THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY • AUSTRALIA

School of Computer Science and Engineering
Faculty of Engineering

FLOGENT

AN INFOMATION FLOW SECURITY FEATURE FOR COGENT

A Thesis (Part C) Report By
Vivian Ye-Ting Dang

Supervised By
Christine Rizkallah

June 27, 2020

Contents

List of Figures

Acknowledgements

I would like to thank Christine Rizkallah, my supervisor, and Amos Robinson for their support and guidance while I am undertaking this honours thesis, I deeply appreciate the mentorship and advice from both of them.

I am extremely grateful to Liam O'Connor, for being an exceptional lecturer, tutor and friend, without whom I would not have been inspired to undertake my thesis in this field, nor would I have been able to get through all the hurdles that I have come across during my years in university.

I would like to also thank Kai Engelhardt for being a great educator who opened the door of Formal Methods and the more theoretical side of computer science to me.

Of course, I want to also thank my mum Monica and my brother Justin for their unlimited support throughout the years. Thank you to all my friends, who have been there for me at different times. I would not have been able to get through my degree without them: Alexander Rowell, Anna Azzam, Annie Liu, David Lei, Edward Thomson, Emmet Murray, Hussein Debel, Kristian Mansfield, Mahika Suri, Nethan Tran, Subramanya Vajiraya, Victor Phan and many others to whom I am very much in-debted.

Chapter 1

Introduction

In a world where the amount of sensitive information being put online is increasing, the risk of this information falling into the hands of malicious people also increases; this calls for more emphasis on information security within systems.

This thesis introduces FLOGENT, an information flow control system for the programming language COGENT (O'Connor et al., 2016). COGENT is a higher-order, purely functional language with *uniqueness types*. The semantics of COGENT are designed to be easy to reason about, in accordance with its aim of reducing the cost of *verification*.

Software verification aims to formally establish by proof that a program fully satisfies its expected behaviour and outcomes (i.e. its specification). Proving program correctness often involves an interactive theorem prover like *Isabelle/HOL* (Nipkow, Paulson, and Wenzel, 2002).

Given a COGENT program, the compiler generates a high-level *shallow embedding* of the program's semantics in Isabelle/HOL, a C program, and a proof that the generated C program is a refinement of the generated Isabelle/HOL embedding. This means that any functional correctness property proved about the shallow embedding also holds for the generated C program. As a result, the need to verify manually written C programs is significantly reduced, which in turn brings down the cost of verification, as C is significantly harder to reason about than COGENT.

COGENT is designed for the implementation of operating systems components such as file-systems. Vulnerabilities in operating systems can compromise the confidentiality and integrity of the entire system. As such, ensuring COGENT software is free of vulnerabilites is particularly important.

A recent presentation by Miller (2019) indicates that approximately 70% of *Com-*

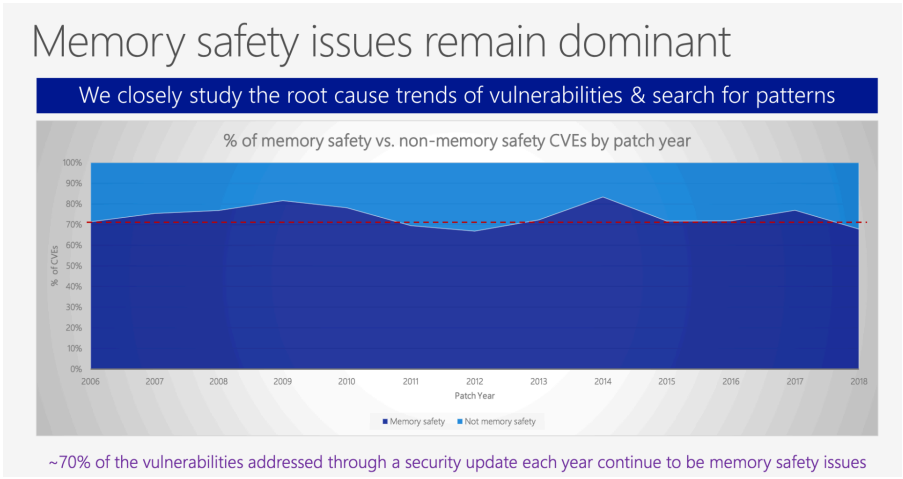


Figure 1.1: Identifying memory-safety-related CVEs (from Miller (2019))

mon Vulnerabilities and Exposures (CVEs) (MITRE Corporation, 2019) that have been reported by Microsoft in the recent decade are the result of vulnerabilities related to *memory-safety* (see Figure 1.1).

COGENT is already equipped to deal with memory safety related problems. Memory safety can be divided into two main categories: temporal safety and spatial safety (Szekeres et al., 2013). Temporal safety issues involve accessing memory at the wrong time, such as trying to make use of uninitialised memory, or using memory after it has been freed. COGENT ensures temporal safety by having uniqueness types, a kind of *linear types*. Uniqueness types ensure that, at any given time, there is only one writable reference to a mutable object; this is done by requiring that variables of a linear type are used exactly once. Such a type system allows memory allocation to be tracked statically, enabling the COGENT compiler to ensure that memory is only accessed when it is safe to do so.

Spatial safety issues involve accessing memory at the wrong place. Common problems include buffer overflows and accessing arrays out-of-bounds, which creates opportunities for stack smashing. Spatial safety is ensured as a consequence of Cogent's type safety theorem, where unsafe operations are not exposed to the users.

Problems that are unrelated to memory safety can be split into two subcategories: those that are not observable on the programming language level, including hard-

ware problems such as Spectre (Kocher et al., 2019) and Meltdown (Lipp et al., 2018), as well as side channel attacks involving timing or power signatures; and problems that are observable on the programming language level, specifically, secret data being directly accessed or modified by unauthorised processes which violates *confidentiality* and *integrity*. The latter kind of vulnerability can be overlooked when trusted programs or applications are handling sensitive information, leaving potential risks to data breaches. FLOGENT aims to reduce these types of mistakes by extending Cogent with an information flow control system.

Chapter 2

Background

One of the very first attempts to express *information flow control* (IFC) was made by D. E. Denning and P. J. Denning (1977), where an information flow *policy* is defined as a set of *security classes* or *labels* for information and a *flow relation* defining the permissible flows between these classes in a *lattice* structure. Each runtime object is *bound* to a security class.[1] Operations that use an object, say x, to derive another, say y, are only allowed if the security class of x flows into that of y according to the flow relation. For example, the classes might be 'high privilege' and 'low privilege' where 'high' privileged data cannot leak to low. Our flow relations would be high to high and low to high, thus disallowing high to low. This control over data dependency is intended to ensure "the absence of information flows from private data to public observers" (Nipkow and Klein, 2014). Infomation flow control is an easy and potentially automatable method to check if programs protect private data such as passwords.[2]

This initial concept has been applied in a variety of domains. One such domain is that of *program logics*, proof calculi for software verification. Based on separation logic (Reynolds, 2002), Costanzo and Shao (2014) present a logic that allows for manual reasoning about IFC, either by pen-and-paper proof or in an interactive theorem prover. Murray et al. (2016) combine this program logic approach with dependent types to produce compositional verification framework. However, because COGENT generates an embedding in a purely functional subset of Isabelle/HOL, putting this embedding into a different logic would require reworking all of the existing infrastructure as well as losing the advantage of purely functional reasoning, such approaches are not well-suited to a COGENT context.

Another style of IFC is *dynamic* (i.e. run-time) security enforcement. Stefan et

[1] More recent literature calls this operation *labelling*.

[2] Note that this is under the assumption that the *trusted computing base* (TCB) is well-behaved.

al. (2011) describe a dynamic, language-based IFC mechanism implemented as a Haskell library called *Labeled IO* (LIO). LIO keeps track of labels by inserting them directly into run-time values and enforcing dynamic checks to ensure IFC policies are adhered to. The *Hybrid LIO* (HLIO) system of Buiras, Vytiniotis, and Russo (2015) extends this by enforcing some checks statically; however, any amount of dynamism would not be suitable in the context of COGENT. This is because COGENT is entirely statically typed and is used for low-level operating system components, where the performance overheads introduced by dynamic checking are not acceptable.

Volpano, Irvine, and Smith (1996) re-formulate this idea of IFC into a *type system* that enforces security properties, along with a proof of soundness. A simplified version of their security type system is approachably introduced by Nipkow and Klein (2014). Nipkow and Klein (2014) define security levels using natural numbers, whereas Volpano, Irvine, and Smith (1996) make use of a more general lattice structure. The soundness result ensures *noninterference* from high security level to low security level, i.e. high does not interfere with low, information only flows from low to high. Here, a judgement $\ell \vdash c$ denotes that the program c contains no information flows to variables lower than level ℓ and the only flows would be from variables greater than or equal to ℓ.

Abadi et al. (1999) generalise security type systems to accommodate a functional language with higher order functions. They propose a general calculus, specifically a λ-calculus for type-based dependency analyses called *Dependency Core Calculus* (DCC). They give DCC a denotational semantics that formalises the idea of noninterference. DCC incorporates a family of monads (Moggi, 1991), indexed by a security level, to express the different levels under which a program executes.

This calculus for static IFC was introduced by Russo (2015) and is adapted into a HASKELL (Haskell, 2010) library called MAC by Vassena et al. (2018). MAC places labels on the type level, and defines a HASKELL monad indexed by these type-level labels. In MAC, *labelled values*—values that are associated with a security level—are an explicit type, written *Labeled* ℓ a. This denotes a value of type a labelled at security class ℓ.

Figure 2.1 outlines the type signatures for the core operations on labelled values in MAC. The library makes use of the *MAC* monad, which associates the computation environment with a label. To create a labelled value, the join operation is used, which transforms a computation of level ℓ_H to a computation of lower level ℓ_L, but the result is labelled with the higher level ℓ_H. This means that the result of the computation cannot be accessed by those who do not have authorisation of level ℓ_H. They also define a simpler constructor for labelled values, label, which can be defined in terms of join as follows:

$$\begin{aligned}&\mathsf{join} &&:: \ell_L \leq \ell_H \Rightarrow \mathit{MAC}\ \ell_H\ a \to \mathit{MAC}\ \ell_L\ (\mathit{Labeled}\ \ell_H\ a)\\&\mathsf{unlabel} &&:: \ell_L \leq \ell_H \Rightarrow \mathit{Labeled}\ \ell_L\ a \to \mathit{MAC}\ \ell_H\ a\end{aligned}$$

Figure 2.1: Core API for MAC (Vassena et al., 2018)

$$\begin{aligned}&\mathsf{label} :: (\ell_L \leq \ell_H) \Rightarrow a \to \mathit{MAC}\ \ell_L\ (\mathit{Labeled}\ \ell_H\ a)\\&\mathsf{label} = \mathsf{join} \circ \mathsf{return}\end{aligned}$$

The second operation defined by MAC is unlabel. To unlabel a value of level ℓ_L, the computation level ℓ_H must be at least as high as ℓ_L.

As Haskell and Cogent are both functional languages with static types, the MAC approach is the most directly applicable to Cogent. However, the use of monads is not appropriate in a Cogent context, because monadic programming requires closures, which Cogent does not support (O'Connor et al., 2016). Since the use of monads is vital to the MAC approach, this presents a challenge for us, to implement the same functionality in Cogent.

There are other statically typed functional languages, such as ML (Milner, Tofte, and MacQueen, 1997) and Mercury (Somogyi, Henderson, and Conway, 1995), where monadic programming is also not common. Pottier and Simonet (2003) present an implementation of Abadi's framework for ML, suggesting that Abadi's framework can be applied without the use of monads.

2.1 Compromises in MAC

Any practical implementation of IFC will come with some caveats not present in the calculus of Abadi et al. (1999). For example, in MAC, a number of compromises had to be made in order to implement IFC in Haskell (Russo, 2015):

unsafePerformIO Haskell includes a built-in unsafe function, *unsafePerformIO*, which allows the user to perform arbitrary side-effects but hide them from Haskell's type system. In MAC, they must assume that any use of *unsafePerformIO* does not violate security properties. For Cogent, a similar assumption must be made about abstract functions imported from C.

Exceptions Haskell includes exceptions that can be thrown from pure functions. This means that it is possible for an attacker to learn secret information by observing which inputs cause an exception to be thrown. MAC includes some support for exceptions, but there are still scenarios where exceptions can lead to vulnerabilites (Hritcu et al., 2013). In Cogent, however, all

functions are total and exceptions are not permitted. Therefore, this is not a problem faced by FLOGENT.

Non-termination Similarly to exceptions, Haskell functions may fail to terminate. This also opens a potential side-channel through which secret information may be leaked. COGENT ensures termination of all functions which limits the problem of non-termination to only the use of abstract functions imported from C.

Because of COGENT's restrictive design, fewer compromises have to be made in the implementation of FLOGENT. This means that our type system can be more informative and better at detecting vulnerabilities, however care must be taken to ensure that the system remains expressive enough for practical use-cases.

Chapter 3

Flogent: Implementation

As we have established in previous chapters, FLOGENT aims to extend COGENT with an information flow control static type system. This includes design, compiler implementation, and formalisation. To ensure that the security properties hold (i.e. confidentiality and integrity) for all well-typed FLOGENT programs, these properties must also be characterised on top of our formalisation.

The main contribution so far in Thesis B is the design and implementation of two primitive operations within MINIGENT, a miniature version of COGENT. We call the operations join and unlock, derived from the join and unlabel operations of Abadi et al. (1999). These are sufficient to ensure information flow control in programming language libraries such as MAC (Vassena et al., 2018). Details of these operations are explained later in this chapter.

The calculus of Abadi et al. (1999) is widely implemented in functional languages (Vassena et al., 2018; Pottier and Simonet, 2003). It is foundational work in the field of information flow security and, according to experts in the field (Murray, 2019), this is a good basis for our work in COGENT. As mentioned in Chapter 2, MAC is a direct HASKELL implementation of this calculus. This adaptation is particularly useful because both HASKELL and COGENT are purely functional languages with a static type system.

3.1 Case Study: Password Confidentiality

To illustrate the design process and the usage of the two operations, we will examine a case study, first presented by Russo (2015). The model has been slightly modified for the example to be more relevant to this project.

We have Alice, a programmer who wants to write a password manager that asks the user for passwords. She wants to prevent users from using common passwords, so she wishes to include a feature that will tell the user whether or not the provided password is found in a list of common passwords. A colleague of hers, Bob, has already implemented such functionality in a different project and Alice wants to make use of that function in her code. Observe that Bob will need access to the password in order to provide this functionality—this breaks confidentiality as sensitive data is leaked. Therefore, we want to make sure that the password is somehow passed to Bob for the check without Bob being able to act maliciously, i.e. using this information for any other (potentially nefarious) purpose.

3.1.1 Implementation using MAC

Alice

```
password :: IO String
password =
  do putStr "Please, select your password:"
     pass ← getLine
     lbool ← runMAC $ do
          lpass ← label pass :: MAC L (Labeled H String))
          Bob.commonPass lpass
     let b = unRes lbool
     if then putStrLn "Your password is too common!" ≫ password
     else return pass
```

Recall in Figure 2.1 that the operations require the two arbitrary security levels ℓ_L and ℓ_H to satisfy the condition of $\ell_L \leq \ell_H$. For this example, we need only a two-element security lattice, so we instantiate ℓ_L to a concrete low level L and ℓ_H to a concrete high level H. First, Alice asks the user for a password. Once the password is obtained, it is then labelled at *H* using the label function, before passing to Bob, who only has access to operations of level *L*. The function *unRes*, which unsafely removes a label, is only permitted in functions that are part of the trusted computing base.

The code for Bob is presented below:

Bob

```
commonPass :: Labeled H String → MAC L (Labeled H Bool)
commonPass lpass =
  do str ← wgetMAC "http://common-password.list"
     let passwords = filter (not ∘ null) (lines str)
     join $ do pass ← unlabel lpass
               return $ isJust $ find (== pass) passwords
```

Here, Bob first fetches the list of common passwords via an HTTP request (the *wgetMAC* function). The password given to Bob is labelled at level *H*, so Bob can only perform the password comparison in a computation of level *H*. Bob uses join to transform this comparison computation to one of level *L*, however the resultant boolean value is now labelled at level *H*. This prevents Bob from leaking information, because during the password comparison computation, he would be unable to make outside requests of level *L*; and outside the password comparison computation, he would be unable to access the inputs or outputs of this secure computation.

3.2 Design

We want to design FLOGENT to have the same information flow control capabilities that MAC demonstrates in the example above. Despite the similarities between COGENT and HASKELL, there are two main differences: The first is the lack of support for monadic programming in COGENT. This necessitates a modification of MAC's approach to implement security-labelling in FLOGENT.

The second main difference is the uniqueness type system in COGENT, which is not a feature that is present in HASKELL or any widely-used security type system. As we will see, our final design for FLOGENT depends crucially on this feature of COGENT.

HASKELL makes use of monads primarily as a way to model side effects while maintaining a purely functional semantics. In COGENT, however, the uniqueness type system is used for this purpose. Specifically, effectful functions are given a type that takes a (model of the) world as input and produces a world as output (O'Connor et al., 2016). Figure 3.1 describes this translation, where the type of the world is denoted as 🌍.

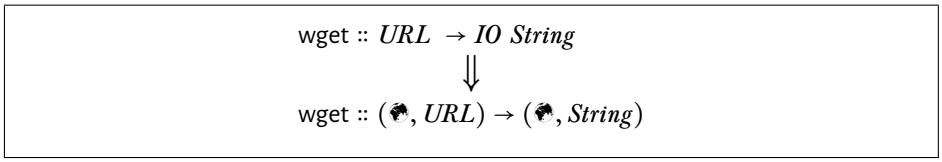


Figure 3.1: Using "World" instead of Monads

3.2.1 Using a "Key"

An intuitive way to represent authorisation for operations of a certain security level is to have a special type that acts like a *key*. This can be thought of as a witness of authorisation—analogous to possessing a key to a room to signify authorisation to access that room. Thus, the unlabel operation of Abadi et al. (1999) becomes unlock (*mutatis mutandis* for *Labeled* ℓ and Locked_ℓ).

Proposed type signatures based on this idea for the two primitive operations join and unlock are shown in Figure 3.2. Similarly to MAC, this design also has a lock function (cf. label) as follows:

$$\begin{aligned}&\mathsf{lock} :: a \to (\mathsf{Locked}_\ell\ a)\\&\mathsf{lock}\ a = \mathsf{join}\ (\lambda x \to a)\end{aligned}$$

Note that the high level computation in MAC is now represented as a function that accepts a *Key* type of the given level.

To return to our case study, with this design Alice's code would be as follows[1]:

[1]For the purposes of illustration, we use a hypothetical uniqueness-typed language with syntax superficially resembling Haskell.

$$\begin{aligned}\mathsf{join}\ &:: (\mathit{Key}\ \ell \to \tau) \to \mathsf{Locked}_\ell\ \tau\\\mathsf{unlock} &:: \mathit{Key}\ \ell \to \mathsf{Locked}_\ell\ \tau \to \tau\end{aligned}$$

Figure 3.2: Proposed type signatures using "Key"

Alice

```
password :: 🌍 → (🌍, Password)
password w0 = let w1          = putString "pick a password" w0
                  (w2, pass)  = getLine w1
                  lockedPass  = lock pass
                  (w3, b)     = Bob.commonPassword (w2, lockedPass)
              in if b then
                   let w4 = putString "your password is weak" w3
                   in password w4
                 else
                   (w3, pass)
```

Note that, unlike the MAC example, no monads are used, and instead a world value (🌍) is passed to every effectful function. Bob's code would be as follows:

Bob

```
commonPassword :: (🌍, Locked_H String) → (🌍, Locked_H Bool)
commonPassword (w0, lockedS) =
    let (w1, str) = wget "http://common-password.list" w0
        passwords = filter (not ∘ null) (lines str)
    in (w1, join (λ k → let s = unlock k lockedS
                        in elem s passwords))
```

Here, Bob uses the join operation to gain a key to the high level locked string, which he can then use to check if the password is in the list.

Confidentiality Loophole

This design ensures that high level Locked values are not accessed without authorisation, but it does not ensure that Bob does not communicate over insecure channels while authorised to access secure values.

For example, if Bob wished to leak passwords to his own server, he could make a malicious call to *wget* inside the join computation as follows:

Bob (Evil)

```
commonPassword (w0, lockedS) =
    let (w1, str) = wget "http://common-password.list" w0
        passwords = filter (not ∘ null) (lines str)
    in join (λ k → let s       = unlock k lockedS
                       (w2, _) = wget ("http://bob.evil/" ++ s) w1
                   in (w2, elem s passwords))
```

This means that this type system does not prevent low-level computations from being executed in a high-level context. There is one caveat, however: The type signature assigned to this computation will make the world inaccessible from a low-level context:

$$commonPassword :: (🌍, \mathsf{Locked}_H\ String) \to \mathsf{Locked}_H\ (🌍, Bool)$$

Such a type signature would be useless in most contexts, as such a locked up world would make it impossible to implement a top-level main function, which would be of type $🌍 \to 🌍$. This means that the security of any part of the program, however, depends on the type of the main function, which is non-compositional and potentially difficult to verify. Another drawback of this design is that the type errors presented to the user in these insecure scenarios are not located at the point of vulnerability. Rather, they are located at some point further up the call chain. In addition to making debugging more difficult, it can also mask vulnerabilities inside trusted functions. For example, suppose a trusted function f calls a function g, and the function g is insecure. This could lead to the function f having an insecure type, without explicitly pointing to g as the source of the vulnerability. Thus, because the function f is trusted, vulnerabilites in g could remain hidden from the programmer.

3.2.2 World Labelling

Rather than have a separate type to denote authorisation, we could instead tag the world itself with a security level. Here, the type signatures, shown in Figure 3.3, require each operation to take a world and return a world. This makes use of the uniqueness typing applied to 🌍 values: now, when the user calls join, they must provide their low-level 🌍 value, which, having been used, cannot be used again. This means that the only 🌍 value accessible to the user inside the join is the high-level one provided to the computation. This prevents the situation seen earlier, where Bob could invoke low-level computations from inside join. Furthermore,

```
join   :: ℓL ≤ ℓH ⇒ 🐏ℓL → (🐏ℓH → (🐏ℓH, τ)) → (🐏ℓL, Locked ℓH τ)
unlock :: ℓL ≤ ℓH ⇒ Locked ℓL τ → 🐏ℓH → (🐏ℓH, τ)
```

Figure 3.3: Implementation using World-Labelling

because the computation must return this high level 🐏 value, the high-level world cannot exist in the same scope as the low-level world.

As with all previous design iterations, we can implement the lock operation in terms of join:

```
lock :: (ℓL ≤ ℓH) ⇒ a → 🐏ℓL → (🐏ℓL, Locked ℓH a)
lock a = join (λ w → (w, a))
```

Returning to our case study again, we can now express the Alice code in much the same way as before, save that the lock operation now requires a world to be passed in:

Alice

```
password :: 🐏L → (🐏L, Password)
password w0 =
    let w1              = putString "pick a password" w0
        (w2, pass)      = getLine w1
        (w3, lockedPass) = lock pass w2
        (w4, b)         = Bob.commonPassword (w3, lockedPass)
    in if b then
         let w5 = putString "your password is weak" w4
         in password w5
       else
         (w4, pass)
```

For Bob's code, he must use the low-level world w_1 before gaining access to the high-level world w_2. While he has access to w_2 he is therefore unable to call

functions such as *wget*, which expect a low-level world:

Bob

```
commonPassword :: (🐏_L, Locked_H String) → (🐏_L, Locked_H Bool)
commonPassword (w0, lockedS) =
    let (w1, str) = wget "http://common-password.list" w0
        passwords = filter (not ∘ null) (lines str)
    in join w1 (λ w2 → let (w3, s) = unlock lockedS w2
                       in (w3, elem s passwords))

wget :: URL → 🐏_L → (🐏_L, String)
```

As we can see, this design addresses the main issues seen in the first iteration, and is also sufficiently expressive for the case study example. Furthermore, we can see a correspondence between this design and the monadic design of Abadi et al. (1999). Specifically, the monadic type that would be expressed in MAC as *MAC* ℓ τ can be expressed in our design as $🐏_\ell \to (🐏_\ell, \tau)$.

3.3 Implementation

The COGENT project includes a simpler compiler for a miniature dialect of COGENT called MINIGENT. While the language of MINIGENT does not lack any essential features from COGENT, the syntax is drastically reduced and additional compiler phases not relevant to type system experimentation are removed. This makes it a perfect candidate for a prototype implementation of FLOGENT.

In our earlier design sketches, we made extensive use of higher order functions to model the primitive operations for IFC. In COGENT (and MINIGENT), however, higher order functions can be quite cumbersome to use, due to the absence of support for closures. To sidestep this problem, we have implemented the fundamental IFC operations as built-in language constructs.

Implementation of these constructs in FLOGENT involves a number of milestones:

- Implementation of the syntactic changes required in the lexer and parser of the compiler.
- Formulation of the typing rules for these constructs.
- Implementation of the typing constraint generation phases for these constructs in the compiler's type checker.
- Solving of security level inequalities in the type constraint solver.

expressions	e	::=	$x \mid v \mid \ell$	*(O'Connor, 2019)*
		|	$e_1 \wr e_2$	*(primops)*
		|	$e_1\ e_2 \mid f[\overrightarrow{\tau_i}]$	*(applications)*
		|	**let** $x = e_1$ **in** e_2	
		|	**if** e_1 **then** e_2 **else** e_3	
		|	$e :: \tau$	*(type signature)*
		|	**unlock** $e_1\ e_2$	
		|	**join** $x \leftarrow e_1$ **in** e_2	
		|	**join•**	
		|	**f•**	
types	τ	::=		*(O'Connor, 2019)*
		|	a	*(type variables)*
		|	α	*(unification variables)*
		|	(τ_1, τ_2)	*(tuples)*
		|	🌍_ℓ	*(worlds)*
		|	$\mathsf{Locked}_\ell\ \tau$	*(secure data)*
literals		::=	True | False | $\mathbb{N}$	
constraints	C, A	::=	$\ldots$	*(O'Connor, 2019)*
		|	$\top$	*(tautology)*
		|	**Drop** τ	*(discardability)*
		|	$C_1 \wedge C_2$	*(conjunction)*
		|	$\tau_1 \sqsubseteq \tau_2$	*(subtyping)*
		|	$\ell_1 \leq \ell_2$	*(level order)*
levels	ℓ	::=	$L \mid H$	
contexts	Γ	::=	$\overline{x : \tau}$	
algorithmic contexts	G	::=	$\overline{x :_n \tau}$	
variables	x, y			
unification variables	α, β			
numbers	n	$\in$	$\mathbb{N}$	

$\overline{\text{overlines}}$ indicate repetition

Summary of Judgements	
$A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$	context splitting
$A, \Gamma \vdash e : \tau$	(non-algorithmic) typing rules
$A \vdash C$	constraint entailment
$G_1 \vdash e : \tau \rightsquigarrow C \mid G_2$	constraint generation

Figure 3.4: Summary of syntactic definitions

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A \vdash \ell_2 \leq \ell_1 \qquad A, \Gamma_1 \vdash e_1 : \text{🐏}_{\ell_1} \qquad A, \Gamma_2 \vdash e_2 : \mathsf{Locked}_{\ell_2}\ \tau}{A, \Gamma \vdash \textbf{unlock}\ e_1\ e_2 : (\text{🐏}_{\ell_1}, \tau)}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad A \vdash \ell_1 \leq \ell_2 \qquad A, \Gamma_1 \vdash e_1 : \text{🐏}_{\ell_1} \qquad A, w : \text{🐏}_{\ell_2}, \Gamma_2 \vdash e_2 : (\text{🐏}_{\ell_2}, \tau)}{A, \Gamma \vdash \textbf{join}\ w \leftarrow e_1\ \textbf{in}\ e_2 : (\text{🐏}_{\ell_1}, \mathsf{Locked}_{\ell_2}\ \tau)}$$

Figure 3.5: Typing rules

- Preliminary tests and examples to exercise the compiler with these new primitives.

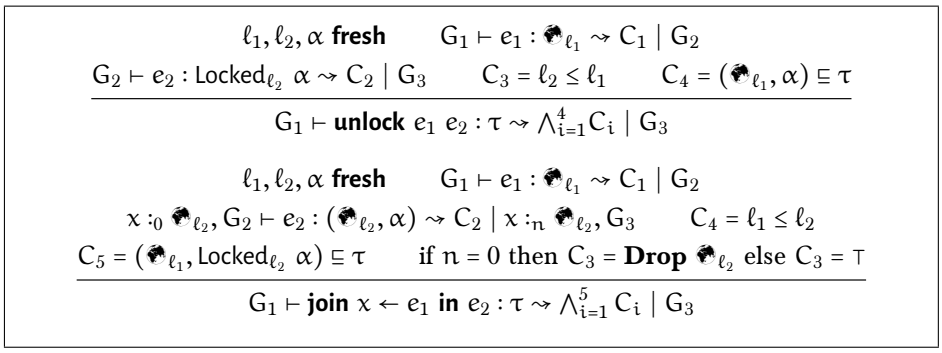
Figure 3.4 outlines the main syntactic changes made to MINIGENT. Broadly, this formalisation is derived from the formalisation by O'Connor (2019), where specific additions are highlighted in cyan. The previously higher-order function invocation join e_1 ($\lambda x \rightarrow e_2$) is now modelled as the built in operation **join** $x \leftarrow e_1$ **in** e_2. The expressions **join•** and **f•** are new additional expressions that are not yet implemented in MINIGENT, these are expressions that we will be using for *term erasure* in the formalisation technique described in Chapter 5.

3.3.1 Typing Rules

Figure 3.5 describes the typing rules for the two new constructs in MINIGENT. The typing judgement is parameterised by an additional constraint A which is the set of all assumptions about polymorphic type variables. This can include level inequalities (e.g. $\ell_L \leq \ell_H$) for functions such as lock, defined in subsection 3.2.2.

Because COGENT (and MINIGENT) has a uniqueness type system, care must be taken to avoid allowing unique variable bindings to be used more than once. Thus, we use the judgement $A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ to explicitly split a context Γ into two parts Γ_1 and Γ_2, where a binding can only be distributed to both Γ_1 and Γ_2 if it is *not* of a unique type. This ensures the uniqueness property, but makes the typing rules *non-algorithmic*—they do not straightforwardly correspond to a type checking algorithm. We describe how algorithmic type checking is achieved in the next subsection.

Other than these changes, the typing rules for these operations resemble the hypothetical type signatures described in subsection 3.2.2.



$$\frac{\ell_1, \ell_2, \alpha \textbf{ fresh} \qquad \Gamma_1 \vdash e_1 : \bullet_{\ell_1} \rightsquigarrow C_1 \mid \Gamma_2 \qquad \Gamma_2 \vdash e_2 : \mathsf{Locked}_{\ell_2}\ \alpha \rightsquigarrow C_2 \mid \Gamma_3 \qquad C_3 = \ell_2 \leq \ell_1 \qquad C_4 = (\bullet_{\ell_1}, \alpha) \sqsubseteq \tau}{\Gamma_1 \vdash \textbf{unlock}\ e_1\ e_2 : \tau \rightsquigarrow \bigwedge_{i=1}^{4} C_i \mid \Gamma_3}$$

$$\frac{\ell_1, \ell_2, \alpha \textbf{ fresh} \qquad \Gamma_1 \vdash e_1 : \bullet_{\ell_1} \rightsquigarrow C_1 \mid \Gamma_2 \qquad x :_0 \bullet_{\ell_2}, \Gamma_2 \vdash e_2 : (\bullet_{\ell_2}, \alpha) \rightsquigarrow C_2 \mid x :_n \bullet_{\ell_2}, \Gamma_3 \qquad C_4 = \ell_1 \leq \ell_2 \qquad C_5 = (\bullet_{\ell_1}, \mathsf{Locked}_{\ell_2}\ \alpha) \sqsubseteq \tau \qquad \text{if } n = 0 \text{ then } C_3 = \textbf{Drop}\ \bullet_{\ell_2} \text{ else } C_3 = \top}{\Gamma_1 \vdash \textbf{join}\ x \leftarrow e_1\ \textbf{in}\ e_2 : \tau \rightsquigarrow \bigwedge_{i=1}^{5} C_i \mid \Gamma_3}$$

Figure 3.6: Constraint Generation

3.3.2 Constraint Generation and Solving

To efficiently check well-typedness of an expression, COGENT (and MINIGENT) divide the type checking process into two phases:

1. Given an expression and a desired type (which may involve some unknown *unification variables*), generate a constraint such that the given expression has the desired type if and only if a satisfying assignment can be found to all unification variables in the constraint.
2. A constraint solving process to find such a satisfying assignment, if one exists.

Figure 3.6 describes the rules of the constraint generation process for **join** and **unlock**. Unlike the typing rules above, these rules describe a computation that has an input and output *algorithmic context.* The difference between an algorithmic context and a non-algorithmic context is that an algorithmic context is part of the type checker state, keeping track of the number of uses of each variable. The constraint generation judgement is written $\Gamma_1 \vdash e : \tau \rightsquigarrow C \mid \Gamma_2$. This is read as: given an input context Γ_1, the expression e, and the type τ, we generate the constraint C and an output context Γ_2.

The only type of constraint we now generate, that previously was not already handled by MINIGENT is the security level inequalities $\ell_1 \leq \ell_2$. Seeing as we have a simple two element lattice (with just L and H), solving these inequalities is trivial.

Chapter 4

Testing

To check that FLOGENT has the information flow control that we have aimed to achieve, we want to ensure that well-typed FLOGENT programs will preserve confidentiality: when secrets can only be read by processes with the right security clearance, and integrity: where data can only flow up the security level lattice. Any program that violates confidentiality or integrity should not be sucessfully type-checked by the compiler. Since these are implemented and tested in the MINIGENT compiler, the MINIGENT concrete syntax are used for the examples presented in this chapter.

4.1 Confidentiality

One example of a well-typed program `ReadSecret`, implemented in MINIGENT below, takes in the user's world at security level `H` and a file that is locked at level `H`; **take** is used to access the fields of records in a way that respects the linearity of the record fields. This function returns the world at the same level and the file that is now unlocked:

ReadSecret

```
readSecret: { world:  World H , input : Locked H U8 }#
                -> { world : World H , output:U8}#;
readSecret r = take r2 { world = w } = r in
        take r3 { input = i } = r2 in
            unlock w i
        end
        end;
```

On the other hand, if we try to implement a function `ReadSecretDodgy` that does the same thing but returns a world that has a security level lower than the input world, this program will not be type-checked. This is because once this function returns the unlocked file along with the lower level world, the file is now accessible to the world with lower security clearance when it should not be:

ReadSecretDodgy

```
readSecretDodgy: { world:  World H , input : Locked H U8 }#
                    -> { world : World L , output:U8}#;
readSecretDodgy r = take r2 { world = w } = r in
                take r3 { input = i } = r2 in
                  unlock w i
                end
              end;
```

Similarly, the type-checker will not allow an implementation of a function that unlock files which is locked with a higher security level than the level of the input world:

ReadSecretBad

```
readSecretBad : { world:  World L , input : Locked H U8 }#
                    -> { world : World L , output:U8}#;
readSecretBad r = take r2 { world = w } = r in
                 take r3 { input = i } = r2 in
                     unlock w i
                 end
                 end;
```

These three MINIGENT examples show the restrictions that FLOGENT placed upon the accessibility of secret data in order to preserve confidentiality.

4.2 Integrity

To preserve integrity, we want to make sure that infomation flows from low to high and not the other way around. The example below shows a function `WriteSecret` that takes in a World of level `L` and a file, then outputs a world of the same level and a `Locked` file of level `H`. This program is type-checked by the compiler as the function is writing data from low to high.

WriteSecret

```
writeSecret : [U8]. {world: World L, input: f}#
                        -> {world: World L, output: Locked H f}#;
writeSecret r = take r1 {world=w} = r
         in take r2 {input=i} = r1
            in join w1 <- w
               in {world = w1, output = i}
               end
            end
         end;
```

If we reverse the direction and try to leak data from high to low, we will get a function that will not type-check such as `WriteSecretBad` below.

WriteSecretBad

```
        writeSecretBad : [U8]. {world: World H, input: f}#
                           -> {world: World H, output: Locked L f}#;
writeSecretBad r = take r1 {world=w} = r
         in take r2 {input=i} = r1
            in join w1 <- w
               in {world = w1, output = i}
               end
            end
         end;
```

4.3 Test result

These examples are corner cases that illustrate how the MINIGENT type checker is capable of statically enforce both confidentiality and integrity at compile time, with no runtime checks required.

Chapter 5

Formalisation

So far in the previous chapters, we have seen the design, implementation and testing of FLOGENT. The next logical step is to formalise and prove the security guarentees that we want Flogent to have. This chapter presents the formalisation and proof techniques that we can use towards achieving that. The proof is not present as it is out of the scope of this thesis.

5.1 Noninterference

Noninterference (Goguen and Meseguer, 1982) is a property that restricts the infomation flow through a system that we want to preserve in FLOGENT; the formal definition of this property will be discussed later in this chapter.

A common proof technique used to prove noninterference in functional programs is known as *term erasure* (Li and Zdancewic, 2010). This technique uses an erasure function ε_ℓ on terms, which rewrites data above the reader's security level ℓ, into a special symbol $\bullet$, this represents data that are supposed to be secret to the reader will be hidden. I describe the erasure function in more detail later in subsection 5.1.1.

Once we have established the erasure function, the proof will involve the relationship between the erasure function and the reduction step, also known as the single-step simulation between the erased terms and the original ones.

We can intuitively understand this relationship by looking at the commutative diagram in Figure 5.1. In this figure, we can see that a program p can take a step to p' and then erase p' to reach $\varepsilon_\ell(\mathrm{p}')$, or the same program p can first be erased to $\varepsilon_\ell(\mathrm{p})$ then take a step and arrive at the same result $\varepsilon_\ell(\mathrm{p}')$. This is important

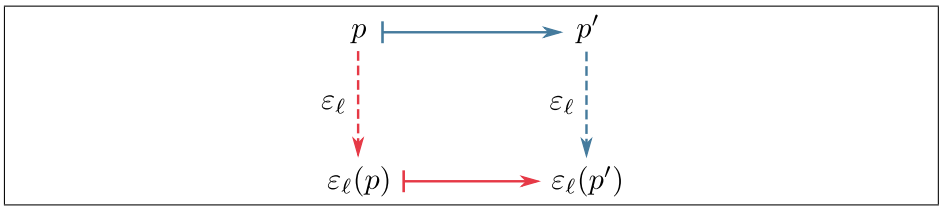


Figure 5.1: Single-step simulation

for proving noninterference as it shows that any erased secret will not affect the behaviour of the program.

Note that in order to reason with the reduction step, we will define the small step semantic for FLOGENT in subsection 5.1.2.

We can formally state the noninterference property using the following propositions and definition:

Proposition 1 (Single-Step Simulation).
If $p_1 \longmapsto p_2$ then $\varepsilon_\ell(p_1) \longmapsto \varepsilon_\ell(p_2)$.

Proposition 2 (Determinancy).
If $p_1 \longmapsto p_2$ and If $p_1 \longmapsto p_3$ then $p_2 \equiv p_3$.

Proposition 3 ($\approx_\ell$ Preservation).
If $c_1 \approx_\ell c_2,\ c_1 \longmapsto c_1',\ c_2 \longmapsto c_2',$ then $c_1' \approx_\ell c_2'$.

Definition (ℓ-equivalence).
Two program p_1 and p_2 are indistinguishable for someone at security level ℓ, written as $c_1 \approx_\ell c_2$, if and only if $\varepsilon_\ell\ (c_1) \equiv \varepsilon_\ell\ (c_2)$.

Definition (Big-step evaluation $\Downarrow$).
We will define the big-step evaluation ($\Downarrow$) as the reflexive transitive closure over small-step ($\longmapsto$) that results in a value.

Theorem (Noninterference Theorem).
If $c_1 \approx_\ell c_2,\ c_1 \Downarrow c_1'$ and $c_2 \Downarrow c_2',$ then $c_1' \approx_\ell c_2'$

Next, we will define the erasure function as well as small-step semantic rules.

5.1.1 Definition of the Erasure Function

This section presents the definition of the erasure function ε_ℓ for every expression and value. Figure 5.2 and Figure 3.4 show the summaries of the value semantics and the syntactic expressions.

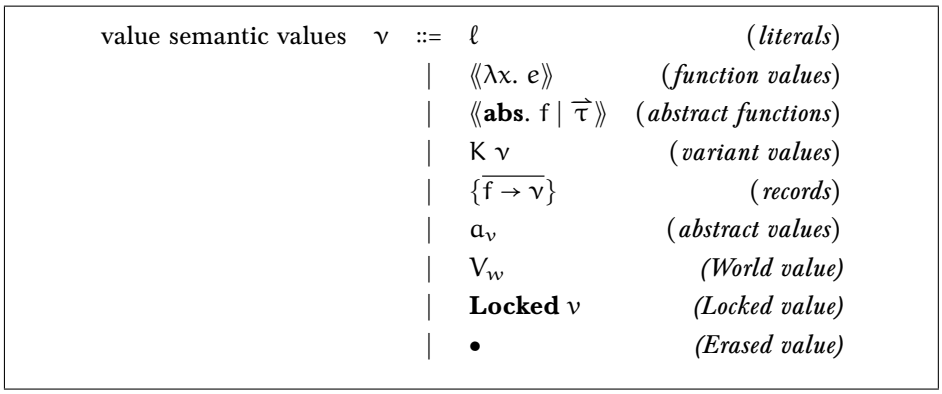


Figure 5.2: Summary of value semantics

Figure 5.3 shows the definitions for the erasure function when given different values and expressions. For most expressions, the erasure function is homomorphically applied to the sub-expressions (e.g. $\varepsilon_\ell(e_1 \wr e_2) = (\varepsilon_\ell(e_1) \wr \varepsilon_\ell(e_2))$). Primitive values (e.g. `True`) that are not sensitive will not be affected by the erasure function and the function will just return what it was given.

The more interesting cases for the erasure functions are cases where sensitive content is rewritten into $\bullet$ when it has a security level that is higher than the reader's access, ensuring that sensitive information is not visible to the reader (e.g. **Locked** v being erased to **Locked** $\bullet$). In the case of $f\bullet$, erasure is delayed until the function body is available and can be erased.

$$\varepsilon_\ell(x) = x \qquad \textit{(variable)}$$

$$\varepsilon_\ell(\ell) = \ell \qquad \textit{(literal)}$$

$$\varepsilon_\ell(v) = \begin{cases} \text{defined below} & \text{if } v = \textbf{Locked } v \\ v & \text{otherwise} \end{cases} \qquad \textit{(value)}$$

$$\varepsilon_\ell(V_w) = V_w \qquad \textit{(world value)}$$

$$\varepsilon_\ell(e_1 \wr e_2) = (\varepsilon_\ell(e_1) \wr \varepsilon_\ell(e_2)) \qquad \textit{(primop)}$$

$$\varepsilon_\ell(e_1\ e_2) = \varepsilon_\ell(e_1)\ \varepsilon_\ell(e_2) \qquad \textit{(application)}$$

$$\varepsilon_\ell(\textbf{let } x = e_1 \textbf{ in } e_2) = \textbf{let } x = \varepsilon_\ell(e_1) \textbf{ in } \varepsilon_\ell(e_2) \qquad \textit{(let)}$$

$$\varepsilon_\ell(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \textbf{if } \varepsilon_\ell(e_1) \textbf{ then } \varepsilon_\ell(e_2) \textbf{ else } \varepsilon_\ell(e_3) \qquad \textit{(if)}$$

$$\varepsilon_\ell(f[\overrightarrow{\tau_i}]) = f_\bullet[\overrightarrow{\tau_i}] \qquad \textit{(fun. app.)}$$

$$\varepsilon(\langle\!\langle \lambda x.\ e[\alpha_i := \overrightarrow{\tau_i}] \rangle\!\rangle) = \langle\!\langle \lambda x.\ \varepsilon(e[\alpha_i := \overrightarrow{\tau_i}]) \rangle\!\rangle \qquad \textit{(abs. function)}$$

$$\varepsilon_\ell(\textbf{join } e_1\ e_2\ ::\ \langle w, \textbf{Locked } \ell'\ \tau\rangle) = \begin{cases} \textbf{join}_\bullet\ \varepsilon_\ell(e_1)\ \varepsilon_\ell(e_2) & \text{if } \ell' > \ell \\ \textbf{join}\ \varepsilon_\ell(e_1)\ \varepsilon_\ell(e_2) & \text{otherwise} \end{cases}$$

Erasure for join.

$$\varepsilon_\ell(\textbf{Locked } v :: \textbf{Locked } \ell'\ \tau) = \begin{cases} \textbf{Locked } \bullet & \text{if } \ell' > \ell \\ \textbf{Locked } \varepsilon_\ell(v) & \text{otherwise} \end{cases}$$

Erasure for Locked value.

Figure 5.3: Erasure Function for expressions and values

5.1.2 Small-step Semantics

COGENT has a big-step operational semantics. To establish the reduction step that we need to investigate the single-step relationship, we present a small-step semantics for COGENT in Figure 5.4 and Figure 5.5 as well as for the new operations that we introduced to FLOGENT: Figure 5.6. The big-step evaluation is defined as the reflexive transitive closure over small-step that results in a value.

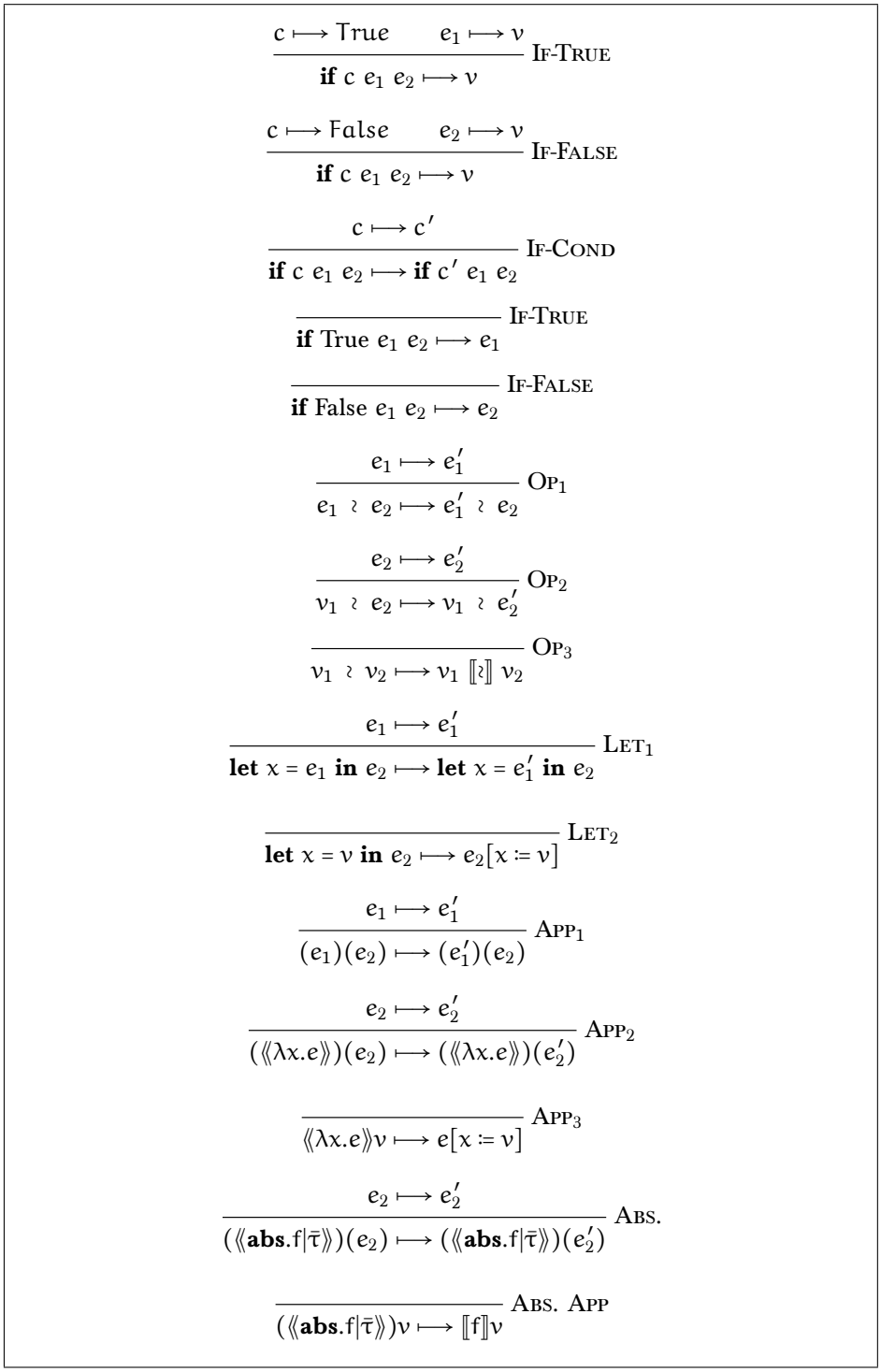


Figure 5.4: Small step semantics of the COGENT language (part 1)

$$\frac{e_1 \longmapsto e_1'}{\textbf{case}\ e_1\ \textbf{of}\ C\ x \to e_2\ \textbf{else}\ y \to e_3 \longmapsto \textbf{case}\ e_1'\ \textbf{of}\ C\ x \to e_2\ \textbf{else}\ y \to e_3}\ \textsc{Case}_1$$

$$\frac{}{\textbf{case}\ C\ v\ \textbf{of}\ C\ x \to e_2\ \textbf{else}\ y \to e_3 \longmapsto e_2[x \coloneqq v]}\ \textsc{Case}_2$$

$$\frac{B \neq C}{\textbf{case}\ B\ v\ \textbf{of}\ C\ x \to e_2\ \textbf{else}\ y \to e_3 \longmapsto e_3[y \coloneqq B\ v]}\ \textsc{Case}_3$$

$$\frac{e \longmapsto e'}{\textbf{esac}\ e \longmapsto \textbf{esac}\ e'}\ \textsc{Esac}_1$$

$$\frac{}{\textbf{esac}\ C\ v \longmapsto v}\ \textsc{Esac}_2$$

$$\frac{e_1 \longmapsto e_1'}{\textbf{take}\ x\ \{f_K = y\} = e_1\ \textbf{in}\ e_2 \longmapsto \textbf{take}\ x\ \{f_K = y\} = e_1'\ \textbf{in}\ e_2}\ \textsc{Take}_1$$

$$\frac{e_1 \longmapsto e_1'}{\textbf{take}\ x\ \{f_K = y\} = \overline{\{f_i = v_i\}}\ \textbf{in}\ e_2 \longmapsto e_2\ [x \coloneqq \overline{\{f_i = v_i\}}\ ][y \coloneqq v_K]}\ \textsc{Take}_2$$

$$\frac{e_1 \longmapsto e_1'}{\textbf{put}\ e_1.f_k \coloneqq e_2 \longmapsto \textbf{put}\ e_1'.f_k \coloneqq e_2}\ \textsc{Put}_1$$

$$\frac{e_2 \longmapsto e_2'}{\textbf{put}\ \{\overline{f_i = v_i}\}.f_k \coloneqq e_2 \longmapsto \textbf{put}\ \{\overline{f_i = v_i}\}.f_k \coloneqq e_2'}\ \textsc{Put}_2$$

$$\frac{\textbf{for}\ i \neq k : v_i' = v_i}{\textbf{put}\ \{\overline{f_i = v_i}\}.f_k \coloneqq v_k' \longmapsto \{\overline{f_i \coloneqq v_i'}\}}\ \textsc{Put}_3$$

$$\frac{\text{funDef}(f) = \langle f :: \forall(\overline{\alpha_i ::_K K_i}).\tau \longmapsto \tau_1' f x = e\rangle}{f[\overrightarrow{\tau_i}] \longmapsto \langle\!\langle \lambda x.e[\overrightarrow{\tau_1}/\alpha_i] \rangle\!\rangle}\ \textsc{VFunC}$$

$$\frac{e \longmapsto e'}{C\ e \longmapsto C\ e'}\ \textsc{VCons}$$

Figure 5.5: Small step semantics of the COGENT language (part 2)

$$\frac{e_1 \longmapsto e_1'}{\textbf{unlock}\ e_1\ e_2 \longmapsto \textbf{unlock}\ e_1'\ e_2}\ \textsc{Unlock}_1$$

$$\frac{e_2 \longmapsto e_2'}{\textbf{unlock}\ (\text{Locked}\ v_1)\ e_2 \longmapsto \textbf{unlock}\ (\text{Locked}\ v_1)\ e_2'}\ \textsc{Unlock}_2$$

$$\frac{}{\textbf{unlock}\ (\text{Locked}\ v_1)\ v_2 \longmapsto v_1\ v_2}\ \textsc{Unlock}_3$$

$$\frac{e_1 \longmapsto e_1'}{\textbf{join}\ w \leftarrow e_1\ \textbf{in}\ e_2 \longmapsto \textbf{join}\ w \leftarrow e_1'\ \textbf{in}\ e_2}\ \textsc{Join}_1$$

$$\frac{e_2[x\ :=\ v_w]\ \Downarrow (v_w', v)}{\textbf{join}\ x \leftarrow v_w\ \textbf{in}\ e_2 \longmapsto (v_w', \text{Locked}\ v)}\ \textsc{Join}_2$$

$$\frac{e_1 \longmapsto e_1'}{\textbf{join}\bullet\ w \leftarrow e_1\ \textbf{in}\ e_2 \longmapsto \textbf{join}\ w \leftarrow e_1'\ \textbf{in}\ e_2}\ \textsc{Join}\bullet_1$$

$$\frac{e_2[x\ :=\ v_w]\ \Downarrow (v_w', v)}{\textbf{join}\bullet\ x \leftarrow v_w\ \textbf{in}\ e_2 \longmapsto (v_w', \text{Locked}\ .)}\ \textsc{Join}\bullet_2$$

$$\frac{\text{funDef}(f) = \langle f :: \forall(\overline{\alpha_i ::_K K_i}).\tau \longmapsto \tau_1' f x = e\rangle}{f \bullet [\overrightarrow{\tau_i}] \longmapsto \varepsilon(\langle\!\langle \lambda x.e[\overrightarrow{\tau_1}/\alpha_i] \rangle\!\rangle)}\ \textsc{VFun}\bullet$$

Figure 5.6: Small step semantics of new operations

5.2 Applying to Flogent

So far, we have designed the type system for FLOGENT that uses linear types which enforces infomation flow control. We have a case study that conceptualising the security features we want to achieve and also tested the implementation on several edge cases which yeild results that we expected. We formalised the proof technique for proving noninterference which involve proving the single-step simulation relationship between program reduction step and term erasure. We defined a small step operational semantics for FLOGENT which allow us to reason with the reduction steps in FLOGENT. We intuitively believe that this system maintains confidentiality and integrity, however, futher work is required to look into other potential proof techniques or improving on the current single step proof technique for it to work on our system. Here is a concrete example of why the current approach does not work: Figure 5.7; this example shows the blue path does not hold in the commutative diagram. This is because ϵ_ℓ would not erase $< v_1, v_2 >$ into $< \bullet, v_2 >$ as the erasure function would not see v_1 as sensitive by itself. We will need to use a new technique to perform the security proofs since the existing technique does not apply to the current design in FLOGENT. Coming up with a novel proof technique to prove security in this context is beyond the scope of this thesis and remains open for future work.

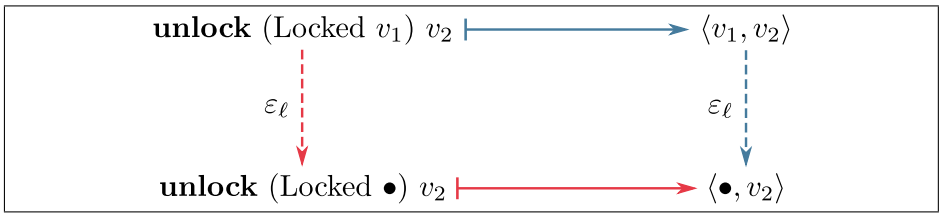


Figure 5.7: Single-step simulation for **unlock**

5.3 Summary

To summarise my contribution in this thesis, I extended MINIGENT with an infomation flow aware language FLOGENT with a type system that ultilise linear types. I implemented the core primitive operations **unlock** and **join** in MINIGENT, I conducted testing on some examples to check that confidentiality and integrity are preserved in different cases. I formalised the noninterference property that we want in FLOGENT which involves creating small-step semantics for the core language and new operations, defining the erasure function as well as investigating the single-step simulation relationship between the erasure function and the reduction step of a program.

Bibliography

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke (1999). "A Core Calculus of Dependency". In: *Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: ACM, pp. 147–160. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292555.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo (2015). "HLIO: Mixing Static and Dynamic Typing for Information-flow Control in Haskell". In: *International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, pp. 289–301. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784758.

David Costanzo and Zhong Shao (2014). "A Separation Logic for Enforcing Declarative Information Flow Control Policies". In: *Principles of Security and Trust*. Vol. 8414, pp. 179–198. DOI: 10.1007/978-3-642-54792-8_10.

Dorothy E. Denning and Peter J. Denning (1977). "Certification of Programs for Secure Information Flow". In: *Communications of the ACM* 20.7, pp. 504–513. ISSN: 0001-0782. DOI: 10.1145/359636.359712.

J. A. Goguen and J. Meseguer (1982). "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*, pp. 11–11.

Haskell (2010). *Haskell 2010 Language Report*. Ed. by Simon Marlow. URL: https://www.haskell.org/onlinereport/haskell2010/.

C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett (2013). "All Your IFCException Are Belong to Us". In: *IEEE Symposium on Security and Privacy*, pp. 3–17. DOI: 10.1109/SP.2013.10.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom (2019). "Spectre Attacks: Exploiting Speculative Execution". In: *Symposium on Security and Privacy*.

Peng Li and S. Zdancewic (2010). "Arrows for Secure Information Flow". In: *Theor. Comput. Sci.* Pp. 1974–1994. DOI: 10.1016/j.tcs.2010.01.025.

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval

Yarom, and Mike Hamburg (2018). “Meltdown: Reading Kernel Memory from User Space”. In: *USENIX Security Symposium*.

Matt Miller (2019). *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. Accessed: 28-04-2019. URL: https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL.

Robin Milner, Mads Tofte, and David MacQueen (1997). *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press. ISBN: 0262631814.

MITRE Corporation (2019). *Common Vulnerabilities and Exposures*. Accessed: 26-04-2019. URL: https://cve.mitre.org/.

Eugenio Moggi (1991). “Notions of Computation and Monads”. In: *Information and Computation* 93.1, pp. 55–92. ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90052-4.

Toby Murray (2019). *Private Correspondance*.

Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah (2016). “Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference”. In: pp. 417–431. DOI: 10.1109/CSF.2016.36.

Tobias Nipkow and Gerwin Klein (2014). *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated. ISBN: 978-3-319-10541-3.

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science.

Liam O’Connor (2019). *Type Systems for Systems Types*. UNSW Sydney.

Liam O’Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein (2016). “Refinement Through Restraint: Bringing Down the Cost of Verification”. In: *International Conference on Functional Programming*. ICFP 2016. Nara, Japan: ACM, pp. 89–102. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951940.

François Pottier and Vincent Simonet (2003). “Information Flow Inference for ML”. In: *Transactions in Programming Languages and Systems* 25.1, pp. 117–158. ISSN: 0164-0925. DOI: 10.1145/596980.596983.

John C. Reynolds (2002). “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Symposium on Logic in Computer Science*. LICS ’02. IEEE Computer Society, pp. 55–74. ISBN: 0-7695-1483-9.

Alejandro Russo (2015). “Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell”. In: *International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, pp. 280–288. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784756.

Z. Somogyi, F. J. Henderson, and T. C. Conway (1995). “Mercury, an Efficient Purely Declarative Logic Programming Language”. In: *Australian Computer Science Conference*, pp. 499–512.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières (2011). "Flexible Dynamic Information Flow Control in Haskell". In: *Haskell Symposium*. Haskell '11. Tokyo, Japan: ACM, pp. 95–106. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034688.

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song (2013). "SoK: Eternal War in Memory". In: *Symposium on Security and Privacy*. SP '13. IEEE Computer Society, pp. 48–62. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.13.

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye (2018). "MAC: A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95, pp. 148–180. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2017.12.003.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith (1996). "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2-3, pp. 167–187. ISSN: 0926-227X.