in our type language, these new constructs are introduced in Section 5.1. The security levels are modelled in a lattice, which are discussed in Section 5.2. We present the security type system in two steps, first a non syntax directed version in Section 5.3. In Section 5.4 we change this system into a syntax directed system.

5.1 Extended type language

We implement the security analysis on top of the type system. We do this by adding information about security levels of expressions to their types, we call these pieces of additional information annotations. The security analysis we implement is polyvariant, meaning that it is polymorphic in the annotations. The relation between annotation variables and restrictions on them are expressed as constraints. These constraints are also added to our types, we do this in the style of qualified types. The idea of qualified types is explained by Jones in [Jones, 1994], and is also used for type classes in Haskell (which represent restrictions on type variables). All this requires some new syntactical categories and an extended type language. We add the following new syntactical categories:

β	$\in$	**AnnVar**	*annotation variables*
ρ	$\in$	**Qualified Types**	*qualified types*
π	$\in$	**Constr**	*constraints*
l	$\in$	**Levels**	*Security levels*
φ	$\in$	**Ann**	*Security annotations*
C	$\in$	**Constraints**	*Constraint set*

The category **AnnVar** of annotation variables is similar to the category **TyVar** of type variables. We point out that these two categories are completely disjoint. The category **Qualified Types** of qualified types ranges over annotated types restricted by constraints. The category **Constr** ranges over all constraints, we only have one sort of constraint namely $(\varphi_1 \sqsubseteq \varphi_2)$. The category **Levels** ranges over all security levels, all security levels for a program together form a lattice which we will describe in Section 5.2. The category **Ann** ranges over all security annotations, which is the union of the categories **Levels** and **AnnVar**. The last category **Constraints** ranges over all sets of constraints.

Top level annotations are not contained in type schemes, but stored separately in the type environment. As a consequence 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 us to lose some expressiveness, we argue that we do not due to the presence of sub-effecting. When we add the type of a let bound value to the type environment we simply choose to default the top level annotation to the lowest possible value, which will be $\bot$.

$$
\begin{array}{lcl}
\tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{List}\ \tau^{\varphi} \mid (\tau_1{}^{\varphi}, \tau_2{}^{\varphi}) \mid \tau_1{}^{\varphi} \to \tau_2{}^{\varphi} \mid \alpha \\
\varphi & ::= & \beta \mid l \\
\pi & ::= & \varphi_1 \sqsubseteq \varphi_2 \\
\rho & ::= & C \Rightarrow \tau \mid \tau \\
\sigma & ::= & \forall\alpha.\sigma \mid \forall\beta.\sigma \mid \exists\beta.\sigma \mid \rho \\
\Gamma & ::= & \emptyset \mid \Gamma\,[x \mapsto (\sigma, \varphi)] \\
C & ::= & \emptyset \mid \{\pi_1, \ldots, \pi_n\} \cup C
\end{array}
$$

In contrast to FlowCaml [Pottier and Simonet, 2003] we can have annotations on pairs. In FlowCaml these annotations were omitted because one would not learn anything about the contents or structure of a pair by protecting it. We agree that one can not learn anything about the structure of a pair or its content but for consistency with other datatypes we choose to have annotations on pairs as well. Our inference algorithm will generate qualified types with constraints which contain variables that do not occur in the type itself. We cannot ignore these variables as they do play a role in the type. Over these sorts of types we which to quantify existentially. In section 6.4 we will discuss this phenomenon in more detail.

5.2 Lattices

The security levels used in our analysis are defined in a lattice $\mathscr{L}$. We require the security lattice $(\mathscr{L}, \sqsubseteq)$ to be finite and have a bottom element. According to Davey and Priestley this implies that the lattice is complete (corollary 2.25 Chapter 2 of [Davey and Priestley, 1990]). They also prove that completeness of a lattice $\mathscr{L}$ implies that $x \bigsqcup y$ and $x \bigsqcap y$ exist for all $x, y \in \mathscr{L}$ (definition 2.4, Chapter 2). From that same chapter it follows that every finite lattice with a bottom element adheres to the Ascending Chain Condition (ACC). The ACC property gives us the guarantee that the work list algorithm presented by Nielson, Nielson and Hankin in chapter 6 of [Nielson et al., 1999] terminates and finds a least fix point. We will use such an algorithm in Section 6.4.

5.3 Inference rules

In Figure 5.1 the non syntax directed rules for type based security analysis are presented. The rules all follow the pattern $\Gamma, C \vdash e : \sigma^{\varphi}$, which is read as under the type environment Γ and constraint environment C expression e is typed σ protected at level φ. The constraint environment contains all relation between security levels from the lattice as well as constraints that are assumed to hold.

The rules [*t-int*], [*t-true*], [*t-false*] all type an element at their respective type. We point out that the φ in these rules actually imply that from each rule a whole family of typing rules can be derived, where φ is replaced by an element of the security lattice. Lists are typed by the rules [*t-Nil*] and [*t-Cons*]. Lists have two security annotations, the annotation on the element type is meant to protect the contents of the list at that particular level. The top level annotation on a list protects the list structure. These two security levels do not have to be equal: a structurally secure list can very well contain non-secure elements and vice versa. All elements of the list, however, have to be of the same type and annotations. The rule for pairs, [*t-Pair*], is straightforward. A variable lookup in the rule [*t-var*] results in a tuple consisting of a type and an annotation, the variable then gets the type σ^{φ}. The rules for function abstraction, [*t-fn*] and [*t-fun*], are also straightforward.

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 itself is the least upper bound of the functions top level annotation and its result annotation. The reason for this is that the function abstraction itself could have been the result of a secure computation and its result might reveal something about that secure computation, or its body could have been a secure computation. Function declared at top level are assigned a top level annotation *bottom*; For uses of those functions the least upper bound is thus solely determined by the results annotation.

Operators can be considered to be primitive functions, which like regular functions have a type. Just like any other type the types of operators have to be extended with security annotations. In the discussion of the rule of [*t-app*] we discussed that the top level annotation of a function can also influence the annotation on the result of the application. For operators such behaviour is not desirable as all operators and their inner workings are known and not considered to be secure. Therefor $\bot$ is used as the top level annotation for all operators. The type of the plus operator in our system is: $(+) :: \texttt{Int@}\beta \rightarrow^{\bot} \texttt{Int@}\beta \rightarrow^{\bot} \texttt{Int@}(\beta \cup \bot)$. However $(\beta \cup \bot)$ is by definition equal to β. The type can thus be simplified to: $(+) :: \texttt{Int@}\beta \rightarrow^{\bot} \texttt{Int@}\beta \rightarrow^{\bot} \texttt{Int@}\beta$. From here on the rules [*t-Bin-op*] and [*t-Un-Op*] follow trivially.

The rules [*t-if*] and [*t-let*] are both straightforward. As we discussed earlier we agree with Pottier that we can learn nothing from the structure of a pair. Nevertheless we use the least upper bound over the top level annotation on the pair (its structural annotation) and the element we wish to retrieve from the pair in the rules [*t-fst*] and [*t-snd*]. In general one can learn something about the structure of a datatype, therefor we decided to treat pairs like we would have treated any other datatype.

The structure of the list, its length, can be classified information. This security property is represented by the top level annotation of the list. The expression **null** e_1 determines whether e1 is an empty list, so we learn something about the structure of the list. The rule [*t-null*] thus states that the resulting boolean is protected at the same level as the lists structure. Successful usage of **hd** and **tl** reveals us that the list is non empty, these constructs are handled by the rules [*t-hd*] and [*t-tl*] respectively. Therefore the results of both are influenced by the structural annotation of the list. The **hd** construct reveals an element of the list, this element has to be protected at least as high as the element annotation and the structural annotation.

Protecting and declassification is done by the the **protect** and **declassify** constructs. Their influence on the security levels are expressed in the rules [*t-declass*] and [*t-protect*]. The expression **protect** e_0 φ_0 is protected at level φ_0, with one condition, that e_0 is protected at a security level φ that is as secure as φ_0. Declassification does exactly the opposite, it lowers the level of security.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5.1: Non-syntax directed rules for security analysis

The requirement of all annotation in the premises being equal is a bit harsh and doesn't make the system very flexible. This can be circumvented by sub-effecting, subeffecting allows us to increase the level of protection for an expression whenever this happens to be necessary. Sub-effecting is added through the rule of [*t-sub*]. Increasing the security level of an expression is of course perfectly safe, the other way might lead to inappropriate use of data and is therefore not allowed.

The rules for generalisation and instantiation, [*t-gen*] and [*t-ins*], are similar to the rules presented in Section 4.3.3. The rules for generalising over and instantiation of annotations presented in the rules [*t-ann-gen*] and [*t-ann-ins*] are also similar to normal generalisation and instantiation but work on annotation variables instead of type variables. The rule [*t-ann-gen*] uses the function $fav(x)$, this function is similar to the function $ftv(x)$ that returned all free type variable in x. The function *fav* ranges over anything that contains types and annotations and returns the set of all free annotation variables in its argument.

Qualification ([*t-qual*]) and resolution ([*t-res*]) are used to add and remove constraints to/from the context of the type. Through qualification it is verified that if e can be assigned the type ρ when π is assumed to hold, then π can be added to the context of ρ. Resolution states that the assumption π can be discarded from the context of the type of e when π holds under C.

In Figure 5.2 some rules for reasoning and proving constraints are provided. The rule [*c-in*] states that if π is in C then π holds. Transitivity is provided through the rule of [*c-trans*], and reflexivity through [*c-reflex*]. [*c-bot*] and [*c-top*] state that any φ is above $\bot$ and below $\top$.

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

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

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

Fig. 5.2: Rules on constraints

5.4 Syntax directed inference rules

The inference rules presented in the previous section are not suitable for use in a type inference algorithm. In this section the system is changed into a syntax directed system of inference rules. The rules [*t-ann-gen*], [*t-gen*], [*t-ann-ins*], [*t-ins*], [*t-qual*], [*t-res*] and [*t-sub*] can all be included in other rules. The rules for proving constraints can also be included in other rules.

In Figure 5.3 the syntax directed inference system is listed. All rules that do not occur in the listing, except for those rules mentioned earlier, have not changed. Our type system features both subeffecting and polyvariance. Subeffecting allows weakening of the annotation, in this case meaning that the security level is increased. The non syntax directed system has a rule for sub-effecting [*t-sub*]. Polyvariance means that we have annotation variables. This allows us to write one function that can be used in different contexts.

For example the identity function can be used with arguments protected at different security levels. At the places were the identity function is used the variable is instantiated to some annotation level (or some variable). Because we have both polyvariance and subeffecting we need to add constraints to the types to deal with subeffecting. We choose to add these constraints to our type system in the style of Qualified Types [Jones, 1994]. This means that the inferred constraints are added to the type as predicates. When the type is then instantiated we require that the chosen values for the annotation variables satisfy the constraints.

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

$$\frac{\Gamma, C \vdash e_1 : \tau^{\varphi_1} \quad \Gamma, C \vdash e_2 : (\texttt{List}\ \tau^{\varphi_2})^{\varphi_3} \quad C \vdash \varphi_2 \sqsubseteq \varphi_1 \quad C \vdash \varphi_3 \sqsubseteq \varphi}{\Gamma, C \vdash \mathbf{Cons}\ e_1\ e_2 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}\ [t\text{-}Cons]$$

$$\frac{\Gamma(x) = (\sigma, \varphi) \quad C \vdash \varphi \sqsubseteq \varphi_1 \quad inst(\sigma) = C' \Rightarrow \tau}{\Gamma, C \cup C' \vdash x : \tau^{\varphi_1}}\ [t\text{-}var]$$

$$\frac{\Gamma, C \vdash e_1 : {\tau_2}^{\varphi_2} \rightarrow^{\varphi} {\tau_0}^{\varphi_0} \quad \Gamma, C \vdash e_2 : {\tau_2}^{\varphi_3} \quad C \vdash \varphi_3 \sqsubseteq \varphi_2 \quad C \vdash \varphi \sqsubseteq \varphi_4 \quad C \vdash \varphi_0 \sqsubseteq \varphi_4}{\Gamma, C \vdash e_1\ e_2 : {\tau_0}^{\varphi_4}}\ [t\text{-}app]$$

$$\frac{\Gamma, C \vdash e_0 : \texttt{Bool}^{\varphi_0} \quad \Gamma, C \vdash e_1 : \tau^{\varphi_1} \quad \Gamma, C \vdash e_2 : \tau^{\varphi_2} \quad C \vdash \varphi_0 \sqsubseteq \varphi \quad C \vdash \varphi_1 \sqsubseteq \varphi \quad C \vdash \varphi_2 \sqsubseteq \varphi}{\Gamma, C \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau^{\varphi}}\ [t\text{-}if]$$

$$\frac{\Gamma, C \vdash e_1 : \tau^{1\,\varphi_1}_{\oplus} \quad \Gamma, C \vdash e_2 : \tau^{2\,\varphi_2}_{\oplus} \quad C \vdash \varphi_1 \sqsubseteq \varphi \quad C \vdash \varphi_2 \sqsubseteq \varphi}{\Gamma, C \vdash e_1 \oplus e_2 : {\tau_{\oplus}}^{\varphi}}\ [t\text{-}Bin\text{-}op]$$

$$\frac{\Gamma, C \vdash e_1 : \tau^{1\,\varphi_1}_{\oplus} \quad C \vdash \varphi \sqsubseteq \varphi_1}{\Gamma, C \vdash u\ e_1 : {\tau_{\oplus}}^{\varphi}}\ [t\text{-}Un\text{-}op]$$

$$\frac{\Gamma, C \vdash e_1 : ({\tau_1}^{\varphi_1}, {\tau_2}^{\varphi_2})^{\varphi} \quad C \vdash \varphi \sqsubseteq \varphi_3 \quad C \vdash \varphi_1 \sqsubseteq \varphi_3}{\Gamma, C \vdash \mathbf{fst}\ e_1 : {\tau_1}^{\varphi_3}}\ [t\text{-}fst]$$

$$\frac{\Gamma, C \vdash e_1 : ({\tau_1}^{\varphi_1}, {\tau_2}^{\varphi_2})^{\varphi} \quad C \vdash \varphi \sqsubseteq \varphi_3 \quad C \vdash \varphi_2 \sqsubseteq \varphi_3}{\Gamma, C \vdash \mathbf{snd}\ e_1 : {\tau_2}^{\varphi_3}}\ [t\text{-}snd]$$

$$\frac{\Gamma, C \vdash e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi_2} \quad C \vdash \varphi_2 \sqsubseteq \varphi}{\Gamma, C \vdash \mathbf{null}\ e_1 : \texttt{Bool}^{\varphi}}\ [t\text{-}null]$$

$$\frac{\Gamma, C \vdash e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi_2} \quad C \vdash \varphi_2 \sqsubseteq \varphi \quad C \vdash \varphi_1 \sqsubseteq \varphi}{\Gamma, C \vdash \mathbf{hd}\ e_1 : \tau^{\varphi}}\ [t\text{-}hd] \qquad \frac{\Gamma, C \vdash e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi_2} \quad C \vdash \varphi_2 \sqsubseteq \varphi}{\Gamma, C \vdash \mathbf{tl}\ e_1 : (\texttt{List}\ \tau^{\varphi_1})^{\varphi}}\ [t\text{-}tl]$$

$$\frac{\Gamma, C' \vdash e_1 : {\tau_1}^{\varphi_1} \quad (C'', \sigma) = gen(C', \tau_1) \quad \Gamma[x \mapsto (\sigma, \varphi_1)], C \cup C'' \vdash e_2 : {\tau_2}^{\varphi_2} \quad C \cup C'' \vdash (\varphi_2 \sqsubseteq \varphi_3)}{\Gamma, C \cup C'' \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : {\tau_2}^{\varphi_3}}\ [t\text{-}let]$$

Fig. 5.3: Syntax directed security analysis

The rule [*t-sub*] from the previous section is inserted into most type rules. We shall not treat all these rules separately, but the general pattern is that the top level annotation of an expression is required to be protected at a "higher" level than its relevant sub-expressions. With relevant sub-expressions we mean only those sub-

expressions that contribute to the end result of the expression. In the rule [*t-fst*] for example the annotation of the second element of the pair does not contribute to the result of **fst** and therefore we do not generate any constraint for it.

Most interesting are the change to the rules [*t-var*] and [*t-let*]. The rules [*t-ann-ins*], [*t-ins*] and [*t-res*] are merged into [*t-var*]. The type environment Γ is a mapping from variables to a pair consisting of a type scheme and an annotation. The type scheme is instantiated by the function *inst*. The function *inst* is presented in Figure 5.4. The *inst* function replaces all variables (both type and annotation variables) that are quantified over by fresh variables, and results in a qualified type. The constraints from the qualified type are added to the constraint environment C. We do not require the variable to be used at exactly the security level that was associated with it in the type environment, but at a higher or equal security level (sub-effecting).

The rules [*t-ann-gen*], [*t-gen*] and [*t-qual*] are merged into [*t-let*]. The let rule adds the type of e_1 to the type environment Γ under which e_2 is typed. The type that is inferred for e_1, τ_1 is only valid when all constraints in C' are proven to hold. This means that the constraint in C' are predicates to the type τ_1. The generalisation function *gen* receives both the constraint environment and the inferred type as an argument (we also pass the type environment Γ and annotation φ as an argument to the generalisation function) and returns a type scheme and a new constraint environment. The generalisation function is described in Figure 5.4. The simplify function simplifies the constraint environment, it removes all trivial constraints and checks whether the set is satisfiable. We will describe the exact working of the simplify function in Section 6.4. The simplify function returns a pair of constraint environments, one environment with all constraints related to the type τ, and one environment with all other constraints. We do not wish to generalise over the 'other' constraints. The *gen* function universally quantifies over all type and annotation variables that occur free in τ but not in Γ and φ. It is however very likely that not all annotation variables that occur in C' also occur in τ. These variables are called pseudo active. Pseudo active variables are a result of the inference process and describe the relation between active variables (variables that do occur in τ). We do wish to quantify over these variables. Universal quantification however seems inappropriate as it is not possible to assign any value to them. These variables can only be assigned a value once the universally quantified variables have been instantiated so that there still exist values for these variables such that all constraints can be satisfied. It therefor seems to be natural to quantify existentially over these variables. The constraints that we do not generalise over are passed on.

$$
\begin{aligned}
&\mathit{inst} :: \mathbf{TyScheme} \rightarrow \mathbf{Qualified\ Types} \\
&\mathit{inst}\ (\forall \alpha_1 .. \alpha_n . \forall \beta_1 .. \beta_m . \rho) = [\alpha_1 \mapsto \alpha_1{}'] \ldots [\alpha_n \mapsto \alpha_n{}'] \, [\beta_1 \mapsto \beta_1'] \ldots [\beta_m \mapsto \beta_m'] \, \rho \\
&\quad \mathbf{where}\ \alpha_1{}' \ldots \alpha_n{}'\ \mathit{are\ fresh} \\
&\qquad \beta_1' \ldots \beta_m'\ \mathit{are\ fresh} \\
&\mathit{gen} :: \mathbf{TyEnv} \rightarrow \mathbf{Ann} \rightarrow \mathbf{Ty} \rightarrow \mathbf{Constraints} \rightarrow (\mathbf{Constraints}, \mathbf{TyScheme}) \\
&\mathit{gen}(\tau, C)_{(\Gamma,\varphi)} = \forall \alpha_1 \ldots \alpha_n . \forall \beta_1 \ldots \beta_m . \exists \beta_{m+1} \ldots \beta_p . (C'', C' \Rightarrow \tau) \\
&\quad \mathbf{where}\ (C', C'') = \mathit{simplify}\ C \\
&\qquad\qquad \{\alpha_1, \ldots, \alpha_n\} = \mathit{ftv}\ (\tau) - \mathit{ftv}\ (\Gamma) \\
&\qquad\qquad \{\beta_1, \ldots, \beta_m\} = \mathit{fav}\ (\tau) - \mathit{fav}\ (\Gamma) - \mathit{fav}\ (\varphi) \\
&\qquad\qquad \{\beta_{m+1}, \ldots, \beta_p\} = \mathit{fav}\ (C') - \mathit{fav}\ (\tau) - \mathit{fav}\ (\Gamma) - \mathit{fav}\ (\varphi)
\end{aligned}
$$

Fig. 5.4: Instantiation and generalisation

6. INFERENCE ALGORITHM

In this chapter an algorithmic version of the inference system from Section 5.4 is presented. The inferencer is based on Algorithm $\mathscr{W}$ [Damas and Milner, 1982]. We will start off by presenting substitutions (Section 6.1). We will discuss our unification algorithm in Section 6.2. In Section 6.3 we present our inference algorithm.

6.1 Substitutions

During type inferencing a variable is introduced whenever a type or annotation cannot be determined, as soon as it can be determined a substitution, represented by θ, is generated that replaces the type or annotation variable by the actual type or annotation.

A substitution is a pair of finite mappings from type variables to types, and 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 $\theta\ \tau$. Two substitutions can be combined by substitution composition. Application of a composed substitution is defined in Figure 6.1.

$$(\theta_2 \circ \theta_1)\ (\tau) \equiv \theta_2\ (\theta_1\ (\tau))$$

Fig. 6.1: Substitution composition

The empty substitution is represented by *Id*. Type variables that are not in the domain of the substitution are not changed when applied to a substitution.

Substitutions are applied recursively to a type τ as presented in Figure 6.2. Application of substitutions on annotation is presented in Figure 6.3.

Substitutions can be applied to type schemes, and qualified types as well. We define substitutions on type schemes in Figure 6.4. We introduce a new syntactic category all variables:

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

Bound variables can not be substituted, therefore we introduce a new sort of substitution θ/ζ that prevents the bound variable ζ from being substituted by θ. In Figure 6.5 we present substitutions on qualified types.

Substitutions can also be applied to type environments (Figure 6.6). A type environment is defined as a list of mappings from a program variable to a pair of a type and annotation. A substitution updates each of the mappings in the environment, so that all variables map to substituted types and annotations. In Figure 6.7 we define substitutions on constraints and constraint environments.

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

Fig. 6.2: Substitution application on types

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

Fig. 6.3: Substitution application on annotations

$$
\begin{array}{ll}
\theta\ (\forall\alpha.\sigma) & = \forall\alpha.((\theta/\alpha)\ \sigma) \\
\theta\ (\forall\beta.\sigma) & = \forall\beta.((\theta/\beta)\ \sigma) \\
\theta\ (\exists\beta.\sigma) & = \exists\beta.((\theta/\beta)\ \sigma) \\
\multicolumn{2}{l}{\textbf{where}} \\
\multicolumn{2}{l}{\quad (\theta/\zeta)\ \zeta_1 = \zeta_1 \quad \textbf{if}\ \zeta \equiv \zeta_1} \\
\multicolumn{2}{l}{\qquad\qquad\ \theta\ \zeta_1\ \textbf{otherwise}}
\end{array}
$$

Fig. 6.4: Substitution application on type schemes

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

Fig. 6.5: Substitution application on qualified types

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

Fig. 6.6: Substitution application on type environments

6.2 Unification

During type inferencing two types are often required to be equal. To ensure that two types are equal we use a unification algorithm $\mathscr{U}$. The unification algorithm is defined both for annotated types and annotations, the result of unification is a substitution. In Figure 6.8 we present unification over types. Unification of

$$\theta\, C = \{\theta\, \pi \mid \pi \in C\}$$
$$\theta\, (\varphi_1 \sqsubseteq \varphi_2) = (\theta\, \varphi_1 \sqsubseteq \theta\, \varphi_2)$$

Fig. 6.7: Substitution application on constraints and constraint environments

two `Int`s or `Bool`s results in the empty substitution *Id*. Unification over container types (lists, pairs and functions) requires both containers to be equal, and that unification of its element types and annotations succeed. The resulting substitution of a unification is always propagated to the remaining elements before they are unified. The end result of the unification is the composition of the two results. Pairs and functions are unified similarly. 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), if the variable does occur in the type unification fails. Unification also fails whenever two different top level type constructors are unified.

In Figure 6.9 the unification algorithm for security annotations is listed. Only two annotation variables can be unified. Our inferencing algorithm will always assign an annotation variable to an expression, the relation to a security level is expressed by a constraint. Annotations for top level bindings are always security levels. Our var rule introduces a fresh variable and generates a constraint that restricts the variable to be at least as secure as the security level in the type environment. This ensures that the invariant of always assigning any expression an annotation variable as annotation.

6.3 Security Analysis Inference Algorithm

With the inference system presented in Section 5.4 as a formal specification we define our inference algorithm $\mathcal{W}_{sec}$. The algorithm infers principal type-schemes, security annotations and constraints on security constraints. The algorithm is based on algorithm $\mathcal{W}$ [Damas and Milner, 1982]. The algorithm is defined by structural induction on e and has type:

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

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. Sub-effecting is handled through the constraints and verified by our constraint solver described in Section 6.4. Our algorithm is listed in the Figures 6.10, 6.11 and 6.12. In Section 5.4 we specified the rules of our algorithm; in this section we discuss how we made an algorithm from the specification. Most alternatives follow straightforwardly from the inference rules or from other alternatives. We will therefore only discuss those rules that are special.

The alternatives that type natural numbers and booleans return a type `Int` and `Bool` respectively, a fresh annotation variable, the empty substitution and an empty constraint set. An annotation variable is generated because there is no information that insists on a specific security level at this point. The alternative that types

$$
\begin{array}{llll}
\mathit{unifyTy} :: \textbf{Type} \to & \textbf{Type} \to & & \textbf{Substitution} \\
\mathit{unifyTy}\ \ \texttt{Int} & \texttt{Int} & = & \mathit{Id} \\
\mathit{unifyTy}\ \ \texttt{Bool} & \texttt{Bool} & = & \mathit{Id} \\
\mathit{unifyTy}\ \ \texttt{List}\,\tau_1{}^{\varphi_1} & \texttt{List}\,\tau_2{}^{\varphi_2} & = & \textbf{let}\ \theta_1 = \mathit{unify}\ \tau_1\ \tau_2 \\
 & & & \quad\ \ \theta_2 = \mathit{unify}\ (\theta_1\ \varphi_1)\ (\theta_1\ \varphi_2) \\
 & & & \textbf{in}\ \ \theta_2 \circ \theta_1 \\
\mathit{unifyTy}\ \ (\tau_{11}{}^{\varphi_{11}}, \tau_{12}{}^{\varphi_{12}}) & (\tau_{21}{}^{\varphi_{21}}, \tau_{22}{}^{\varphi_{22}}) & & \\
 & & = & \textbf{let}\ \theta_1 = \mathit{unifyTy}\ \tau_{11}\ \tau_{21} \\
 & & & \quad\ \ \theta_2 = \mathit{unifyAnn}\ (\theta_1\ \varphi_{11})\ (\theta_1\ \varphi_{21}) \\
 & & & \quad\ \ \theta_3 = \mathit{unifyTy}\ (\theta_2 \circ \theta_1\ \tau_{12})\ (\mathit{apply}\ (\theta_2 \circ \theta_1)\ \tau_{22}) \\
 & & & \quad\ \ \theta_4 = \mathit{unifyAnn}\ (\theta_3 \circ \theta_2 \circ \theta_1\ \varphi_{12})\ (\theta_3 \circ \theta_2 \circ \theta_1\ \varphi_{22}) \\
 & & & \textbf{in}\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \\
\mathit{unifyTy}\ \ \tau_{11}{}^{\varphi_{11}} \to \tau_{12}{}^{\varphi_{12}} & \tau_{21}{}^{\varphi_{21}} \to \tau_{22}{}^{\varphi_{22}} & & \\
 & & = \textbf{let} & \quad\ \ \theta_1 = \mathit{unifyTy}\ \tau_{11}\ \tau_{21} \\
 & & & \quad\ \ \theta_2 = \mathit{unifyAnn}\ (\theta_1\ \varphi_{11})\ (\theta_1\ \varphi_{21}) \\
 & & & \quad\ \ \theta_3 = \mathit{unifyTy}\ (\theta_2 \circ \theta_1\ \tau_{12})\ (\theta_2 \circ \theta_1\ \tau_{22}) \\
 & & & \quad\ \ \theta_4 = \mathit{unifyAnn}\ (\theta_3 \circ \theta_2 \circ \theta_1\ \varphi_{12})\ (\theta_3 \circ \theta_2 \circ \theta_1\ \varphi_{22}) \\
 & & \textbf{in} & \quad\ \ \theta_4 \circ \theta_2 \circ \theta_2 \circ \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. 6.8: Type Unification

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

Fig. 6.9: Annotation Unification

the empty list, returns the type $\texttt{List}\ \alpha^{\beta_1}$. The element type cannot be determined at this point hence the introduction of the type variable. The annotation variable is generated because there is no information at this point that insists on a specific security level.

The alternatives for function abstraction and recursive function abstraction are a bit more involved. The type of the function argument is not known at this point, we therefore type the function body with a type environment extended with the variable associated with a fresh type and annotation variable. During type inference of the body information about the actual type may be discovered, this then results in a substitution.

The returned substitution is applied to the domain part of the function type. The returned annotation is a fresh variable as it cannot be determined yet at what level the function should be protected. In recursive functions the type of the recursive call also needs to have a type. The argument of the recursive call is of the same type as the function's argument. The result type and annotation are fresh variables. After inferring the type and annotation of the function body these variables are unified with the type and annotation of the function body.

The alternative for typing variables straightforwardly follows the original inference rule. The type scheme and annotation of the variable is retrieved from in the environment. The type scheme is instantiated as described in Section 5.4. The resulting qualified type is then divided into a type and a constraint environment. 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.

The function application alternative is a bit more involved. The type of e_2 has to match the type expected by the function. This type can however not easily be extracted from the inferred type of e_1 and the result type of the whole application can also not easily be extracted from the type of e_1. The types cannot be easily extracted from the type of e_1 because at this point it is not yet known whether the inferred type τ_1 is indeed a function type. We introduce fresh variables for the result type and annotation as well as for the argument annotation (sub-effecting). These variables are combined in a function type together with the inferred type of e_2. This type is then unified with the inferred type of e_1. If unification is successful the type of e_2 matches the expected argument type, and the substitution that is returned by the unification maps the introduced type and annotation variables to the values that were inferred for e_2.

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. The result is generalised, as explained in Figure 5.4, resulting in a type scheme and a set of constraints. The returned constraints contain annotation variables that occur unbound in the type environment and therefore are passed on. The type scheme and annotation variable are added to the type environment under which e_2 will be typed. The resulting type of the whole let binding is the type of e_2. Its annotation is the fresh variable β which is constrained to be at least as secure as φ_2.

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.

6.4 Constraint Solving

The inference system presented in the previous section generates constraints. Our generalisation algorithm presented in Figure 5.4 uses a function *simplify* that simplifies, partitions, and verifies satisfiability of a set of constraints. In this section we will discuss the workings of this function. The simplify function is defined in Figure 6.13. The function first determines the set of variables that cannot be instantiated. It then computes the range of allowed security levels for each annotation variable. After that it verifies whether the constraint set is satisfiable. The satisfiability check results in a substitution that replaces the variables that may be instantiated with the optimal (lowest possible) security level. Finally the constraint set is partitioned. The

constraints are partitioned so that those that restrict the types security annotations can be added to the type as qualifiers. We discuss all these steps below in more detail.

A constraint set is simplified by instantiating as many annotation variables as possible. Instantiation of a variable can limit the use of the expression where it was generated, by instantiating it at a security level higher than strictly necessary, or because it further restricts another annotation variable. Both situations are highly undesirable. Therefore only annotation variables that will not restrict the type that is being generalised will be instantiated, or annotation variables that occur free in the type environment (these represent types that will be generalised over at some other time). We call the annotation variables that occur free in the type τ that is being generalised over 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, not the complete set of variables that may not be instantiated. Other variables that do not occur in the type, but are the result of the inference algorithm can be restricted by active variables. In the constraint $\beta_1 \sqsubseteq \beta_2$, for example, 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 thus it is not clear what the lowest possible security level for β_2 is. The lowest possible security level will restrict the use of an expression the least, we therefore choose to delay instantiation of β_2 until β_1 is known. Variables such as β_2 are known as *pseudo active variables*. In Figure 6.14 we present a function that computes the set of all pseudo active variables, given the set of all active variables and a set of constraints.

All variables that are not (pseudo-) active can be instantiated, if there exists a non empty range of security levels for every variable. If, for any variable, it is not possible to choose a value such that all constraints are satisfied the program contains an error. In that case the compiler will explain where the problem arises and what caused it. Error explanation is discussed in Chapter 7. The range of allowed values for each variable is computed by a work list algorithm. We present the algorithm in Figure 6.15. The algorithm is based on the work list algorithm presented in Chapter 3 of [Nielson et al., 1999]. The algorithm first initialises the work list *worklist*, the result of the algorithm *analysis*, and two edge arrays *ub* and *lb*. We use two instead of one edge arrays as we compute both a lower bound and an upper bound for each variable. The array *ub* is used to propagate new values for the upper bound downward. The set $ub\ [\beta]$ contains all constraints where 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 update 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: $\lfloor analysis\ [\beta] \rfloor$ denotes the current lower bound of the range associated with β, and $\lceil analysis\ [\beta] \rceil$ 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 set of mappings from an annotation variable to a range of security levels. The function satisfiable (presented in Figure 6.16) processes the results of the work list algorithm. If all variables have a non empty range it will result in a substitution that replaces all non (pseudo-) active variable by the lower bound of the range. If any variable has an empty range an error has been found and the simplification and inference process stop.

Partitioning the constraint set is the final step of the simplification process. The partition of the constraint set consists of two sets. The first set contains constraints that contain at least one variable that is active in Γ. This set of constraints is propagated further by the inference algorithm. It contains information about annotation variables that are later generalised over. The other set contains all other constraints. These constraints will become predicates in the type that is generalised over. We present the partition algorithm in Figure 6.17.

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

$$
\begin{array}{lll}
\mathcal{W}_{sec}\ (\Gamma, n) & = \textbf{let} & \beta \textit{ be fresh} \\
& & \textbf{in } (\texttt{Int},\ \beta, id, \emptyset) \\
\mathcal{W}_{sec}\ (\Gamma, \mathit{True}) & = \textbf{let} & \beta \textit{ be fresh} \\
& & \textbf{in } (\texttt{Bool}, \beta, id, \emptyset) \\
\mathcal{W}_{sec}\ (\Gamma, \mathit{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 \circ \theta_2 \circ \theta_1 \\
& & \textbf{in } (\theta\ (\texttt{List}\ {\tau_1}^{\beta_1}), \beta, \theta, \theta\ (C_1 \\
& & \quad \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 \circ \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 \circ \theta_0)\ \beta_r) \\
& & \theta = \theta_2 \circ \theta_1 \circ \theta_0 \\
& & \textbf{in } (\theta\ ({\alpha_x}^{\beta_x} \to {\tau_0}^{\varphi_0}), (\theta_2 \circ \theta_1)\ \beta, \theta, C_0) \\
\mathcal{W}_{sec}\ (\Gamma, x) & = \textbf{let} & \beta_1 \textit{ be fresh} \\
& & (\sigma, \varphi) = \Gamma\ (x) \\
& & C' \Rightarrow \tau = \mathit{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 \circ \theta_2 \circ \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)
\end{array}
$$

Fig. 6.10: Type and Security Analysis Inference Algorithm

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

$$\begin{array}{lll}
\mathscr{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) = \mathscr{W}_{sec}\ (\Gamma, e_0) \\
& (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\theta_0\ \Gamma, e_1) \\
& (\tau_2, \varphi_2, \theta_2, C_2) = \mathscr{W}_{sec}\ ((\theta_1 \circ \theta_0)\ \Gamma, e_2) \\
& \theta_3 = \mathscr{U}\ ((\theta_2 \circ \theta_1)\ \tau_0, \texttt{Bool}) \\
& \theta_4 = \mathscr{U}\ (\theta_3\ \tau_2, (\theta_3 \circ \theta_2)\ \tau_1) \\
& \theta = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \circ \theta_0 \\
& C = C_0 \cup C_1 \cup C_2 \\
& \quad \cup \{ \varphi_0 \sqsubseteq \beta, \varphi_1 \sqsubseteq \beta, \varphi_2 \sqsubseteq \beta \} \\
& \textbf{in}\ (\theta\ \tau_2, \beta, \theta, \theta\ C) \\
\mathscr{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) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& (\sigma, C') = \mathit{gen}\ (\theta_1\ \Gamma, \varphi_1, \tau_1, C_1) \\
& (\tau_2, \varphi_2, \theta_2, C_2) = \mathscr{W}_{sec}\ (\theta_1\ \Gamma\ [x \mapsto (\sigma, \varphi_1)], e_2) \\
& \textbf{in}\ (\tau_2, \beta, \theta_2 \circ \theta_1, \theta_2\ (C' \cup C_2 \cup \{ \varphi_2 \sqsubseteq \beta \})) \\
\mathscr{W}_{sec}\ (\Gamma, e_1 \oplus e_2) = \textbf{let} & \beta\ \textit{be fresh} \\
& (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& (\tau_2, \varphi_2, \theta_2, C_2) = \mathscr{W}_{sec}\ (\theta_1\ \Gamma, e_2) \\
& \tau^1_{\oplus} \to \tau^2_{\oplus} \to \tau_{\oplus} = \Gamma_{\oplus}\ (\oplus) \\
& \theta_3 = \mathscr{U}\ (\theta_2\ \tau_1, \tau^1_{\oplus}) \\
& \theta_4 = \mathscr{U}\ (\theta_3\ \tau_2, \tau^2_{\oplus}) \\
& \theta = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \\
& \textbf{in}\ (\tau_{\oplus}, \beta, \theta, \theta\ (C_1 \cup C_2 \cup \{ \varphi_1 \sqsubseteq \beta, \varphi_2 \sqsubseteq \beta \})) \\
\mathscr{W}_{sec}\ (\Gamma, u\ e_1) = \textbf{let} & \beta\ \textit{be fresh} \\
& (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& \tau^1_u \to \tau_u = \Gamma_{\oplus}\ (u) \\
& \theta_2 = \mathscr{U}\ (\tau_1, \tau^1_u) \\
& \textbf{in}\ (\tau_u, \beta, \theta_2 \circ \theta_1, \theta_2\ (C_1 \cup \{ \varphi_1 \sqsubseteq \beta \})) \\
\mathscr{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) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& \theta_2 = \mathscr{U}\ (\tau_1, ({\alpha_1}^{\beta_1}, {\alpha_2}^{\beta_2})) \\
& \textbf{in}\ (\theta_2\ \alpha_1, \beta, \theta_2 \circ \theta_1, \theta_2\ (C_1 \cup \{ \varphi_1 \sqsubseteq \beta, \beta_1 \sqsubseteq \beta \})) \\
\mathscr{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) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& \theta_2 = \mathscr{U}\ (\tau_1, ({\alpha_1}^{\beta_1}, {\alpha_2}^{\beta_2})) \\
& \textbf{in}\ (\theta_2\ \alpha_2, \beta, \theta_2 \circ \theta_1, \theta_2\ (C_1 \cup \{ \varphi_1 \sqsubseteq \beta, \beta_2 \sqsubseteq \beta \}))
\end{array}$$

Fig. 6.11: Type and Security Analysis Inference Algorithm (Continued)

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

$$\begin{array}{lll}
\mathscr{W}_{sec}\ (\Gamma, \textbf{null}\ e_1) & = \textbf{let} & \alpha_1, \beta_1, \beta\ \textit{be fresh} \\
& & (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& & \theta_2 = \mathscr{U}\ (\tau_1, \texttt{List}\ \alpha_1{}^{\beta_1}) \\
& & \textbf{in}\ (\texttt{Bool}, \beta, \theta_2 \circ \theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\})) \\
\mathscr{W}_{sec}\ (\Gamma, \textbf{hd}\ e_1) & = \textbf{let} & \alpha_1, \beta_1, \beta\ \textit{be fresh} \\
& & (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& & \theta_2 = \mathscr{U}\ (\tau_1, \texttt{List}\ \alpha_1{}^{\beta_1}) \\
& & \textbf{in}\ (\theta_2\ \alpha_1, \beta, \theta_2 \circ \theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta, \beta_1 \sqsubseteq \beta\})) \\
\mathscr{W}_{sec}\ (\Gamma, \textbf{tl}\ e_1) & = \textbf{let} & \alpha_1, \beta_1, \beta\ \textit{be fresh} \\
& & (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\Gamma, e_1) \\
& & \theta_2 = \mathscr{U}\ (\tau_1, \texttt{List}\ \alpha_1{}^{\beta_1}) \\
& & \textbf{in}\ (\theta_2\ \tau_1, \beta, \theta_2 \circ \theta_1, \theta_2\ (C_1 \cup \{\varphi_1 \sqsubseteq \beta\})) \\
\mathscr{W}_{sec}\ (\Gamma, \textbf{declassify}\ e\ \varphi) & = \textbf{let} & \beta\ \textit{be fresh} \\
& & (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\Gamma, e) \\
& & \textbf{in}\ (\tau_1, \beta, \theta_1, C_1 \cup \{\varphi \sqsubseteq \varphi_1, \varphi \sqsubseteq \beta\}) \\
\mathscr{W}_{sec}\ (\Gamma, \textbf{protect}\ e\ \varphi) & = \textbf{let} & \beta\ \textit{be fresh} \\
& & (\tau_1, \varphi_1, \theta_1, C_1) = \mathscr{W}_{sec}\ (\Gamma, e) \\
& & \textbf{in}\ (\tau_1, \beta, \theta_1, C_1 \cup \{\varphi_1 \sqsubseteq \varphi, \varphi \sqsubseteq \beta\})
\end{array}$$

Fig. 6.12: Type and Security Analysis Inference Algorithm (Continued)

$$\begin{array}{l}
\mathit{simplify}(\Gamma, \varphi, \tau, C) = \textbf{do} \\
\quad \mathit{active}\ = \mathit{pseudoActive}\ (\mathit{fav}\ (\Gamma) \cup \mathit{fav}\ (\textbf{Ty}), C) \\
\quad \mathit{ranges} = \textbf{range}\ C \\
\quad \theta\ \ = \textbf{satisfiable}\ (\mathit{active}, \mathit{ranges}) \\
\quad c'\ \ = \theta\ C \\
\quad \textbf{return}\ \mathit{partition}(\Gamma, c')
\end{array}$$

Fig. 6.13: Simplification algorithm

$$\begin{array}{l}
\mathit{pseudoActive}\ (\mathit{active}, C) = \textbf{do} \\
\quad \mathit{vars} = \{u \mid (l \sqsubseteq u)\ \in\ C\ \wedge\ l\ \in\ \mathit{active}\} \cup \mathit{active} \\
\quad \textbf{if}\ ((\mathit{vars} \cap \mathit{active}) \neq \emptyset) \\
\quad\quad \textbf{then do} \\
\quad\quad\quad \mathit{pseudoActive}\ (\mathit{vars}, C) \\
\quad\quad \textbf{else do} \\
\quad\quad\quad \textbf{return}\ \mathit{vars}
\end{array}$$

Fig. 6.14: Determine set of pseudo active variables

```
range(C) = do
  worklist = { }
  analysis = [ ]
  ub       = [ ]
  lb       = [ ]
  for all π in C do worklist := worklist ∪ { π }
  for all β in fav(C) do
    analysis = analysis[β ↦ (⊥, ⊤)]
    ub       = ub[β ↦ { }]
    lb       = lb[β ↦ { }]
  for all π@(β1, β2) in C do
    ub[β2]   = ub[β2] ∪ { π }
    lb[β1]   = lb[β1] ∪ { π }
  while worklist ≠ { } do
    let   C1 ∪ { π } = worklist
      in do worklist := C1
            case C of
              (β1 ⊑ β2) ⇒ if (analysis[β1] ≠ analysis[β1] ⊓ ⌈analysis [β2]⌉)
                           then do
                             worklist = worklist ∪ ub[β1]
                             analysis[β1] = analysis[β1] ⊓ ⌈analysis [β2]⌉
                         if (analysis[β2] ≠ analysis[β2] ⊔ ⌊analysis [β1]⌋)
                           then do
                             worklist = worklist ∪ lb[β2]
                             analysis[β2] = analysis[β2] ⊔ ⌊analysis [β1]⌋
              (φ ⊑ β)   ⇒ if (analysis[β] ≠ analysis[β] ⊔ φ)
                           then do
                             worklist = worklist ∪ lb[β]
                             analysis[β] = analysis[β] ⊔ φ
              (β ⊑ φ)   ⇒ if (analysis[β] ≠ analysis[β] ⊓ φ)
                           then do
                             worklist = worklist ∪ ub[β]
                             analysis[β] = analysis[β] ⊓ φ
  return analysis
```

Fig. 6.15: Worklist algorithm for range computation

```
satisfiable(active, analysis) = do
  substitution = Id
  for all [β ↦ (l, u)] in analysis do
    if (l ⊑ u)
      then do
        if (β ∉ active ∨ l ≡ u)
          then do
            substitution = [β ↦ l] ∘ substitution
      else do
        error
  return substitution
```

Fig. 6.16: Checking satisfiability

```
partition (Γ, C) = do
  active = fav(Γ)
  prop   = {(l ⊑ u) ∈ C | (l ∈ active ∨ u ∈ active)}
  qual   = {π ∈ C | π ∉ prop}
  return (prop, qual)
```

Fig. 6.17: Partitioning of constraints

7. ERROR MESSAGES

In this chapter we discuss how errors are generated when an inconsistency is found in the constraint set during simplification. We combine the approaches of Heeren [Heeren, 2005] and, Haack and Wells [Haack and Wells, 2004]. Haack and Wells generate type error slices when a type error is found in a program. A 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 type error slice together form a minimal unsatisfiable set of constraints. The very same approach applies to security type errors.

In Section 7.1 we discuss how a minimal unsatisfiable set of constraints is extracted from the complete set of constraints that were found to be inconsistent in the simplification phase. Instead of presenting the program slice that follows from this minimal set we use heuristics to find the constraint that caused the inconsistency and explain why the program is rejected; This is similar to Heerens approach to providing type error messages. In Section 7.3 we revisit the login program introduced in Chapter 2 to show the results of our approach.

7.1 Minimal unsatisfiable constraint set

When a constraint set is unsatisfiable, an error has to be reported to the programmer. Many of the constraints that were generated do not contribute to the error. It would be a waste to process all of these constraints to find one erroneous constraint, therefore we compute a minimal set of unsatisfiable constraints, as this set will also contain a constraint that causes the inconsistency. This set is likely to be much smaller than the whole set of constraints. Haack and Wells, and Stuckey, Sulzmann and Wazny both present an algorithm for computing such as set [Haack and Wells, 2004], [Stuckey et al., 2003]. Both use this minimal set of unsatisfiable constraints to produce a so called type error slice. A type error slice is a fragment of the program that contribute to the error. We use the algorithm presented by Stuckey, Sulzmann and Wazny in Section 7 of [Stuckey et al., 2003], we present the algorithm in Figure 7.1. The algorithm takes an unsatisfiable set of constraints D and builds another set of constraints M that in the end contains a minimal unsatisfiable set of constraints. The algorithm is built from two nested while loops: the inner loop adds a constraint from the original set D minus C, to a copy of the unsatisfiable set thus far C. When the set C becomes unsatisfiable 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 satisfiable.

From the minimal unsatisfiable set of constraints we can build a program slice. 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 [Haack and Wells, 2004].

```
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. 7.1: Computing a minimal unsatisfiable set of constraints

7.2 Heuristics

Given the minimal set of errors we could present the program slice resulting from that set of constraints as the error message. But in practice these slices do not actually explain what the problem is, they only show the parts of the program that contribute to the inconsistency. Using a heuristic we can extract the actual cause of the inconsistency from the minimal unsatisfiable set of constraints. Hage and Heeren implemented several heuristics for explaining type errors in the TOP framework [Hage and Heeren, 2006]. In TOP, heuristics are used to find the best candidate to put the blame on, and if possible suggest a fix. We implemented a similar system to find the most probable cause and suggest a fix. Without going into the details we like to point out that our framework of heuristics is very similar to the heuristics framework in TOP. In the remainder of this section we discuss some of the heuristics that we implemented and provide some examples of their uses. We group the heuristics by their type; we distinguish between two sorts of heuristics namely filter heuristics and selector heuristics.

A filter heuristic removes some constraints from the set of constraints. Removing constraints from the unsatisfiable set is done because a lot of constraints in this set are not likely to be the cause of the inconsistency and can easily be identified. In the case that a heuristic would remove all constraints, it is not applied. A filter heuristic can remove a constraint from the minimal unsatisfiable set when it is not likely that the constraint actually caused the inconsistency. A filter heuristic will never remove all constraints from the set, as that would not leave any constraint left to blame. When a heuristic leaves only one constraint it behaves like a selector heuristic (discussed in the next section) and generates an error message blaming that particular constraint.

Selector heuristics select one particular constraint that caused the inconsistency for a reason specific to the heuristic. When a selector heuristic can select a constraint no further heuristics are applied, it directly displays the error message. Multiple selector heuristics can be combined in a voting mechanism. All heuristics are then applied to the same set of constraints. If multiple heuristics select a constraint the heuristic with the highest rating will generate the error message. A selector heuristic that can not select a constraint has no effect. We now consider a number of heuristics in detail.

7.2.1 Filter irrefutable constraints

All constraints that we generate are tagged with an amount of trust. The amount of trust in a constraint depends on the node where the constraint is generated. At some nodes we generate a constraint just for uniformity with the rest of the type system, knowing that this constraint can never be wrong. Such constraints obviously should never be blamed. The irrefutable constraints filter heuristic removes these constraints from the set of constraints. 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 for obvious reasons is always true, and we therefore never want to blame it. This filter heuristic was introduced by Heeren in Section 8.3 of **??**.

7.2.2 Filter propagation constraints

Most of the constraint that are generated only propagate security levels. Consider for example the program in Figure 7.2. The value *secureVal* is passed to the identity function once and twice to the increment function 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. All applications in this chain generate constraints that together form a chain. The endpoints are formed by a constraint stating that H is less or equally secure as some variable β and a constraint that restricts that a variable β_1 is equally or less secure than L. The other constraints form a chain of variables between β and β_1. They are generated by the applications of *incr* and *id*. These constraint are called propagation constraints. When the function associated with a propagation constraint would receive the blame it is not possible to tell that the expected argument doesn't match the given one, it did expect a value protected at any level (otherwise it would have been an endpoint of the propagation chain)! What is actually wrong is what is done with the result of the application in the end, but for that this application can not be blamed. It would be possible to suggest that a declassification at this position would solve the problem. Or a function that accepts the secure value and returns a lesser protected value of the same type would also be correct. However it is not easy for the compiler to judge if a value can be declassified, without revealing too much information. And it is also not possible for the compiler to give a meaningful suggestion for an alternative function as it cannot determine what the purpose of the program is. The compiler will thus have to accept that the path in between the end points is correct and does what the programmer intended (it was checked to be type correct before the security constraints were checked). The propagation constraints can thus never be blamed, and the filter propagation heuristic removes all propagation constraints from the minimal error set.

$$
\begin{aligned}
&\mathit{secureVal} :: \mathtt{Int}^{\mathsf{H}} \\
&\mathit{incr} :: \forall \beta_1 . \forall \beta_2 . \beta_1 \sqsubseteq \beta_2 \Rightarrow \mathtt{Int}^{\beta_1} \rightarrow \mathtt{Int}^{\beta_2} \\
&\mathit{id} :: \forall \alpha . \forall \beta_1 . \forall \beta_2 . \beta_1 \sqsubseteq \beta_2 \Rightarrow \alpha^{\beta_1} \rightarrow \alpha^{\beta_2} \\
&\mathit{print} :: \forall \alpha . \alpha^{\mathsf{L}} \rightarrow \alpha^{\mathsf{L}} \\
&\mathit{print}\ (\mathit{incr}\ (\mathit{incr}\ (\mathit{id}\ \mathit{secureVal})))
\end{aligned}
$$

Fig. 7.2: Propagating constraints

7.2.3 Sibling heuristic

The sFun++ language has two constructs to change the security level namely **declassify** and **protect**. These constructs are very similar and can be mixed up easily. It is possible that a programmer, by mistake, tries to declassify an expression by using the **protect** statement and a lower security level, or vice versa. In Figure 7.3 we present an expression that contains such a mistake. The value secureValue has type $\texttt{Bool}^{H}$. The function *printValue* prints a value that is protect at security level Low. The programmer of the program tried to declassify the secureValue by protecting it to the level *Low*, the protect statement can, however, not be used to declassify information. The sibling heuristic will generate the error message presented in Figure 7.4. The error suggests to replace the protect statement with the declassify statement, so that this will indeed result in a correct program. The sibling heuristic requires that there are only two constraints in the minimal error set, both containing one non-variable annotation. It will look at the nodes where the constraints where generated if these happen to be a **declassify** or **protect** node the the heuristic applies and blames the constraint generated by the **declassify** or **protect** node, that restricts the security level of the expression it declassifies or protects.

$$\begin{aligned} &\mathit{secureValue} = \textbf{protect}\ \mathit{True}\ \mathit{High} \\ &\mathit{printSecure} = \mathit{printValue}\ (\textbf{protect}\ \mathit{secureValue}\ \mathit{Low}) \end{aligned}$$

Fig. 7.3: Example program: misuse of protect statement

```
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
```

Fig. 7.4: Error generated by sibling heuristic for the program in Figure 7.3

7.2.4 Majority heuristic

The meaning of a security error is that some protected value is leaked to a less trusted place. Johnson and Walz introduced the idea to look at the amount of evidence for one constraint to be an error in [**?**]. 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. From this set of constraints it counts for each security level the constraints that state that the expression should at least have that security level. If the amount of constraints that testify that the expression should have that lesser security level is say 4 times larger than the amount of constraints demanding a higher security level the subexpressions where these constraints that demand a higher security level where generated might be the actual cause of the problem. If there exists an overwhelming majority testifying that the expression should have a lower security level, the subexpressions that force a higher level are added to the error message stating that they caused the security level to be too high. It is of course not possible to determine whether these subexpressions are wrong or that the context of the whole expression is wrong. It is therefore only mentioned that these subexpressions cause the whole expression to be protected at a level that is too high.

In Figure 7.5 an erroneous program is listed. The first five lines declare some values, the first four are protected at level *Low*, 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 function print expects value that is protected at at most level *Low*. 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*... this is indeed an incorrect application. As there is very little evidence stating that the sum should be protected at level *High* the heuristic also tells what caused the expression to be protected at this level. The programmer can now decide whether the use of *five* in this place was incorrect or the use of *print* that expected an argument protect at level *Low*.

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

```
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)
```

Fig. 7.6: Error generated by the majority heuristic in Figure 7.5

7.2.5 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 good a 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 receive a higher trust value than constraints that were generated locally. 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 with this construct it is most easy to introduce an error with function application. In next section we present an example of a program where the least trusted constraint heuristic accurately blames the correct application in Figure 7.12. In Figure 7.13 the error that is generated by the least trusted constraint is listed.

7.2.6 When all heuristics fail

In the occasion that none of the aforementioned heuristics can determine the cause of the error it is still possible to present a program slice to the user as is done by Haack and Wells [Haack and Wells, 2004]. The AST nodes that correspond to each constraint in the minimal unsatisfiable set of constraints exactly match all program points where a change could be made to solve the problem. The program presented in Figure 7.5 would, by the method of Haack and Wells, result in the slice presented in Figure 7.7. This program slice 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 sort 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 still provides a good error message. We have not implemented a way of presenting program slices due to time constraints. We do however print a list of all program points that contribute to the inconsistency, and we believe that it is possible to construct a program slice from this information.

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

Fig. 7.7: Program slice error from the program in Figure 7.5

7.3 Discussion

In this section we discuss the result of our error reporting system. In Figure 7.8 we present the correct version of a login program similar to the programs presented in Chapter 2. We shortened the usernames and password in our program both to just a single integer, as our language does not have strings.

The type of the password file is given in Figure 7.9. The password file is a list of which the structure and elements are not protected, and that contains pairs consisting of an unprotected username and a protected password. The function *findUser* retrieves the username 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 username is not known. The function *login* receives a username 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*. The result of the comparison is therefore first declassified before it is printed.

In Figure 7.10 we present a variation on our login program where declassification is forgotten. This means that the boolean value that was protected *High* is passed to the print function. The error message our compiler gives is listed in Figure 7.11. The application is blamed by the least trusted constraint heuristic, and correctly blames the application where a secure value is printed through an insecure print function.

A mistake that is easily made is that the programmer tries to declassify information through the **protect** statement. In Figure 7.12 we present another variation on our login program that uses protect to declassify

```
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)
```

Fig. 7.8: sFun++ Login program

$$passwordFile :: \forall\beta_1.\forall\beta_2.(\texttt{List}\ (\texttt{Int}^{\beta_1}, \texttt{Int}^{High})^{\beta_2})^{Low}$$

Fig. 7.9: Type of password file

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

Fig. 7.10: sFun++ Login program without declassification

```
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 given argument: "let userRecord = ((findUser u) passwordFile) in if null
    userRecord then False else (snd head userRecord == p)"
Is protected at a higher level.
```

Fig. 7.11: sFun++ error given for program in Figure 7.10

information. The error the sFun++ compiler gives is listed in Figure 7.13. The sibling heuristic through the constraints detects faulty usage of the protect statement. It generates an error message that suggests to replace the **protect** statement by a declassify *statement*.

The results we presented in this and the previous section are a good indication that we indeed have more

$$
\begin{array}{l}
\mathit{login} = \textbf{fn}\ u \Rightarrow \textbf{fn}\ p \Rightarrow \mathit{print}\ (\textbf{protect}\ (\textbf{let}\ \mathit{userRecord} = (\mathit{findUser}\ u\ \mathit{passwordFile}) \\
\quad \textbf{in}\ (\textbf{if}\ (\textbf{null}\ \mathit{userRecord}) \\
\quad\quad \textbf{then}\ \mathit{False} \\
\quad\quad \textbf{else}\ ((\textbf{snd}\ (\textbf{hd}\ \mathit{userRecord})) \equiv p)))\ \mathit{Low})
\end{array}
$$

Fig. 7.12: sFun++ Login program without declassification

```
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
```

Fig. 7.13: sFun++ error given for program in Figure 7.12

control over the content of our error messages by using constraints. By separating type and security error messages we can provide less obfuscated error messages than the Haskell library implementation provides (Figure 2.10). By using heuristics we have more control over the contents of the error message. With this control over the contents of the message we can explain the cause of the inconsistency instead of only reporting an illegal information flow (Figure 2.6). The heuristics accurately blame the correct cause and suggest, if possible, a useful fix for the found inconsistency. We only implemented one program construct specific heuristic to adapt the error message to commonly made mistakes, we point out that for other programming constructs such heuristics can be implemented as well, making the error messages even more precise.

8. CONCLUSIONS

In this last chapter we summarise our contributions we presented in this thesis in Section 8.1. In Section 8.2 we conclude by discussing some future work.

8.1 Conclusion

Security analysis is a validating analysis, meaning that it can approve or reject a program. In Chapter 2 we looked at two existing solutions for security analysis: FlowCaml which is an OCaml dialect, and a Haskell library. Both systems rejected programs with an error that contained little or obfuscated information about the reason why the program was rejected. In this thesis we presented our sFun++ language in Chapter 4. In the chapters 5 and 6 we defined and implemented a security analysis for this language. Even though the constraints for our security analysis are generated during type inferencing we do not combine type and security error messages. The separation of type errors and security errors results in error messages that do not contain information about the other. In our opinion this results in more readable security error messages because they contain no information that is not relevant to the error (type information).

As a default error message we use program slices as presented by Haack and Wells [Haack and Wells, 2004]. From an inconsistent constraint set we extract a minimal unsatisfiable set of constraints. All constraints are associated with a node in the source program, the constraints in the minimal unsatisfiable constraint set together form a program slice. This program slice cab be used as a default error message. The program slice contains all points in the source program that can be changed to fix the inconsistency and explains what security levels were expected and provided.

The default error is given if a more specialised error message can not be given. Heuristics can generate more accurate error messages that explain what caused the problem and if possible suggest how to fix the problem. In Section 7.2 we presented heuristics that generate specialised error messages. A filter heuristic can remove constraints from the minimal unsatisfiable set that are not likely to have caused the problem. We presented two filter heuristics: the propagation constraints heuristic and the irrefutable constraints heuristic. Other heuristics, selector heuristics, blame a constraint for the inconsistency and generate an explanation why it caused an inconsistency. The selector heuristics we presented are: the sibling heuristic, the majority heuristic and the least trusted constraint heuristic. We believe that the error messages given by our system, are an improvement over the errors given by existing systems.

8.2 Future work

In this section we conclude with a list of possible future work.

- *Specialised heuristics for program constructs.* In this thesis we presented a system that reconstructs security types that gives us more control over the explanation in the error message. The heuristics that we presented use this extra control over the error messages. We presented a heuristic that is specialised in common mistakes made with **protect** and **declassify** statements. For mistakes made with other constructs it is also possible to generate specialised messages. For example the **if** e_0 **then** e_1 **else** e_2 construct can also easily lead to unexpected security errors if the programmer doesn't fully understand the construct. The programmer could for instance assume that only the security levels of e_1 and e_2 contribute to the security level of the result, because the value of these two constructs will be the value of the whole expression. In the case that the programmer uses a highly secure conditional and two low protected expression for the branches, the result of the whole expression will be highly secure. In the case that this leads to a security error a specialised heuristic could explain that the whole expression was highly secure as the outcome of the expression reveals the value of the conditional.

- *Explicit type signatures.* Adding explicit type signatures to the language adds new sorts of inconsistencies that can be found by the security analysis. New sorts of heuristics have to be introduced to locate the cause of the mismatch between the inferred security type and the provided type.

- *Constraint solving over a larger scope.* The system presented in this thesis solves constraint sets, and thus detects inconsistencies at the level of a binding group. When the scope would be moved a higher level (module level for example), different sorts of diagnoses and error messages could be given. In this setting it would be possible to suggest that a definition must be changed if it is used wrongly consistently through out the module.

- *Hints for optimising analysis in the source program.* Regular optimising analyses can never go wrong. Therefore no error messages have to be reported for these analyses. Currently some programmers change their programs so that the compiler can perform better optimisation on their programs. As an alternative it is possible to give the programmer the possibility to provide hints to the compiler to perform better optimisations. Such hints can be wrong, and may lead to inconsistencies. Such inconsistencies then have to be reported to the user. We believe that is is worth investigating the possibility to give the programmer the opportunity to provide hints for an optimising analyses and providing error messages for inconsistencies found in a similar manner as presented in this thesis.

APPENDIX

A. BASIC INFERENCE ALGORITHM

Below we present the underlying type inference system. The formal specification is presented in Section 4.3. The algorithm is based on algorithm $\mathscr{W}$ [Damas and Milner, 1982].

$$
\begin{array}{lll}
\mathscr{W}_{ul}\ (\Gamma, n) & = & (\texttt{Int}, id) \\
\mathscr{W}_{ul}\ (\Gamma, True) & = & (\texttt{Bool}, id) \\
\mathscr{W}_{ul}\ (\Gamma, False) & = & (\texttt{Bool}, id) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{Nil}) & = \textbf{let} & \alpha \ be\ fresh \\
 & \quad \textbf{in} & (\texttt{List}\ \alpha, id) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{Cons}\ e_1\ e_2) & = \textbf{let} & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
 & & (\tau_2, \theta_2) = \mathscr{W}_{ul}\ (\theta_1\ \Gamma, e_2) \\
 & & \theta_3 \quad = \mathscr{U}\ (\tau_2, \texttt{List}\ (\theta_2\ \tau_1)) \\
 & \quad \textbf{in} & (\theta_3\ \tau_2, \theta_3 \circ \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, (e_1, e_2)) & = \textbf{let} & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
 & & (\tau_2, \theta_2) = \mathscr{W}_{ul}\ (\theta_1\ \Gamma, e_2) \\
 & \quad \textbf{in} & ((\theta_2\ \tau_1, \tau_2), \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, x) & = & (inst\ (\Gamma\ (x)), id) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{fn}\ x \Rightarrow e_0) & = \textbf{let} & \alpha \ be\ fresh \\
 & & (\tau_0, \theta_0) = \mathscr{W}_{ul}\ (\Gamma\ [x \mapsto \alpha], e_0) \\
 & \quad \textbf{in} & ((\theta_0\ \alpha) \rightarrow \tau_0, \theta_0) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{fun}\ f\ x \Rightarrow e_0) & = \textbf{let} & \alpha_x, \alpha_r \ be\ fresh \\
 & & (\tau_0, \theta_0) = \mathscr{W}_{ul}\ (\Gamma\ [f \mapsto \alpha_x \rightarrow \alpha_r]\ [x \mapsto \alpha_x], e_0) \\
 & & \theta_1 \quad = \mathscr{U}\ (\tau_0, \theta_0\ \alpha_r) \\
 & & \theta \quad = \theta_1 \circ \theta_0 \\
 & \quad \textbf{in} & ((\theta\ \alpha_x) \rightarrow (\theta_1\ \tau_0), \theta) \\
\mathscr{W}_{ul}\ (\Gamma, e_1\ e_2) & = \textbf{let} & \alpha_r \ be\ fresh \\
 & & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
 & & (\tau_2, \theta_2) = \mathscr{W}_{ul}\ (\theta_1\ \Gamma, e_2) \\
 & & \theta_3 \quad = \mathscr{U}\ (\theta_2\ \tau_1, \tau_2 \rightarrow \alpha_r) \\
 & & \theta \quad = \theta_3 \circ \theta_2 \circ \theta_1 \\
 & \quad \textbf{in} & (\theta_3\ \alpha_r, \theta) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & = \textbf{let} & (\tau_0, \theta_0) = \mathscr{W}_{ul}\ (\Gamma, e_0) \\
 & & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\theta_0\ \Gamma, e_1) \\
 & & (\tau_2, \theta_2) = \mathscr{W}_{ul}\ (\theta_1 \circ \theta_0\ \Gamma, e_2) \\
 & & \theta_3 \quad = \mathscr{U}\ (\theta_2 \circ \theta_1\ \tau_0, \texttt{Bool}) \\
 & & \theta_4 \quad = \mathscr{U}\ (\theta_3\ \tau_2, \theta_3 \circ \theta_2\ \tau_1) \\
 & & \theta \quad = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \circ \theta_0 \\
 & \quad \textbf{in} & (\theta_4 \circ \theta_3\ \tau_2, \theta)
\end{array}
$$

$$
\begin{array}{lll}
\mathscr{W}_{ul}\ (\Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2) & = \textbf{let} & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & (\tau_2, \theta_2) = \mathscr{W}_{ul}\ (\theta_1\ \Gamma\ [x \mapsto gen\ (\tau_1)], e_2) \\
& \textbf{in} & (\tau_2, \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, e_1 \oplus e_2) & = \textbf{let} & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & (\tau_2, \theta_2) = \mathscr{W}_{ul}\ (\theta_1\ \Gamma, e_2) \\
& & \tau^1_{\oplus} \to \tau^2_{\oplus} \to \tau_{\oplus} = \Gamma_{\oplus}\ (\oplus) \\
& & \theta_3 = \mathscr{U}\ (\theta_2\ \tau_1, \tau^1_{\oplus}) \\
& & \theta_4 = \mathscr{U}\ (\theta_3\ \tau_2, \tau^2_{\oplus}) \\
& & \theta\ = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \\
& \textbf{in} & (\tau_{\oplus}, \theta) \\
\mathscr{W}_{ul}\ (\Gamma, u\ e_1) & = \textbf{let} & (\tau_1, \theta_1)\ = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & \tau^1_u \to \tau_u = \Gamma_{\oplus}\ (u) \\
& & \theta_2 \qquad = \mathscr{U}\ (\tau_1, \tau^1_u) \\
& \textbf{in} & (\tau_u, \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{fst}\ e_1) & = \textbf{let} & \alpha_1, \alpha_2\ \textit{be fresh} \\
& & (\tau_1, \theta_1)\ = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & \theta_2 \qquad = \mathscr{U}\ (\tau_1, (\alpha_1, \alpha_2)) \\
& \textbf{in} & (\theta_2\ \alpha_1, \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{snd}\ e_1) & = \textbf{let} & \alpha_1, \alpha_2\ \textit{be fresh} \\
& & (\tau_1, \theta_1)\ = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & \theta_2 \qquad = \mathscr{U}\ (\tau_1, (\alpha_1, \alpha_2)) \\
& \textbf{in} & (\theta_2\ \alpha_2, \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{null}\ e_1) & = \textbf{let} & \alpha_1\ \textit{be fresh} \\
& & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & \theta_2 = \mathscr{U}\ (\tau_1, \texttt{List}\ \alpha_1) \\
& \textbf{in} & (\texttt{Bool}, \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{hd}\ e_1) & = \textbf{let} & \alpha_1\ \textit{be fresh} \\
& & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & \theta_2 = \mathscr{U}\ (\tau_1, \texttt{List}\ \alpha_1) \\
& \textbf{in} & (\theta_2\ \alpha_1, \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{tl}\ e_1) & = \textbf{let} & \alpha_1\ \textit{be fresh} \\
& & (\tau_1, \theta_1) = \mathscr{W}_{ul}\ (\Gamma, e_1) \\
& & \theta_2 = \mathscr{U}\ (\tau_1, \texttt{List}\ \alpha_1) \\
& \textbf{in} & (\theta_2\ \tau_1, \theta_2 \circ \theta_1) \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{declassify}\ e\ s) & = \mathscr{W}_{ul}\ (\Gamma, e) & \\
\mathscr{W}_{ul}\ (\Gamma, \textbf{protect}\ e\ s) & = \mathscr{W}_{ul}\ (\Gamma, e) &
\end{array}
$$

BIBLIOGRAPHY

[Abadi, 2007] Abadi, M. (2007). Access control in a core calculus of dependency. *Electron. Notes Theor. Comput. Sci.*, 172:5–31.

[Abadi et al., 1999] Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. G. (1999). 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. ACM.

[Damas and Milner, 1982] Damas, L. and Milner, R. (1982). 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. ACM.

[Davey and Priestley, 1990] Davey, B. A. and Priestley, H. A. (1990). *Introduction to lattices and order / B.A. Davey, H.A. Priestley*. Cambridge University Press, Cambridge [England] ; New York :.

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

[Hage and Heeren, 2006] Hage, J. and Heeren, B. (2006). Heuristics for type error discovery and recovery (revised revised). Technical Report UU-CS-2006-054, Department of Information and Computing Sciences, Utrecht University.

[Hage and Heeren, 2009] Hage, J. and Heeren, B. (2009). Strategies for solving constraints in type and effect systems. *Electron. Notes Theor. Comput. Sci.*, 236:163–183.

[Hage et al., 2007] Hage, J., Holdermans, S., and Middelkoop, A. (2007). A generic usage analysis with subeffect qualifiers. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 235–246, New York, NY, USA. ACM.

[Haskell, 2009] Haskell (2009). `http://www.haskell.org`.

[Heeren, 2005] Heeren, B. J. (2005). *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands.

[Heintze and Riecke, 1998] Heintze, N. and Riecke, J. G. (1998). 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. ACM.

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

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

[Pottier and Simonet, 2002] Pottier, F. and Simonet, V. (2002). Information flow inference for ml. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, New York, NY, USA. ACM.

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

[Russo et al., 2008] Russo, A., Claessen, K., and Hughes, J. (2008). 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. ACM.

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

[Stuckey et al., 2003] Stuckey, P. J., Sulzmann, M., and Wazny, J. (2003). Interactive type debugging in haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83, New York, NY, USA. ACM.

[Stuckey et al., 2004] Stuckey, P. J., Sulzmann, M., and Wazny, J. (2004). Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 80–91, New York, NY, USA. ACM.

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

[Volpano and Smith, 2000] Volpano, D. and Smith, G. (2000). Verifying secrets and relative secrecy. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–276, New York, NY, USA. ACM.

[Volpano and Smith, 1997] Volpano, D. M. and Smith, G. (1997). A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK. Springer-Verlag.