Flow Locks

Towards a Core Calculus for Dynamic Flow Policies

Niklas Broberg David Sands

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Sweden

DRAFT[*]
May, 2006

Abstract Security is rarely a static notion. What is considered to be confidential or untrusted data varies over time according to changing events and states. The static verification of secure information flow has been a popular theme in recent programming language research, but information flow policies considered are based on multilevel security which presents a static view of security levels. In this paper we introduce a very simple mechanism for specifying dynamic information flow policies, flow locks, which specify conditions under which data may be read by a certain actor. The interface between the policy and the code is via instructions which open and close flow locks. We present a type and effect system for an ML-like language with references which permits the completely static verification of flow lock policies, and prove that the system satisfies a semantic security property generalising noninterference. We show that this simple mechanism can represent a number of recently proposed information flow paradigms for declassification.

1 Introduction

Unlike access control policies, enforcing an information flow policy at run time is difficult because information flow is not a runtime property; we cannot in general characterise when an information leak is about to take place by simply observing the actions of a running system. From this perspective, statically determining the information-flow properties of a program is an appealing approach to ensuring secure information flow. However, security *policies*, in practice, are rarely static: a piece of data might only be untrusted until its signature has been verified; an activation key might be secret only until it has been paid for.

This paper introduces a simple policy specification mechanism based on the idea that the reading of storage location ℓ by certain actors (principals, levels) is guarded by boolean flags, which we call *flow locks*. For example, the policy $\ell_{\{High; paid \Rightarrow Low\}}$ says that ℓ can always be read by an actor with a high clearance level, and also by an actor with a low clearance level providing the "paid" lock is open.

[*] This is an extended version of an article in the 15th European Symposium on Programming, ESOP 2006, LNCS 3924

The interface between the flow lock policies and the security relevant parts of the program is provided by simple instructions for opening and closing locks. The program itself does not depend on the lock state, and the intention is that by statically verifying that the dynamic flow policy will not be violated, the lock state does not need to be computed at run time.[1]

In addition to the introduction of flow locks, the main contributions of this paper are:

- The definition of a type system for an ML-like language with references which permits the completely static verification of flow lock policies
- A formulation of the semantics of secure information flow for flow locks, and a proof that well typed programs are flow-lock secure (the reader is referred to the extended version of this article for the details).
- The demonstration that flow lock policies can represent a number of recently proposed information flow paradigms.

Regarding the last point, the work presented here can be viewed as a study of *declassification* mechanisms. In a recent study by Sabelfeld and Sands [18], declassification mechanisms are classified along four dimensions: *what* information is released, *who* releases information, *where* in the system information is released, and *when* information can be released. One of the key challenges stated in that work is to *combine* these dimensions. In fact, combination is perhaps not difficult; the real challenge is to combine these dimensions without simply amassing the combined complexities of the contributing approaches. Later in this paper we argue that flow locks can encode a number of recently proposed "declassification" paradigms, including the lexically scoped flow policies introduced by Almeida Matos and Boudol [2], Chong and Myers' notion of *noninterference until declassification* [5], and Zdancewic and Myers *robust declassification* [22, 13]. These examples, represent the "where", "when" and "who" dimensions of declassification, respectively, suggesting that flow locks have the potential to provide a core calculus of dynamic information flow policies.

The remainder of the paper is organised as follows. Section 2 gives an informal introduction to flow locks by showing a few motivating examples. In Section 3 we then present the system formally, and outline a semantic security condition in Section 4. Section 5 discusses related systems, with an emphasis on how we can use flow locks to encode them. Finally Section 6 concludes.

2 Motivating Examples

First let us assume we have a simple imperative language without any security control mechanisms of any kind. Borrowing an example from Chong and Myers [5], suppose we want to implement a system for online auctions with hidden bids in this language. We could write part of this system as the code on the right.

```
1 int aBid = getABid();
2 int bBid = getBBid();
3 makePublic(aBid);
4 makePublic(bBid);
5 ... decide winner + sell item
```

This surely works, but there is nothing in the language that prevents us from committing a serious security error. We could for instance accidently switch the lines 2 and 3, resulting in A's bid

[1] The term *dynamic* flow policy could have different interpretations. We use it in the sense that the flow policies vary over time, but they are still statically known at compile time.

being made public before B places her bid, giving B the chance to tailor her bid after A's.

Flow locks are a mechanism to ensure that these and other kinds of programming errors are caught and reported in a static check of the code.

The basic idea is very similar to what many other systems offer. To deny the flow of data to places where it was not meant to go, we annotate variables with policies that govern how the data held by those variables may be used. Looking back on our example, a proper policy annotation on the variable `aBid` could be $\{A;\texttt{BBid}\Rightarrow B\}$. The intuitive interpretation of this policy is that the data held by variable `aBid` may always be accessed by A, and may also be accessed by B whenever the condition `BBid`, that B has placed a bid, is fulfilled. `BBid` here is a *flow lock* — only if the lock is *open* can the data held by this variable flow to B. To know whether the lock is open or not we must look at how the functions for getting the bids could be implemented.

The function shown on the right first fetches the bid sent by A. We model the incoming channel as a global variable that can be read from, one with the same policy as `aBid`. When the bid has been read, the function signals this by opening the `ABid` lock—A has now placed a bid and the program can act accordingly. The implementation of `getBBid` follows the same pattern, and will result in `BBid` being open.

```
function getABid(){
  int {A;BBid⇒B} x
    = bidChanFromA;
  open ABid;
  return x;
}
```

```
function makePublic(bid){
  publicChannel = bid;
}
```

Now both bids have been placed and can thus be released. The `makePublic` function would be implemented as shown on the left. The outgoing `publicChannel` is also modelled as a global variable that can be written to. This one has the policy $\{A;B\}$ attached to it, denoting that both A and B will be able to access any data written into it. At the points in the program where `makePublic` is applied, both A and B will have placed their bids, the locks `ABid` and `BBid` will both be open, and the flows to the public channel will both be allowed. However, if the lines 2 and 3 were now accidently switched, it would be a different story. Then we would attempt to release A's bid, guarded by the policy $\{A;\texttt{BBid}\Rightarrow B\}$, onto the public channel with policy $\{A;B\}$. Since the flow lock `BBid` will then not yet be opened, this flow is illegal and the program can be rejected.

Taking the example one step further, assume that we have two items up for auction, one after the other. We can implement this rather naively as the program to the right. The locks `ABid` and `BBid` will both be opened on the first calls to the `getXBid` functions. But unless we have some means to reset them, there is again nothing to stop us from accidently switching lines to make our program insecure, this time lines 9 and 10. The same problem could also be seen from a different angle: what if the locks were already open when we got to this part of the program? Clearly we need a closing mechanism to go with the open. The function `auctionItem` could then be implemented as shown here. By closing the locks when an auction is initiated, we can rest assured that both A and B must place new bids for the new item before either bid is made public.

```
1  auctionItem(firstItem);
2  aBid = getABid();
3  bBid = getBBid();
4  makePublic(aBid);
5  makePublic(bBid);
6  ... decide winner + sell item
7  auctionItem(secondItem);
8  aBid = getABid();
9  bBid = getBBid();
10 makePublic(aBid);
11 makePublic(bBid);
12 ... decide winner + sell item
```

```
function auctionItem(item){
  close ABid, BBid;
  ... present item ...    }
```

It should be fairly easy to see that what we have here is a kind of state machine. The state at any program point is the set of locks that are open at that point, and the open and close statements form the state transitions. A clause $\sigma \Rightarrow A$ in a policy means that A may access any data guarded by that policy in any state where σ is open.

Our lock-based policies also give us an easy way to separate truly secret data from data that is currently secret, but that may be released to other actors under certain circumstances. Assume for instance that payment for auctioned items is done by credit card, and that the server stores credit card numbers in memory locations `aCCNum` and `bCCNum` respectively. Assume further that the line `aBid := aCCnum;` is inserted, either by sheer mistake or through malicious injection, just before where `aBid` is made public. This would release A's credit card number to B, however, the natural policy on `aCCNum` would be $\{A\}$, meaning only A may view this data, ever. Thus when we attempt the assignment above, it will be statically rejected since the policy on `aBid` is too permissive.

All the above are examples of policies to track confidentiality. The dual of confidentiality is integrity, i.e. deciding to what extent data can be trusted, and it should come as no surprise that flow locks can handle both kinds.

Returning to the example with the credit card, we assume that when A gives her credit card number, it must be validated (in some unspecified way) before we can trust it. To this end we introduce a "pseudo" actor T (for "trusted") who should only be allowed to read data that is fully trusted. We then use an intermediate location `tmpACCNum` to hold the credit card number when it is submitted by A. This location is given the policy $\{A; \texttt{ACCVal} \Rightarrow T\}$, stating that this data is trusted only if the lock `ACCVal` is open, which is done when the submitted number has been validated. Once validated we can transfer the value to `aCCNum`, which now has the policy $\{A; T\}$ stating that this data is trusted.[2]

3 A Secure Type and Effect System

In the previous section we used a simple imperative language to give an easy introduction to the concept of flow locks. In this section we define the type system for flow locks in the more general context of an ML-like language with recursion and references (but without polymorphism).

3.1 The language λ_{FL}

The terms and types of our language, dubbed λ_{FL}, are listed in Figure 1.

The policy language is worth some extra attention. The flow lock policies with which we work assumes a set of *actors* (or *levels, principals*) ranged over by A, B, and a set of flow locks ranged over by σ, with Σ for sets of locks. Both actors and flow locks are global in a program. A *policy* is a set of *clauses*, where each clause of the form $\Sigma \Rightarrow A$ states the circumstances (Σ) under which A may view the data governed by this policy. Σ is a set of locks which we name

[2] In order to prevent overwriting this data with a new number that hasn't been validated, we should also be sure to close the lock `ACCVal` once the assignment is done.

$$\begin{array}{ll}
\textbf{Policies:} & p ::= \{\, c_1; \dots; c_n\} \qquad c ::= \{\, \sigma_1, \dots, \sigma_k\} \Rightarrow A \\
\textbf{Values and types:} & v \;::=\; n \;\mid\; b \;\mid\; () \;\mid\; \lambda x.M \;\mid\; \ell_{p,\tau} \\
 & \tau \;::=\; \mathit{int} \;\mid\; \mathit{bool} \;\mid\; \mathit{unit} \;\mid\; (\tau, p) \xrightarrow{\Sigma,p,p,\Sigma} \tau \;\mid\; \mathit{ref}_p\, \tau \\
 & \\
\textbf{Terms:} & M \;::=\; v \mid x \mid MM \mid \textbf{if } M \textbf{ then } M \textbf{ else } M \mid \textbf{rec } x.M \\
 & \qquad \mid \textbf{ref}_{p,\tau}\; M \mid\; !M \mid M := M \mid \textbf{open } \sigma \mid \textbf{close } \sigma \\
 & \\
\textbf{Derived forms:} & \textbf{let } x = M_1 \textbf{ in } M_2 \equiv (\lambda x.M_2)M_1 \qquad M_1; M_2 \equiv (\lambda\_.M_2)M_1
\end{array}$$

Figure 1: The λ_{FL} language

the *guard* of the clause, and interpret it as a conjunction. Thus for the guard to be fulfilled, all the locks in Σ must be open. We can however have more than one clause for the same A, in which case the separate clauses also form a conjunction — A may read the data if either of the guards are fulfilled. In the special case where the guard contains no locks, signifying that the corresponding actor A may always view the data, we write the clause as only A instead of $\{\} \Rightarrow A$. From a logical perspective a policy is just a conjunction of definite Horn clauses, i.e. $\bigwedge_i \{\sigma_{i1} \wedge \cdots \wedge \sigma_{in} \Rightarrow A_i\}$. We implicitly identify policies up to logical equivalence.[3]

Now we can continue with the language itself. Apart from the terms from standard λ calculus with recursion, λ_{FL} has constructs for creating (ref), dereferencing (!) and assigning to (:=) memory locations ($\ell_{p,\tau}$) through references. In addition to the core terms, we can also derive a few useful language constructs as is also shown in Figure 1.

The reference creation construct takes an extra parameter p which is the policy that the contents should be governed by. The same parameter also shows up on the memory locations themselves, together with the base type τ of the contents. In many cases this τ is irrelevant, or clear from the context, and in those cases we omit it and just write ℓ_p. Function types are annotated with read and write policies, and start and end states, and arguments are annotated with a reading policy. We discuss the meaning of these when we define the type system. There are also the open and close terms for manipulation flow locks, thereby changing the state of the program.

The semantics of the language is standard, but apart from the term M and a memory μ, the configurations include the current state Σ. This state is the set of currently open locks, which are effected by the execution of **open** and **close** expressions. The small-step semantics of these are simply:

$$\langle \Sigma, \textbf{open } \sigma, \mu \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, (), \mu \rangle \qquad \langle \Sigma, \textbf{close } \sigma, \mu \rangle \rightarrow \langle \Sigma \setminus \{\sigma\}, (), \mu \rangle$$

It is important to note that the only interaction between a program and the lock state is via the open and close instructions. This is because we are aiming for a completely static verification — we include the lock state in the semantics only to be able to prove properties about flows, but the state is not actually represented at runtime. For this reason we also do not need to consider potential covert channels introduced by the flow lock state.

[3] It is worth noting that we do not allow negative flow policies. Our policy language is monotonic, i.e. the more locks that are open, the more flows are allowed.

3.2 Some intuitions about flow-lock security

Before we define our type system, it is useful to get some intuitions about which programs we deem secure/insecure. At this point we only concern ourselves with information leaks arising from direct or indirect data flows. In particular we will not consider timing or termination sensitivity.

(1) $\ell_{\{A\}} := !m_{\{B\}}$

(2) $\ell_{\{A;B\}} := !m_{\{B\}}$

(3) $\ell_{\{A\}} := !m_{\{A;B\}}$

(4) $\ell_{\{\sigma \Rightarrow A;B\}} := !m_{\{B\}}$

(5) $\ell_{\{A\}} := !m_{\{\sigma \Rightarrow A\}}$

A few small example programs are presented on the right. All of these contain insecure direct data flows, except (3). In (1) the contents of $m_{\{B\}}$ may only be read by B, but we are attempting to leak them into a location readable by A. Same thing goes for (2) — even though B can read the contents of the target location, we are still leaking the contents of $m_{\{B\}}$ to A. The simple pattern is that we may not write data to a memory location if that location may be read by someone who cannot already access the data. What's more, this should hold for future time as well. Thus if a reader could access the data from the location we are writing to in some future state, that reader must also have access to the data that is being written, in that same state. Thus the example $m_{\{\sigma \Rightarrow A\}} :=!\ell_{\{\sigma \Rightarrow A\}}$ is secure while program (4) is not. In program (5) we attempt to take data not yet readable by A, and put it in a location where A could read it right away. This should clearly not be allowed for the same reasons as for (4).

(6) $\textbf{open}\ \sigma; \ell_{\{A\}} := !m_{\{\sigma \Rightarrow A\}}$

(7) $\ell_{\{A\}} := \ (\textbf{open}\ \sigma; !m_{\{\sigma \Rightarrow A\}})$

The lock state in effect at the point of the assignment determines its validity, so the programs (6) and (7) are secure. However, we also want a program like (8) below to be considered secure, so we should take the policy of data read from some memory location to be the policy on the location, but taking into account the current state.

(8) $$\ell_{\{A\}} := \textbf{let}\ x = (\textbf{open}\ \sigma; !m_{\{\sigma \Rightarrow A\}})\ \textbf{in}\ (\textbf{close}\ \sigma; x)$$

In program (8) above, the data read from the reference will thus have the policy $\{A\}$ and not $\{\sigma \Rightarrow A\}$, since it is read in a state where σ is open.

Putting all this slightly more formally, data may be written to a memory location if and only if the policy on the location is at least as restrictive as the one on the data, with respect to the state in effect at the point of the assignment. We give a formal definition of this in the next section.

We must also handle indirect flows that arise from various branching situations. A very simple example program containing an invalid indirect flow is

(9) $$\textbf{if}\ !\ell_{\{A\}}\ \textbf{then}\ m_{\{B\}} := \textbf{true}\ \textbf{else}\ m_{\{B\}} := \textbf{false}$$

This program is obviously insecure since it will leak the value of $\ell_{\{A\}}$ into $m_{\{B\}}$, but for some programs it is not so easy to tell. Consider the three programs

(10) $\textbf{if}\ !\ell_{\{\sigma \Rightarrow A\}}\ \textbf{then}\ (\textbf{open}\ \sigma; m_{\{A\}} := \ \textbf{true})\ \textbf{else}\ (\textbf{open}\ \sigma; m_{\{A\}} := \ \textbf{false})$

(11) $\textbf{if}\ !\ell_{\{\sigma \Rightarrow A\}}\ \textbf{then}\ (\textbf{open}\ \sigma; m_{\{A\}} := \ \textbf{true}; \textbf{close}\ \sigma)\ \textbf{else}\ ()$

(12) $\textbf{if}\ (\textbf{open}\ \sigma; !\ell_{\{\sigma \Rightarrow A\}})\ \textbf{then}\ (\textbf{close}\ \sigma; m_{\{A\}} := \ \textbf{true})\ \textbf{else}\ ()$

Program (10) could be argued correct since at the points where we leak the information to A, i.e. the assignments, the state allows A to access the result of the branching conditional directly, and hence the leak is secure.

However, as program (11) shows it is not that simple. If the second branch in (11) is chosen, the value of the condition is still leaked to A by the absence of a write, but at no point does the state allow the flow. The leaks come from knowing which of the two branches is taken, which suggests that the leak actually occurs at the branch point. Thus it is the policy of the condition, taken in the state in effect at the branch point, that decides what writes the branches may perform. This means that (9), (10) and (11) are all insecure, while (12) is secure even though the lock is closed again before the write.

(13) $(!\ell_{\{A\}})\ ()$

(14) $(!\ell_{\{\sigma \Rightarrow A\}})\ ()$

(15) $(!\ell_{\{\sigma \Rightarrow A\}})\ (\mathbf{open}\ \sigma; ())$

(16) $(!\ell_{\{A\}}) :=\ 0$

(17) $(!\ell_{\{\sigma \Rightarrow A\}}) :=\ (\mathbf{open}\ \sigma; 0)$

(18) $(\lambda x.\ell_{\{B\}} :=\ x)\ (!m_{\{A\}})$

(19) $(\lambda x.\ell_{\{B\}} :=\ 0)\ (!m_{\{A\}})$

Another possible source of indirect leaks is function application. If the function itself is secret, an attacker could still get information about what that function is by observing its effects, just like he could know which branch was taken by observing the effects of a conditional expression. Thus in a sense we can view function application as a kind of branching.

Consider the programs (13) – (19). In the program (13) we must ensure that the function read from the reference does not write to locations visible by anyone other than A, otherwise we could leak information about which function that was used. As an example, if the function read from $\ell_{\{A\}}$ in (13) is $(\lambda x.m_{\{B\}} :=\ 1)$ or $(\lambda x.m_{\{B\}} :=\ 2)$, B can determine which of the two that was used by reading $m_{\{B\}}$. We treat the application point in the same way as the branch point of a conditional, so in program (14) the body of the function must not write to a location directly visible to A, even if it first opens σ. However, since we have a call-by-value semantics, in program (15) the function body may perform writes to locations directly visible to A, even if it first closes σ, since σ will be open at the application point.

A similar situation is assignment to a reference that in turn has been read from a reference, as illustrated in program (16) which should be disallowed if the reference read from $\ell_{\{A\}}$ is visible to anyone other than A. In particular, the contents of $\ell_{\{A\}}$ could be $m_{\{B\}}$ or $n_{\{B\}}$, in which case B can determine the contents of $\ell_{\{A\}}$ by checking which of the two latter locations that contain the value 0. However, just as for application, program (17) is secure if the reference assigned to has policy $\{A\}$, or any policy that is more restrictive than $\{A\}$, since σ is opened before the assignment takes place.

We also need to look at how functions handle the values passed to them as arguments. Clearly we want to rule out a direct leak in the function body, as the one in example (18). One solution attempt could be to rule out all functions that write to "low" memory, i.e. locations with less restrictive policies that the one placed on the argument. But this also rules out perfectly secure programs such as (19) which in particular would mean that we could not derive a sequential composition form as in figure 1 without placing too heavy restrictions on the writing capabilities of the second sub-program. Thus we want our type system to treat these two programs differently — (18) should be deemed insecure, but not (19).

Other issues such as whether our system is termination sensitive or timing sensitive (see [16] for an overview of these concepts) are orthogonal to the above discussion. We choose to develop a type system and semantics for termination and timing insensitive security. Termination insensitivity makes the type system simpler but the semantics more complex.

3.3 The Type System

Now we have all the intuition needed to construct the type system. We choose to model our system as a type and effect system in the style of Almeida Matos and Boudol [2]. This means in particular that all expressions will be given a *reading effect* and a *writing effect*. In our system the reading effect of an expression is a policy which states who may read the result of that expression, and in what lock states they may do so. The writing effect is also a policy, which records which actors and in what lock states they can see the memory effect of the expression's execution. Type judgments then have the form

$$\Gamma; \Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$$

- Γ is a typing environment for variables giving a type and policy for each variable.
- Σ is the state, i.e. the set of locks currently open.
- τ is the type of the term
- (r, w) are the reading and writing effects of the term, both on the form of policies
- Σ' is the state the program will be in after evaluating the term

First we need to define a few operators on policies that we will use in the typing rules. The aforementioned ordering of how restrictive policies are is defined as

$$p_1 \preceq p_2 \equiv \forall (\Sigma_2 \Rightarrow A) \in p_2 . \exists (\Sigma_1 \Rightarrow A) \in p_1 . \Sigma_1 \subseteq \Sigma_2$$

Read out, we say that p_1 is less restrictive than p_2 if and only if every clause in p_2 is matched by a clause in p_1 for the same A with a less restrictive guard (one with no additional locks). From the logical perspective, this ordering corresponds directly to implication. The most restrictive policy is $\{\}$, also written $\top$, and data with this policy can never be accessed by anyone. On the other end of the spectrum is $\bot$, defined as the set of all actors in the system. In other words, data marked with $\bot$ can be read by everyone at all times.

To join two policies means combining their respective clauses, thereby forming the logical disjunction. We define

$$p_1 \sqcup p_2 \equiv \{\Sigma_1 \cup \Sigma_2 \Rightarrow A \mid \Sigma_1 \Rightarrow A \in p_1,\ \Sigma_2 \Rightarrow A \in p_2\}$$

It should be intuitively clear that the join of two policies is at least as restrictive as each of the two operands, i.e. $p \preceq p \sqcup p'$ for all p, p'. In contrast, forming the union of two policies, i.e. the meet, corresponding to $\sqcap$ or logical conjunction, makes the result less restrictive, so we have $p \sqcap p' \preceq p$ for all p, p'. Both $\sqcap$ and $\sqcup$ are clearly commutative and associative.

Finally we need to define using a policy with respect to a particular state, or normalising to a state. We say that policy p normalised at state Σ is

$$p(\Sigma) \equiv \left\{\Sigma' \setminus \Sigma \Rightarrow A \mid \Sigma' \Rightarrow A \in p\right\}$$

Informally, we remove all open locks from all guards in p, since these no longer restrict data governed by p. This function is antimonotonic, so $\Sigma \subseteq \Sigma' \implies p(\Sigma') \preceq p(\Sigma)$, and in particular $p(\Sigma) \preceq p$ for all Σ. Logically this operation is a partial evaluation, where all variables (locks)

$$\frac{}{\Gamma;\Sigma \vdash n : \mathit{int}, (\bot,\top)\!\Rightarrow \Sigma} \qquad \frac{}{\Gamma;\Sigma \vdash b : \mathit{bool}, (\bot,\top)\!\Rightarrow \Sigma}$$

$$\frac{}{\Gamma;\Sigma \vdash \ell_{p,\tau} : \mathbf{ref}_p\ \tau), (\bot,\top)\!\Rightarrow \Sigma} \qquad \frac{}{\Gamma;\Sigma \vdash () : \mathit{unit}, (\bot,\top)\!\Rightarrow \Sigma}$$

$$\frac{\Gamma, x : (\tau, r_\alpha); \Delta \vdash M : \tau', (r,w)\!\Rightarrow \Delta'}{\Gamma;\Sigma \vdash \lambda x.M : (\tau, r_\alpha) \xrightarrow{\Delta,r,w,\Delta'} \tau', (\bot,\top)\!\Rightarrow \Sigma} \qquad \frac{x : (\tau, r) \in \Gamma}{\Gamma;\Sigma \vdash x : \tau, (r(\Sigma),\top)\!\Rightarrow \Sigma}$$

$$\frac{}{\Gamma;\Sigma \vdash \mathbf{open}\ \sigma : \mathit{unit}, (\bot,\top)\!\Rightarrow \Sigma \cup \{\sigma\}} \quad \frac{}{\Gamma;\Sigma \vdash \mathbf{close}\ \sigma : \mathit{unit}, (\bot,\top)\!\Rightarrow \Sigma \setminus \{\sigma\}}$$

$$\frac{\Gamma, x : (\tau, r); \Sigma \vdash M : \tau, (r,w)\!\Rightarrow \Sigma}{\Gamma;\Sigma \vdash \mathbf{rec}\ x.M : \tau, (r,w)\!\Rightarrow \Sigma}$$

$$\frac{\Gamma;\Sigma \vdash M : \tau, (r,w)\!\Rightarrow \Sigma'}{\Gamma;\Sigma \vdash \mathbf{ref}_p\ M : \mathbf{ref}_p\ \tau, (\bot, w \sqcap p)\!\Rightarrow \Sigma'} \qquad \frac{\Gamma;\Sigma \vdash M : \mathbf{ref}_p\ \tau, (r,w)\!\Rightarrow \Sigma'}{\Gamma;\Sigma \vdash\ !M : \tau, (r \sqcup p(\Sigma'), w)\!\Rightarrow \Sigma'}$$

$$\frac{\Gamma;\Sigma \vdash M_1 : \mathbf{ref}_p\ \tau, (r_1,w_1)\!\Rightarrow \Sigma' \quad \Gamma;\Sigma' \vdash M_2 : \tau, (r_2,w_2)\!\Rightarrow \Sigma''}{\Gamma;\Sigma \vdash M_1 := M_2 : \mathit{unit}, (\bot, w_1 \sqcap w_2 \sqcap p)\!\Rightarrow \Sigma''} \; r_1(\Sigma'') \sqcup r_2(\Sigma'') \preceq p$$

$$\frac{\Gamma;\Sigma \vdash M_0 : \mathit{bool}, (r_0,w_0)\!\Rightarrow \Sigma' \quad \Gamma;\Sigma' \vdash M_i : \tau, (r_i,w_i)\!\Rightarrow \Sigma_i \quad r_0(\Sigma') \preceq w_1 \sqcap w_2}{\Gamma;\Sigma \vdash \mathbf{if}\ M_0\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \tau, (r_0 \sqcup r_1 \sqcup r_2, w_0 \sqcap w_1 \sqcap w_2)\!\Rightarrow \Sigma_1 \cap \Sigma_2}$$

$$\frac{\begin{array}{c} r_1(\Sigma_2) \preceq w_f \\ \Gamma;\Sigma \vdash M_1 : (\tau, r_2) \xrightarrow{\Sigma_2,r_f,w_f,\Sigma_3} \tau', (r_1,w_1)\!\Rightarrow \Sigma_1 \quad \Gamma;\Sigma_1 \vdash M_2 : \tau, (r_2,w_2)\!\Rightarrow \Sigma_2 \end{array}}{\Gamma;\Sigma \vdash M_1\ M_2 : \tau', (r_1 \sqcup r_f, w_1 \sqcap w_2 \sqcap w_f)\!\Rightarrow \Sigma_3}$$

Figure 2: Type and Effect system

that appear in Σ are set to *true* in p.

The type and effect system is presented in Figure 2. The rules for literal values are straightforward, giving all such values the reading effect bottom. However, from the variable rule we see that variables are given a reading policy. This is used to keep track of the reading policies of function arguments, as can be seen from the rules for abstraction and application, and the purpose is to disallow programs like (18) while still allowing (19). It is important to note that we do *not* check that $r_2(\Sigma_2) \preceq w_f$ in the application rule, since doing so would invalidate program (19). Instead we rely on the type checking of the body of the function to find any leaks inside it, with the help of the annotation on its parameter.

In the rule for abstractions, we annotate the function arrow with the latent read and write effects that will be accurate for the function body once it is applied. We also annotate the arrow with the state that the program will be in at the application point, and the state the program will be in after evaluating the body. The interpretation of a function with type $(\tau, r_\alpha) \xrightarrow{\Delta,r,w,\Delta'} \tau'$ is thus that when applied in state Δ on an argument of type τ and with reading policy r_α, it will produce a result of type τ' with reading policy r. The writing policy w states who could see that the function has been applied, and the whole program will be in state Δ' afterwards. This is all mirrored by the appropriate states in the application rule.

Direct leaks, like the ones in programs (1), (2), (4) and (5), are handled by the check $r_2(\Sigma'') \preceq p$ in the rule for assignment. Since we normalise the policy r_2 of the assignee to the state in effect at the point of the assignment, program (5) would be secure if run in a state where σ is open, which is exactly what happens in programs (6) and (7). Also the normalisation to the current state in the dereferencing rule, i.e. $p(\Sigma')$ in the reading effect of the conclusion, means that program (8) will be deemed secure. The same kind of normalisation also appears in the variable rule.

The check $r_0(\Sigma') \preceq w_1 \sqcap w_2$ in the conditional rule will ensure that an indirect leak like the one in (9) will not be allowed. The normalisation of r_0 to Σ' means that it is the state at the branch point that is important, which disallows (10) and (11) but lets (12) through. The branches may open and close different locks, so the end states can differ. Since policies are monotonic, we can use the intersection of the end states as a safe approximation for the following program.

The checks $r_1(\Sigma'') \preceq p$ in the assignment rule, and the corresponding $r_1(\Sigma_2) \preceq w_f$ in the application rule handle indirect flows like in (13), (14) and (16), but allow (15) and (17).

In the assignment rule, the reading effect in the conclusion is $\perp$. The reason is that the result of an assignment is always (), independent of the result values of the two expressions M_1 and M_2, so no information is leaked by making the () result public. For similar reasons, r_2 does not show up in the reading effect in the conclusion of the application rule. Since function arguments are annotated with their reading effects, if the result of M_2 has any effect on the result of the whole application expression, this fact will be seen through r_f.[4]

4 Semantic Security Properties

In this section we define the semantic security property appropriate for flow locks, and outline the proof that the flow lock type system does indeed satisfy this property.

4.1 CORE$_{FL}$

The first observation we make, which we will explain in more depth in section 4.5, is that the λ_{FL} language and the given substitution semantics are not well suited when defining the semantic security property. In order to assert the properties we require, we need to be able to reason about values resulting from evaluating each subterm, and λ_{FL} does not give us the means to do this.

To this end we define a monadic core language, CORE$_{FL}$, defined in figure 3. The main difference from λ_{FL} is that we have made sequential computation explicit in the language by the introduction of a bind construct. All other terms in the language have been syntactically restricted to contain no subterms other than variables in positions suited for reduction. Another difference is that variables are now annotated with a policy and a type, just like locations. This means that references need not be typed since their type is given by the annotion on the variable argument. We use boldface metavariables $\boldsymbol{x}$, $\boldsymbol{y}$ etc., to range over policy- and type-annotated variables of the form $x_{p,\tau}$, $y_{p',\tau'}$.

[4] The rules involving functions are fairly restrictive as they are formulated here. One could easily imagine various forms of subsumption, both for lock states and argument policies, that would make the system less restrictive. However, adding subsumption would complicate the overall formulation of the type system, so we leave it for the full version of the paper.

$$
\begin{array}{lllllllllll}
\textbf{Annotated variables} & \boldsymbol{x} & ::= & x_{p,\tau} & & & & & & & \\
\textbf{Values and types:} & v & ::= & n & | & b & | & () & | & \lambda \boldsymbol{x}.M & | \ \ \ell_{p,\tau} \\
 & \tau & ::= & \mathit{int} & | & \mathit{bool} & | & \mathit{unit} & | & (\tau, p) \xrightarrow{\Sigma,p,p,\Sigma} \tau & | \ \ \mathit{ref}_p\, \tau \\
 & & & & & & & & & & \\
\textbf{Terms:} & M & \multicolumn{9}{l}{::= v \mid \boldsymbol{x} \mid \boldsymbol{x}\ \boldsymbol{y} \mid \textbf{if } \boldsymbol{x} \textbf{ then } M \textbf{ else } M \mid \textbf{rec } \boldsymbol{x}.v} \\
 & & \multicolumn{9}{l}{\quad \mid \textbf{ref}_p\ \boldsymbol{x} \mid !\boldsymbol{x} \mid \boldsymbol{x} := \boldsymbol{y} \mid \textbf{open } \sigma \mid \textbf{close } \sigma} \\
 & & \multicolumn{9}{l}{\quad \mid \textbf{bind } \boldsymbol{x} = M \textbf{ in } M}
\end{array}
$$

Figure 3: The CORE$_{FL}$ language

4.2 Semantics

The semantics for CORE$_{FL}$, presented in figure 4, and is given by single-step labelled transitions of the form

$$\langle \Sigma, M, S \rangle \xrightarrow{p} \langle \Sigma', N, S' \rangle$$

where

- Σ is the set of flow locks currently open,
- M is the term being computed,
- S is the *store*: a finite mapping from annotated values and locations to CORE$_{FL}$ values.
- p records the policy relating to any store access that that takes place during that step (and is simply $\top$ if there is no memory access in that step).

We assume the usual well-formedness conditions for configurations $\langle \Sigma, M, S \rangle$, namely that the free variables and the locations in M and in the range of S are in the domain of S.

We will write $\langle \Sigma, M, S \rangle \rightarrow \langle \Sigma', N, S' \rangle$ to mean $\exists p. \langle \Sigma, M, S \rangle \xrightarrow{p} \langle \Sigma', N, S' \rangle$, and $\langle \Sigma, M, S \rangle \Uparrow$ to mean that the configuration diverges – i.e. can be reduced indefinitely

$$\langle \Sigma, M, S \rangle \rightarrow \langle \Sigma_0, N_0, S_0 \rangle \rightarrow \cdots \rightarrow \langle \Sigma_i, N_i, S_i \rangle \rightarrow \cdots$$

4.3 Type System for CORE$_{FL}$

The type system for λ_{FL} is valid also for CORE$_{FL}$ terms with the addition of a typing rule for the bind construct. However, since CORE$_{FL}$ terms are simpler than their λ_{FL} counterparts, we can specialise the type rules for CORE$_{FL}$ terms, and use the simpler formulations to good effect in our proofs. The result of this specialisation can be found in figure 5. Note that the type environment is now redundant since each variable carries its type.

We can establish some standard properties relating well-typed programs and reduction: *progress*, which says that well-typed programs do not get "stuck", and *preservation* (subject reduction), which says roughly that well-typed terms reduce to well-typed terms. We simply state these properties as lemmas here while the proofs are given in appendix A.1.

$$\langle \Sigma, x_{p,\tau}, S\rangle \xrightarrow{p} \langle \Sigma, S(x_{p,\tau}), S\rangle$$

$$\langle \Sigma, \mathbf{ref}\ x_{p,\tau}, S\rangle \xrightarrow{\top} \langle \Sigma, \ell_{p,\tau}, S[\ell_{p,\tau} \mapsto S(x_{p,\tau})]\rangle \quad \ell_{p,\tau} \notin dom(S)$$

$$\langle \Sigma, !x_{p,\tau}, S\rangle \xrightarrow{p \sqcap p'} \langle \Sigma, S(S(x_{p,\tau})), S\rangle \quad \text{where } S(x_{p,\tau}) = \ell_{p',\tau'}$$

$$\langle \Sigma, \boldsymbol{x} := \boldsymbol{y}, S\rangle \xrightarrow{\top} \langle \Sigma, (), S[S(\boldsymbol{x}) \mapsto S(\boldsymbol{y})]\rangle$$

$$\langle \Sigma, \mathbf{if}\ x_{p,\tau}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, S\rangle \xrightarrow{p} \langle \Sigma, M_0, S\rangle \quad \text{if } S(x_{p,\tau}) = \mathbf{true}$$

$$\langle \Sigma, \mathbf{if}\ x_{p,\tau}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, S\rangle \xrightarrow{p} \langle \Sigma, M_1, S\rangle \quad \text{if } S(x_{p,\tau}) = \mathbf{false}$$

$$\langle \Sigma, x_{p,\tau}\ \boldsymbol{y}, S\rangle \xrightarrow{p} \langle \Sigma, M[\boldsymbol{y}/\boldsymbol{z}], S\rangle \quad \text{where } S(x_{p,\tau}) = \lambda \boldsymbol{z}.M,\ \boldsymbol{z} \text{ fresh}$$

$$\langle \Sigma, \mathbf{open}\ \sigma, S\rangle \xrightarrow{\top} \langle \Sigma \cup \{\sigma\}, (), S\rangle$$

$$\langle \Sigma, \mathbf{close}\ \sigma, S\rangle \xrightarrow{\top} \langle \Sigma \setminus \{\sigma\}, (), S\rangle$$

$$\langle \Sigma, \mathbf{rec}\ \boldsymbol{x}.v, S\rangle \xrightarrow{\top} \langle \Sigma, v, S[\boldsymbol{x} \mapsto v]\rangle$$

$$\langle \Sigma, \mathbf{bind}\ \boldsymbol{x} = v\ \mathbf{in}\ M, S\rangle \xrightarrow{\top} \langle \Sigma, M, S[\boldsymbol{x} \mapsto v]\rangle \quad \boldsymbol{x} \notin dom(S)$$

$$\frac{\langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle}{\langle \Sigma, \mathbf{bind}\ \boldsymbol{x} = M\ \mathbf{in}\ N, S\rangle \xrightarrow{p} \langle \Sigma', \mathbf{bind}\ \boldsymbol{x} = M'\ \mathbf{in}\ N, S'\rangle}$$

Figure 4: Store-based semantics for CORE$_{FL}$

Lemma 1 (Progress). *If* $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ *then either*

- $M \in \mathrm{Val}$, *or*
- *for all* S *such that* $\mathrm{dom}(S) \supseteq \mathrm{fv}(M) \cup \mathrm{loc}(M)$ *and* $\vdash S$ *then* $\exists \Sigma', M', S'. \langle \Sigma, M, S\rangle \to \langle \Sigma', M', S'\rangle$.

Lemma 2 (Preservation). *If* $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ *and* $\vdash S$ *and* $\mathrm{dom}(S) \supseteq \mathrm{fv}(M) \cup \mathrm{loc}(M)$ *and* $\langle \Sigma, M, S\rangle \to \langle \Sigma', M', S'\rangle$ *then* $\vdash S'$ *and* $\Sigma' \vdash M' : \tau, (r', w') \Rightarrow \Delta$ *where* $r' \preceq r$ *and* $w \preceq w'$.

4.4 Semantic security property

To prove standard noninterference one needs to show that the observable behaviour of a program, from the perspective of a given actor, does not change when the values of secrets (things not readable by that actor) are changed. At the top level we may settle for a notion of "observable behaviour" to mean the results of computations — the final state or values.

In the next section we will show that our notion of flow lock security does indeed imply a standard noninterference property. However, since we have dynamic policies we are forced to consider the intermediate states of a computation, because it is at such state that the policy may change.

Visibility An actor α can directly observe the contents of a memory location $\ell_{p,\tau}$ in lock state Σ, when there is a clause $\Sigma' \Rightarrow \alpha \in p$ such that $\Sigma' \subseteq \Sigma$, or equivalently, when $\{\} \Rightarrow \alpha \in p(\Sigma)$. In this case we sometimes say that *α can see p at Σ*. This kind of property is used often, so we introduce some specific notations:

$$\frac{}{\Sigma \vdash n : \mathit{int}, (\bot, \top) \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash b : \mathit{bool}, (\bot, \top) \Rightarrow \Sigma}$$

$$\frac{}{\Sigma \vdash \ell_{p,\tau} : \mathit{ref}_p\, \tau, (\bot, \top) \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash () : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma}$$

$$\frac{\Delta \vdash M : \tau, (r_f, w_f) \Rightarrow \Delta'}{\Sigma \vdash \lambda x_{p',\tau'}.M : (\tau', p') \xrightarrow{\Delta, r_f, w_f, \Delta'} \tau, (\bot, \top) \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash x_{p,\tau} : \tau, (p(\Sigma), \top) \Rightarrow \Sigma}$$

$$\frac{}{\Sigma \vdash \mathbf{open}\ \sigma : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma \cup \{\sigma\}} \qquad \frac{}{\Sigma \vdash \mathbf{close}\ \sigma : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma \setminus \{\sigma\}}$$

$$\frac{\Sigma \vdash v : \tau, (\bot, \top) \Rightarrow \Sigma}{\Sigma \vdash \mathbf{rec}\ x_{\bot,\tau}.v : \tau, (\bot, \top) \Rightarrow \Sigma}$$

$$\frac{p(\Sigma) \preceq p'}{\Sigma \vdash \mathbf{ref}_{p'}\ x_{p,\tau} : \mathbf{ref}_{p'}\ \tau, (\bot, p') \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash !x_{p,\mathit{ref}_{p'}\,\tau} : \tau, (p(\Sigma) \sqcup p'(\Sigma), \top) \Rightarrow \Sigma}$$

$$\frac{p(\Sigma'') \sqcup p'(\Sigma'') \preceq p''}{\Sigma \vdash x_{p,\mathit{ref}_{p''}\,\tau} := y_{p',\tau'} : \mathit{unit}, (\bot, p'') \Rightarrow \Sigma}$$

$$\frac{\Sigma' \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Sigma' \qquad p(\Sigma) \preceq w_0 \sqcap w_1}{\Sigma \vdash \mathbf{if}\ x_{p,\mathit{bool}}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1 : \tau, (r_0 \sqcup r_1 \sqcup p(\Sigma), w_0 \sqcap w_1) \Rightarrow \Sigma'}$$

$$\frac{p(\Sigma) \preceq w_f}{\Sigma \vdash x_{p,\tau_f}\ y_{p',\tau'} : \tau, (p(\Sigma) \sqcup r_f, w_f) \Rightarrow \Sigma'}\ \mathit{where}\ \tau_f = (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau$$

$$\frac{\Sigma \vdash M_0 : \tau, (r_0, w_0) \Rightarrow \Sigma' \quad \Sigma' \vdash M_1 : \tau', (r_1, w_1) \Rightarrow \Sigma'' \quad r_0(\Sigma') \preceq p}{\Sigma \vdash \mathbf{bind}\ x_{p,\tau} = M_0\ \mathbf{in}\ M_1 : \tau', (r_1, w_0 \sqcap w_1) \Rightarrow \Sigma''}$$

Figure 5: Specialized Type and Effect system for $CORE_{FL}$

Definition 1 (Visibility).

$$\alpha \lhd p \overset{\text{def}}{=} (\{\} \Rightarrow \alpha) \in p \qquad (\alpha \text{ can see } p)$$

$$\alpha \not\lhd p \overset{\text{def}}{=} \neg(\alpha \lhd p) \qquad (\alpha \text{ can't see } p)$$

$$\alpha \not\lhd^{\Omega} p \overset{\text{def}}{=} \forall \Theta . \alpha \not\lhd p(\Theta \setminus \Omega) \qquad (\alpha \text{ can't see } p \text{ without } \Omega)$$

$$\mathit{guards}_{\alpha}(p) \overset{\text{def}}{=} \begin{cases} \{\{\}\} & \mathit{if}\ \alpha \lhd p \\ \{\Phi \mid \Phi \Rightarrow \alpha \in p\} & \mathit{otherwise} \end{cases} \qquad (\text{The guards of } \alpha \text{ in } p)$$

The last of these definitions, the guards of an actor α in policy p, definies the sets of locks which have an influence on the visibility of the policy to α. We can connect the guards of a policy and its visibility through the following lemma:

Lemma 3 (Guard lemma). *If $\alpha \not\lhd p$, then $\alpha \not\lhd^{\Omega} p$ where $\Omega = \bigcup \mathit{guards}_{\alpha}(p)$.*

The proof of this lemma can be found in appendix A.2

For the visibility operators we note that the $\alpha \lhd p$ relation is anti-monotonic in its policy argument, i.e.

$$\alpha \lhd p \,\&\, p' \preceq p \implies \alpha \lhd p'$$

Clearly $\alpha \ntriangleleft p$ and $\alpha \ntriangleleft^{\Omega} p$ are then monotonic. Also $\alpha \ntriangleleft^{\Omega} p$ is monotonic in its lock-set argument, i.e.

$$\alpha \ntriangleleft^{\Omega} p \;\&\; \Omega' \supseteq \Omega \Longrightarrow \alpha \ntriangleleft^{\Omega'} p$$

Actor indistinguishable stores In order to charactersise when information has leaked we first need to characterise when two stores are indistinguishable for a given actor. In order to do this we need to take into account which locks are open. Once we know wich locks are open we can compute which parts of the store are visible to the actor.

Definition 2 (α-indistinguishable stores $=^{\Theta}_{\alpha}$). Define two stores S and T to be indistinguishable by α at lock state Θ, written $S =^{\Theta}_{\alpha} T$, if the location domains of S and T are the same, and for all policies p such that $\alpha \ \lhd \ p(\Theta)$,

1. for all locations $\ell_{p,\tau}$ in S and T we have $S(\ell_{p,\tau}) = T(\ell_{p,\tau})$, and
2. for all variables $x_{p,\tau} \in \mathrm{dom}(S) \cap \mathrm{dom}(T)$ we have $S(x_{p,\tau}) = T(x_{p,\tau})$.

The definition asserts the equality, in S and T respectively, of locations $\ell_{p,\tau}$ and variables $x_{p,\tau}$ which are visible to actor α at lock state Θ. The stronger requirement on locations – that S and T have the same locations – is due to the fact that locations are first class values that can be passed around and inspected, and their values can be updated, so an actor can potentially observe the presence or absence of a given memory location in a store. Variables on the other hand can never be observed directly.

The relation $=^{\Theta}_{\alpha}$ is not transitive in general since the domains may vary freely in the parts that deal with variables. As an example of this we could have $\{x_{\perp,\tau} \mapsto v\} =^{\Theta}_{\alpha} \{\}$ and $\{x_{\perp,\tau} \mapsto v'\} =^{\Theta}_{\alpha} \{\}$, but clearly not $\{x_{\perp,\tau} \mapsto v\} =^{\Theta}_{\alpha} \{x_{\perp,\tau} \mapsto v'\}$.

However, we are going to need to argue about transitivity in our proofs, so we need to assert that transitivity holds for a certain domain of memories. In particular we can show that $S =^{\Theta}_{\alpha} S'$ and $S =^{\Theta}_{\alpha} T$ gives $S' =^{\Theta}_{\alpha} T$, assuming that $\mathrm{dom}(S') \backslash \mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$. We would then have that $\mathrm{dom}(S') \cap \mathrm{dom}(T) \subseteq \mathrm{dom}(S) \cap \mathrm{dom}(T)$, and thus for all variables $x_{p,\tau} \in \mathrm{dom}(S') \cap \mathrm{dom}(T)$ we have $S'(x_{p,\tau}) = T(x_{p,\tau})$ as required.

Whenever we argue transitivity in our proofs, we implicitly mean this restricted form, but the condition on domains will always be true in the contexts where we use it.

Flow lock security Our definition of flow-lock security follows the "self-bisimulation" approach from [17], whereby security is characterised by a more general property of two programs being bisimilar with respect the the observable parts of memory. One particular feature of the definition from [17] is that the bisimulation is defined over programs and not configurations (program-memory pairs). The idea is that at each step of the bisimulation the pair of programs under comparison are inspected in all pairs of memory states which are indistinguishable to the attacker. This very strong requirement was needed to make the definition of security compositional with respect to parallel composition. But this approach of "resetting" the store at each step has another very useful property: it enables us to reset the state in the event of a policy change. For example, one particular difficulty is that when the current policy becomes *more* restrictive — in our case when locks are closed — then we need a way to reestablish a stronger security requirement at that point in the execution. It is notable that two previous semantic accounts of temporary policy weakening mechanisms, Mantel and

Sands's language based intransitive noninterference condition [8], and Almeida Matos and Boudol's *nondisclosure* policy [2], both rely on such a "resetting" bisimulation not only to deal with threads, but more importantly to provide a semantics to local policy change mechanisms. Our definition is close in spirit to Almeida Matos and Boudol's definition, although our less structured (more general) policy-change mechanism creates additional problems.

Without further ado, we now provide the definition of bisimulation upon which our notion of security is based.

Definition 3 ($\sim_\alpha^\Omega$). For any actor α let $\{\sim_\alpha^\Omega\}$ be the lock-set indexed family (i.e. Ω is a set of locks) of relations defined to be the largest symmetric relations on *preconfigurarions* (lockstate-term pairs) such that if

$$\langle \Sigma, M\rangle \sim_\alpha^\Omega \langle \Delta, N\rangle \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle$$
$$\& \; \Theta \supseteq \Sigma \;\&\; S =_\alpha^\Theta T \;\&\; \mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$$

then there exists Δ', N', T' such that

either $\langle \Delta, N, T\rangle \rightarrow^* \langle \Delta', N', T'\rangle \;\&\; S' =_\alpha^{\Theta\backslash\Omega} T' \;\&\; \langle \Sigma', M'\rangle \sim_\alpha^{\Omega'} \langle \Delta', N'\rangle$,

or $\langle \Delta, N, T\rangle \Uparrow$,

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

Now we can state that a program is secure if and only if it is bisimilar to itself:

Definition 4 (Flow-lock security). We say that a term M is flow-lock secure, written $M \in$ *FL*, if and only if $\langle \{\}, M\rangle \sim_\alpha^{\{\}} \langle \{\}, M\rangle$

4.5 The bisimulation definition explained

We will try to explain our definition using a sequence of "attempts", each of which introduces parts of the final solution. These are:

1. Bisimulation up to nontermination – adding termination insensitivity to a configuration-level bisimulation-based noninterference condition.
2. A location-resetting bisimulation – adding lock states to the bisimulation, and motivating the "resetting" style of bisimulation.
3. A store-resetting bisimulation – motivating why we have to reset not only the locations but also the variables
4. Future-sensitive bisimulation – why we have to quantify over all lock states which include the current lock state;
5. Past-sensitive bisimulation – why we have to add the lockset Ω.

Let us begin with a view of an attacker (an actor) who can observe intermediate states of computation, but not the speed of computation. Let us further suppose a simple semantics without lockstate, and in which the state is just a mapping for locations (ranged over by μ and ν), and that there are no free variables in the term (i.e. we have a suubstitution semantics). Intuitively, any program when run with two inputs which are indistinguishable to an actor should produce intermediate states indistinguishable to that actor. With no flow locks and only static policies, a possible bisimulation formulation could be of the form:

Attempt 1 (Bisimulation up to nontermination). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation such that if $\langle M, \mu \rangle \sim_\alpha \langle N, \nu \rangle$ then $\mu =_\alpha \nu$, and if $\langle M, \mu \rangle \to \langle M', \mu' \rangle$ then there exists N', ν' such that either $\langle N, \nu \rangle \to^* \langle N', \nu' \rangle$ and $\langle M', \mu' \rangle \sim_\alpha \langle N', \nu' \rangle$, or $\langle N, \nu \rangle \Uparrow$.

We use here the obvious notion of low-equivalence of stores, $=_\alpha$, which ensures that we start with inputs that do not differ in the public parts, i.e. locations visible to α. To match a single computation step from the first configuration we can take zero or more steps. This makes the definition insensitive to timing issues. The divergence clause is added simply to make the definition termination insensitive, so that we cannot (by choice) detect leaks which are encoded in the termination behaviour alone.

This definition is clearly inadequate in the presence of locks. Our next step is to observe that we need to define the bisimulation relation over $\langle \Sigma, M \rangle$ pairs, which we call *preconfigurations*. This is because in order to characterise which states are indistinguishable to a given actor α we need to know the lock state. With dynamic policies we need to take into account the fact that when the policy changes, memory locations that were previously considered secret could now be public, and vice versa. We handle this, as mentioned previously, by resetting the memory at each computation step. This brings us to our second attempt:

Attempt 2 (Memory-resetting bisimulation). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation on preconfigurations such that if

$$\langle \Sigma, M \rangle \sim_\alpha \langle \Delta, N \rangle \ \& \ \langle \Sigma, M, \mu \rangle \xrightarrow{p} \langle \Sigma', M', \mu' \rangle \ \& \ \mu =_\alpha^\Sigma \nu$$

then there exists Δ', N', ν' such that

either $\langle \Delta, N, \nu \rangle \to^* \langle \Delta', N', \nu' \rangle \ \& \ \mu' =_\alpha^\Sigma \nu' \ \& \ \langle \Sigma', M' \rangle \sim_\alpha \langle \Delta', N' \rangle$,

or $\langle \Delta, N, \nu \rangle \Uparrow$,

This definition is somewhat similar in spirit to non-disclosure [2]. For the moment we still view stores as containing memory locations only, and thus assume a semantics which avoids free variables altogether. This attempt takes into account that the effective secrecy status of memory locations can change during program execution, but this is not enough. In this rich language it is also possible to do the same for values that have been computed in the term, as shown by program (16) in section 3:

$$(!\ell_{\{\sigma \Rightarrow A\}}) := (\mathbf{open}\ \sigma; ())$$

In this example, we first compute a value on the left-hand side, which will be given the reading policy $\{\sigma \Rightarrow A\}$). From the point of view of A, this is a secret value, and could thus be different values in different runs of the program. However, when we compute the right-hand side, the value on the left-hand side is declassified, though it can still be different values in different runs, which means we could have an α-observable difference in the output of the two programs. This difference is fine though, since we explicitly changed the state to allow the flow to α, but we must still ensure that there are no other observable differences that do not arise from the newly opened lock. To check this, we want to continue the bisimulation but assume that we in fact had the same value on the left-hand side, and continue as before. This is the same thing that we do when "resetting" the memories, but we need to do the same thing for values in the term. In order to do this for values, we need a handle on those values, which is the motivation behind using the monadic CORE_{FL} language and the store-based operational semantics. Thus we arrive at our third attempt:

Attempt 3 (Store-resetting bisimulation). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation on preconfigurations such that if

$$\langle \Sigma, M\rangle \sim_\alpha \langle \Delta, N\rangle \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle$$
$$\& \; S =^{\Sigma}_{\alpha} T \; \textit{where} \; \mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$$

then there exists Δ', N', T' such that

either $\langle \Delta, N, T\rangle \rightarrow^{*} \langle \Delta', N', T'\rangle \;\&\; S' =^{\Sigma}_{\alpha} T' \;\&\; \langle \Sigma', M'\rangle \sim_\alpha \langle \Delta', N'\rangle$,

or $\langle \Delta, N, T\rangle \Uparrow$,

The main difference from the previous attempt is not in the formulation itself, but rather in the use of stores S and T ranging over both variables and locations instead of memories μ and ν, and the corresponding different formulation of the $=^{\Sigma}_{\alpha}$ relation.

The condition $\mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$ is just a hygiene condition that states that new variables introduced in S' are chosen to be distinct from the variables already present in T. Since the operational semantics is free to choose any locations this is not a restriction *per se*. In the "attempts" that follow we will tacitly elide this hygiene condition, but it is needed in all cases.

This definition of bisimulation is still not strong enough. It is not enough to require only that memories should be α-indistinguishable in the current state. A program such as $\ell_{\{\sigma \Rightarrow A\}} := !m_{\{\sigma' \Rightarrow A\}}$ is not secure (unless σ' is open), but with the above definition both locations would be considered unobservable by α, and hence no α-observable differences could be observed. The problem is that this insecure flow might only be revealed at some *future* time. To capture this problem we need to check the α-indistinguishability of the two memories in a state where σ is open but σ' is not. More generally, we must take into account all possible (more permissive) future lock states. Thus our fourth attempt at a definition is:

Attempt 4 (Future-sensitive bisimulation). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation on preconfigurations such that if

$$\langle \Sigma, M\rangle \sim_\alpha \langle \Delta, N\rangle \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \;\&\; \Theta \supseteq \Sigma \;\&\; S =^{\Theta}_{\alpha} T$$

then there exists Δ', N', T' such that

either $\langle \Delta, N, T\rangle \rightarrow^{*} \langle \Delta', N', T'\rangle \;\&\; S' =^{\Theta}_{\alpha} T' \;\&\; \langle \Sigma', M'\rangle \sim_\alpha \langle \Delta', N'\rangle$,

or $\langle \Delta, N, T\rangle \Uparrow$, Now we will rule out programs like the one above, but we're still not quite there. The final problem is that the definition is now actually too strong – it rules out some (well-typed) programs that should be considered secure, such as (somewhat simplified)

$$\mathbf{if}\ x_{\{\sigma \Rightarrow \alpha\},\tau}\ \mathbf{then}\ \ell_{\{\sigma \Rightarrow \alpha\}} := 0\ \mathbf{else}\ ().$$

The indirect flow from x to ℓ should be fine since they have the same policy, but since x is considered secret to α, the above definition requires us to show (after one computation step) that $\langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := 0 \rangle \sim_\alpha \langle \{\}, () \rangle$, which clearly does not hold $\forall \Theta \supseteq \Sigma$; in particular it will not hold when $\sigma \in \Theta$.

The problem is that opening σ means that the condition that we branched on becomes visible to α as well, but we've passed that point in the program and don't have access to the condition any more. To be sure we don't rule out programs such as these we must remember what branches we have taken, and in particular what possible future states that could make any of the branches visible to α, and make sure that we ignore leaks in those states. Thus our fifth and final attempt is formulated by parameterising the bisimulation relation by the set of locks that were closed at earlier branching points, to ensure that we are not future-sensitive to these locks.

Attempt 5 (Past-aware bisimulation). For any actor α let $\{\sim_\alpha^\Omega\}$ be the lock-set indexed family of relations defined to be the largest symmetric relations on preconfigurarions such that if

$$\langle \Sigma, M \rangle \sim_\alpha^\Omega \langle \Delta, N \rangle \ \&\ \langle \Sigma, M, S \rangle \xrightarrow{p} \langle \Sigma', M', S' \rangle \ \&\ \Theta \supseteq \Sigma \ \&\ S =_\alpha^\Theta T$$

then there exists Δ', N', T' such that

either $\langle \Delta, N, T \rangle \to^* \langle \Delta', N', T' \rangle \ \&\ S' =_\alpha^{\Theta \setminus \Omega} T' \ \&\ \langle \Sigma', M' \rangle \sim_\alpha^{\Omega'} \langle \Delta', N' \rangle$,

or $\langle \Delta, N, T \rangle \Uparrow$,

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

The difference to the previous attempt is that we allow stores to differ after computation as long as those differences are only visible in certain states — in particular those states in which a previous branching point would not have led to a branch at all.

This definition is less restrictive than the former in order to not rule out programs with indirect flows like the one presented above. There might be some concern as to whether this definition is now too weak, since we allow stores to differ in certain states. In particular, what of a direct leak observable only in such a state, like in the program $\mathbf{if}\ x_{\{\sigma \Rightarrow \alpha\},\tau}\ \mathbf{then}\ \ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'}\ \mathbf{else}\ ()$. This leak will indeed not be caught when we are working with $\Omega = \{\sigma\}$, so we have $\langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'} \rangle \sim_\alpha^{\{\sigma\}} \langle \{\}, () \rangle$. But recall that we still quantify over all $\Theta \supseteq \{\}$ when considering the conditional expression. Then for any $\Theta \supseteq \{\sigma\}$ the variable whose value we branch on will be considered public, and we will continue with the same branch in both cases. Also since the variable was public, there will be no states in which we allow future memories to differ in what α can see, and we must have $\langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'} \rangle \sim_\alpha^{\{\}} \langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'} \rangle$ which cannot hold.

This fifth attempt is our actual definition of a bisimulation.

4.6 Non-circular reasoning for bisimulation

As the sharp-eyed reader may well have noticed, our notion of a bisimulation implicitly constrains the stores used to be well-typed, i.e. if a location or variable is said to hold an integer value, it does indeed hold an integer value. This is not an unreasonable assumption to make in general, and since we reset the stores before each computation step and require the bisimulation properties to be fulfilled for *any* stores that are equal, it is an assumption that is crucial for this to work at all. It would be impossible for all but the simplest programs to be considered secure otherwise.

But unfortunately this assumption leads to a circular reasoning when we want to prove that our type system guarantees flow lock security. We allow the store to contain not only simple values like ints, but also functions with arbitrary terms as their bodies. In order to show that such a value is well-typed we need to use the full power of the type system.

Thus we end up in a situation where we want to show that well-typed terms are bisimilar to themselves, but the notion of bisimilarity already depends on the type system. To break this loop we can give a more general definition of a bisimulation where we parametrise the relation on some well-formedness predicate on stores:

Attempt 6 (Parametrised bisimulation). For any actor α, let $\sim_\alpha^\Omega$ be the lock-set indexed family of relations defined to be the largest symmetric relations on preconfigurations such that if

$$\langle\Sigma, M\rangle \sim_\alpha^\Omega \langle\Delta, N\rangle \;\&\; \mathcal{P}(S) \;\&\; \langle\Sigma, M, S\rangle \xrightarrow{p} \langle\Sigma', M', S'\rangle \;\&\; \mathcal{P}(S') \;\&\; \Theta \supseteq \Sigma \;\&\; S =_\alpha^\Theta T \;\&\; \mathcal{P}(T)$$

then there exits Δ', N', T' such that

either $\langle\Delta, N, T\rangle \rightarrow^* \langle\Delta', N', T'\rangle \;\&\; \mathcal{P}(T') \;\&\; S' =_\alpha^{\Theta\setminus\Omega} T' \;\&\; \langle\Sigma', M'\rangle \sim_\alpha^{\Omega'} \langle\Delta', N'\rangle$,

or $\langle\Delta, N, T\rangle \Uparrow$,

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

We would then prove that well-typed terms are bisimilar to themselves, assuming they start off in well-typed, well-formed stores, i.e. $\mathcal{P}(S) \;=\; \vdash S$. Subject reduction gives us that the typeability of stores is retained, so this would not complicate the proofs the least.

This is the complete, most general definition of a bisimulation. The previous definition, which we actually use, can be seen as an instantiation of this definition for the proper $\mathcal{P}$, and we will use that one for simplicity.

4.7 Well-typed Programs are Flow-Lock Secure

We now want to prove that all programs typeable with our type system are indeed secure. The proof follows a similar structure to the corresponding proof from Almeida Matos and Boudol [2].

The basic approach is to utilise the coinductive nature of the bisimulation definition. We show that for well-typed closed M, $\langle\emptyset, M\rangle \sim_\alpha \langle\emptyset, M\rangle$ by construction of a candidate relation R_α^α, that in particular contains the pair $(\langle\emptyset, M\rangle, \langle\emptyset, M\rangle)$, and which can be shown to be an α-bisimulation. This gives us that $(\langle\emptyset, M\rangle, \langle\emptyset, M\rangle) \in R_\alpha^\alpha \subseteq \sim_\alpha$.

To be able to define the candidate relation R_α^Ω we need the notion of programs that are *high* with respect to some actor α. We say that a program is α-Ω-high if it does not modify any locations that α could see while all the locks in Ω remains closed. However, this operational notion of being high is a bit akward to work with, so instead we use a stronger, syntactic notion stating that a program is *syntactically* α-Ω-high if it does not *write* to any locations that α could see while the locks in Ω remain closed.

Definition 5 (Syntactically α-Ω-high programs: H_α^Ω). Let H_α^Ω be the set of all terms M such that $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ and $\alpha \not\preccurlyeq^\Omega w$.

Now we can define our candidate relation:

Definition 6 (Candidate relation R_α^Ω). Let R_α^Ω be a symmetric relation on well-typed preconfigurations, inductively defined as follows:

$$1\,\frac{}{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, M\rangle} \qquad 2\,\frac{M, N \in H_\alpha^\Omega}{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle}$$

$$3\,\frac{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Sigma, N\rangle \quad \alpha \not\preccurlyeq^\Omega p}{\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = M\ \mathbf{in}\ M']\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N\ \mathbf{in}\ M']\rangle}$$

where $\mathbb{E}[\cdot]$ are the evaluation contexts for $\textsc{core}_{FL}$, given by

$$\mathbb{E}[\cdot] ::= [\cdot] \mid \mathbf{bind}\ \boldsymbol{x} = \mathbb{E}[\cdot]\ \mathbf{in}\ M$$

In words, two well-typed preconfigurations are related by R_α^Ω if the programs in them are either equal, both are high, or they are two sub-programs related by R_α^Ω inside nested (equal) bind constructs, where the results of those sub-computations are secret to α. The lock-state components constrain what preconfigurations are in the relation only through the typeability requirement.

The final piece of the puzzle is now to show that this candidate relation is indeed a bisimulation.

Lemma 4 ($\bigcup\limits_\Omega R_\alpha^\Omega$ is a bisimulation). *If* $\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle$ *and* $\vdash S$ *and*

$$\langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \ \&\ \Theta \supseteq \Sigma \ \&\ S =_\alpha^\Theta T \ \&\ \vdash T$$

then $\vdash S'$*, and there exists* Δ', N', T' *such that*

either $\langle \Delta, N, T\rangle \rightarrow^* \langle \Delta', N', T'\rangle \ \&\ \vdash T' \ \&\ S' =_\alpha^{\Theta\setminus\Omega} T' \ \&\ \langle \Sigma', M'\rangle R_\alpha^{\Omega'} \langle \Delta', N'\rangle$,

or $\langle \Delta, N, T\rangle \Uparrow$,

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

We prove this by induction on the size of the typing derivation of $\langle \Sigma, M\rangle$. The details of this proof can be found in appendix A.2.

$$\frac{p = \bigsqcup_{\ell \in E} \mathrm{level}(\ell).}{\vdash_{NI} E : p} \qquad \frac{\vdash_{NI} E : q \qquad p \sqcup q \sqsubseteq \mathrm{level}(\ell)}{p \vdash_{NI} u := \ E} \qquad \frac{p \vdash_{NI} C_1 \quad p \vdash_{NI} C_2}{p \vdash_{NI} C_1; C_2}$$

$$\frac{\vdash_{NI} E : q \quad p \sqcup q \vdash_{NI} C_i \quad i = 1, 2}{p \vdash_{NI} \textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2} \qquad \frac{\vdash_{NI} E : q \quad p \sqcup q \vdash_{NI} C}{p \vdash_{NI} \textbf{while } (E)\ C}$$

Figure 6: Standard Noninterference Type System

5 Relating to Other Systems and Idioms

Standard Noninterference As a first example of the expressiveness of our system, consider a standard termination insensitive noninterference property for a lattice-based security model in the standard Denning style [6].

In this setting we have a lattice of security levels $\langle \mathcal{L}, \sqsubseteq, \sqcup \rangle$, and a policy $\mathrm{level} : \mathsf{Loc} \to \mathcal{L}$ that fixes the intended security level of the storage locations in the program (and of variables). Given such a policy we can define noninterference. To do this let us first assume that all policies are made up of sets of clauses of the form $\{\} \Rightarrow \alpha$, and that programs do not use lock open/close operations. Furthermore, for simplicity we consider programs of unit type which do not perform any allocation of new references (locations). In what follows let metavariables P and Q range over such programs.

Definition 7 (Noninterference). Given two stores S and T, and a level $k \in \mathcal{L}$, define S and T to be *location indistinguishable at level* k, written $S =_k T$, iff the location domains of S and T are the same, and for all $\ell \in \mathrm{dom}(S)$ such that $\mathrm{level}(\ell) \sqsubseteq k$ we have $S(\ell) = T(\ell)$.

Then we say that variable-free program P is *noninterfering* if for all k, whenever $\langle P, S \rangle \to^* \langle (), S' \rangle$, and $\langle P, T \rangle \to^* \langle (), T' \rangle$, then $S =_k T$ implies $S' =_k T'$.

To represent a lattice policy we do not need any locks; we represent the reading level of a variable by the set of levels at which it may be read. Thus the policy for a storage location ℓ is the upwards closure of its lattice level, written $\uparrow\mathrm{level}(\ell)$, where $\uparrow k = \{\{\} \Rightarrow j \mid j \sqsupseteq k\}$.

In what follows we will *implicitly* identify lattice levels k with the corresponding flow lock policy $\uparrow k$

Given this, we have the following:

Theorem 1. *If P is flow lock secure then P is noninterfering.*

The details of the proof are given in Appendix A.3.

But it is perhaps not too surprising that our security specification is stronger than standard noninterference. A reasonable concern might be that the definition, or indeed the type system, is too strong to be useful. Here we show that despite being stronger, we are still able to type just as much as "typical" systems for regular noninterference.

Figure 6 presents a simple type system for a while language which can be seen as a straightforward reformulation of the typing system presented by Volpano, Irvine and Smith [21].

Define the following translation $\lceil\cdot\rceil$ from terms in the while language to λ_{FL}:

$$\begin{aligned}
\lceil \mathbf{while}\ (E)\ C \rceil &= \mathbf{rec}\ x.\mathbf{if}\ \lceil E \rceil\ \mathbf{then}\ \lceil C \rceil; x\ \mathbf{else}\ () \\
\lceil \mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \rceil &= \mathbf{if}\ \lceil E \rceil\ \mathbf{then}\ \lceil C_1 \rceil\ \mathbf{else}\ \lceil C_2 \rceil \\
\lceil C_1; C_2 \rceil &= \lceil C_1 \rceil; \lceil C_2 \rceil \\
\lceil \ell :=\ E \rceil &= \ell_p := \lceil E \rceil \quad \text{where } p = \uparrow\text{level}(\ell) \\
\lceil E \rceil &= E' \quad \text{where } E' \text{ is the result of replacing} \\
&\qquad\qquad \text{each location } \ell \text{ in } E \text{ with } \ell_{\uparrow\text{level}(\ell)}.
\end{aligned}$$

To make our formulations easier, let us restrict the language of expressions to booleans (so we do not have to consider typing issues). Now we can state that whenever something is typeable in the simple noninterference system, a corresponding derivation holds for the flow locks system:

Theorem 2. *Let Γ_0 be the type environment that maps every storage location to bool. Then*

1. *If $\vdash_{NI} E : k$ then $\Gamma_0; \emptyset \vdash \lceil E \rceil : bool, (r, \top) \Rightarrow \emptyset$ where $r = \uparrow k$*
2. *If $pc \vdash_{NI} C$ then $\Gamma_0; \emptyset \vdash \lceil C \rceil : unit, (r, w) \Rightarrow \emptyset$ where $w \subseteq \uparrow pc$*

We also expect that a similar theorem holds for some suitable termination-insensitive version of DCC [1], although we have not attempted to show this formally.

Simple Declassification We can encode a simple declassification mechanism in the same Denning-style setting as used in the previous example. The needed extra step is to extend all policies with clauses to allow declassification. For each level j not in the policy already, we introduce a flow lock σ_j representing a declassification to that level. The new policies then look like

$$\{k \mid k \sqsupseteq \text{level}(\ell)\} \cup \{\sigma_j \Rightarrow k \mid j \not\sqsupseteq \text{level}(\ell), k \sqsupseteq j\}$$

We can now define a declassification operator to level j as

$$declassify_j \equiv (\lambda v.\mathbf{let}\ x = (\mathbf{open}\ \sigma_j; v)\ \mathbf{in}\ (\mathbf{close}\ \sigma_j; x))$$

It is easy to verify from the type system that the only effect of applying this function to some value is that the value will then be readable also at level j, as was our intention.

Lexically Scoped Flows In the setting of a multilevel security model, Almeida Matos and Boudol describe how to build a system with lexically scoped dynamic flow policies [2]. They start from a λ-calculus with recursion and references like we do, and introduce a construct *"flow $\alpha \prec \beta$ in M"* that extends the current global flow policy to also allow flows from level α to β in the scope of M. These flows are transitive, so if the current policy already allows flows from say β to γ, flows from α to γ would also be allowed in M.

Modelling scoped flows using flow locks is easy, but the global nature of policies in Almeida Matos and Boudol's system, as opposed to our local policies on memory locations, needs special treatment. We introduce a lock $\sigma_{\alpha\prec\beta}$ for each pair of levels α and β that data could flow between. Each policy on some data must record the fact that a future flow declaration could allow that data to flow to many new locations due to the transitive nature of flows. Thus if a location in Almeida Matos and Boudol's system would have level A, we could represent that as

$$A \cup \left\{\sigma_{\alpha\prec\beta_0}, \sigma_{\beta_0\prec\beta_1}, \ldots, \sigma_{\beta_{k-1}\prec\beta_k} \Rightarrow \beta_k \mid \alpha \in A, \beta_i \notin A\right\}$$

where the $\notin$ is taken with respect to some universal set of levels. In effect, each location records all possible future transitive flows from it. We then derive our representation of the "flow" construct that opens a lock in the scope of some subprogram:

$$\mathbf{flow}\ \sigma\ \mathbf{in}\ M \equiv \mathbf{let}\ x = (\mathbf{open}\ \sigma; M)\ \mathbf{in}\ (\mathbf{close}\ \sigma; x)$$

Almeida Matos and Boudol also include parallel execution in their system, and as a consequence make their type system and semantic security definition, called *non-disclosure*, sensitive to possible non-termination. Our system has no parallel execution so we cannot model their full system, only the sequential subset.

Intransitive Noninterference Flow locks represent a lower level abstraction than lattice-based information flow models in the sense that the lattice ordering is not "built in" but must be represented explicitly. One advantage of such a lower level view is that it can also represent *intransitive noninterference* policies [15, 14] — i.e. ones in which the flow relation is intentionally not transitive. Since intransitive policies are the default case for flow locks, it is straightforward to represent simple language-based intransitive policies such as the one described by Mantel and Sands [8].

Noninterference Until Declassification Chong and Myers' [5] introduce a class of temporal declassification policies. This is achieved by annotating variables with types of the form $k_0 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_n}{\rightsquigarrow} \underline{k_n}$, which intuitively means that a variable with such an annotation may be successively declassified to the levels $k_1, \ldots, k_n$, and that the conditions $c_1, \ldots, c_n$ will hold at the execution of the corresponding declassification points. The exact nature of the conditions are left unspecified, and it is assumed in the type system that these conditions are verified at certain key program points by some external tool.

We can achieve a similar effect fairly naturally using flow locks, where we would use a distinct lock C_i for each condition c_i. One should then insert **open** C_i constructs in the program at points where the intended declassification takes place, and verify (with an external tool) that the corresponding condition c_i does indeed hold at these points, and that lock C_{i-1} has been opened (we assume that locks are never closed in this encoding). The policy above could then be represented as

$$\{k_0; \{C_1\} \Rightarrow k_1; \cdots; \{C_1, \ldots, C_n\} \Rightarrow k_n\}.$$

Robust Declassification Information flow may be used to verify integrity properties, to ensure that untrusted (low integrity) data does not influence the values of trusted (high integrity) data. Since flow lock policies are neutral with respect to whether we are dealing with confidentiality or integrity properties it is no problem to add such integrity policies to data, and we can easily have clauses for integrity and confidentiality in the same policy. The interesting case, however, is the interaction between confidentiality and integrity in the presence of dynamic policies.

Zdancewic and Myers [22] introduced the concept of *robust declassification* to characterise the property that an attacker (who controls low integrity data) cannot influence what is declassified. This guarantees that the attacker cannot manipulate the amount of information which is released through declassification.

In the setting of flow lock policies, "declassification" can be thought of as the process of opening locks, since whenever a lock is opened more flows are enabled. Thus we can interpret robust declassification as the question of whether low integrity data can influence the decision to open locks. [5]

One possible way of enforcing robust declassification using flow locks is to observe the following: since we cannot perform any computation with locks, the only way that an open operation can be influenced by low integrity data is via indirect information flow from low integrity data. Suppose that our policies use an indexed set of locks $\sigma_i, i \in I$ to control confidentiality. These are unguarded (i.e. we ignore *endorsement*). Let us assume that in addition to the actors of the system we have the pseudo-actor *trusted* used to track integrity information, just as we did in Section 2.

In order to prevent indirect flow from low integrity data to the opening of locks, we will log each use of an open operation by writing to a variable log. An obvious way to enforce this is to define a "robust" version of open:

$$\mathbf{ropen}\ \sigma_i \equiv \mathbf{open}\ \sigma_i; log := \ i$$

Now we give log the policy $\{trusted\}$. This ensures that the assignment is always safe from a confidentiality perspective (since normal actors can never read it anyway), and that the open operation can never have taken place in a low integrity context (since otherwise the assignment would cause information to flow from untrusted to trusted data). Finally, to additionally prevent the declassification of low integrity data we can syntactically enforce that lock-guarded policies are only used on high integrity data.

The Decentralized Label Model In the Decentralized Label Model (DLM) [10, 11, 12], data is said to be *owned* by a set of principals. These principals may allow other principals to read the data, and the effective reader set is those principals that all owners agree may read the data. Allowing a new reader roughly corresponds to declassification, and we can model it similarly. The DLM also defines a global principal hierarchy, where one principal may allow another principal to *act for* it, which means it may read all the same things. This is very similar in spirit to introducing a new flow in the system by Almeida Matos and Boudol, including transitivity, and we can model it in the same way. Apart from clauses for declassification and hierarchic flows, the policies must also include clauses for the combination of the two, e.g. A can read the data if B owns it, has declassified it for C to read it, and A acts for C.

A common extension of the DLM [22, 20, 19] deals with integrity and trust. The interesting part for us is the integration with the principal hierarchy, where if A trusts some data and A acts for B, then B also trusts that data. This can be modelled as the reverse of the normal clauses for transitive flows, and the clauses will be very similar to those for forward flows.

The complete general policy for a DLM variable encoded with flow locks would be fairly large and awkward, so we do not show it here.

[5] If we also take the view from [13], then we extend this concept with the requirement that we should not be able to declassify low integrity data

Other Related Work The JFlow language [9], as well as several recent papers [19, 23, 7], supports runtime mechanisms to enforce security in situations where this cannot be determined statically, e.g. permissions on a file that cannot be known at compile time. Our flow locks is a static, compile-time mechanism only, and thus cannot handle these issues.

Banerjee and Naumann [4] describe a combination of stack-based access control and information flow types to allow the static checking of policies such as "the method returns a result at level L unless the caller has permission p". It may be possible to encode these kinds of policies in a straightforward way using flow locks, but this remains a topic for future work.

6 Conclusions and Future Work

Flow locks are a very simple mechanism that generalises many existing systems and idioms for dynamic information flow policies. We have only just started looking at flow locks however, and much remains to be done.

To really establish flow locks as a core calculus, we need to show more formally how to embed other systems and idioms, and prove that our semantic condition is sufficiently strong compared to the semantic conditions of these other systems. It would also be worthwhile to look at extensions of our core system, in order to handle systems that we definitely cannot model at this point. Examples of such systems include the parallel execution of Almeida Matos and Boudol [2], and also systems that use various runtime mechanisms [19, 23, 7].

Furthermore, we would need to investigate how to implement the flow locks system as a programming language, and to determine what kinds of inference would be needed for policies and locks. Also, flow locks are fairly low-level in nature, being a raw mechanism for controlling data flows in a program. As such it is nontrivial to write and maintain correct flow lock programs. It would therefore be useful to look at what higher-level abstractions and design patterns that could be used together with flow locks. There exists some work specifically targeting the question of patterns, for instance the *seal* pattern by Askarov and Sabelfeld [3].

Acknowledgements Thanks to Ulf Norell and our colleagues in the ProSec group for helpful comments, and to the anonymous referees (ESOP'06) for numerous helpful comments and suggestions. This work was partly supported by the Swedish research agencies SSF, VR and Vinnova, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, Jan. 1999.

[2] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

[3] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, 2005.

[4] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.

[5] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, Oct. 2004.

[6] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[7] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proc. Foundations of Computer Security Workshop*, 2005.

[8] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, Nov. 2004.

[9] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[10] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, Oct. 1997.

[11] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

[12] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[13] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[14] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.

[15] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

[16] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[17] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[18] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, 2005.

[19] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *Proc. Symposium on Security and Privacy*, 2004.

[20] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 279–294. Springer-Verlag, Apr. 2005.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[22] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

[23] L. Zheng and A. Myers. Dynamic security labels and noninterference. In *Proc. Workshop on Formal Aspects in Security and Trust*, 2004.

A Appendix

A.1 Proofs that the type system guarantees semantic soundness

Lemma 1 (Progress). If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ then either

- $M \in \text{Val}$, or
- for all S such that $\text{dom}(S) \supseteq \text{fv}(M) \cup \text{loc}(M)$ and $\vdash S$ then $\exists \Sigma', M', S'. \langle \Sigma, M, S\rangle \rightarrow \langle \Sigma', M', S'\rangle$.

Proof. In $CORE_{FL}$ the syntax restricts the terms of the language to be in a reductive form, except the *bind* construct. This is thus the only case for which the lemma does not trivially hold. By induction on the size of the typing derivation for M we get for the *bind* case from the induction hypothesis that it holds for the bound expression, and thus it holds for M. □

Lemma 2 (Preservation). If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ and $\vdash S$ and $\text{dom}(S) \supseteq \text{fv}(M) \cup \text{loc}(M)$ and $\langle \Sigma, M, S\rangle \rightarrow \langle \Sigma', M', S'\rangle$ then $\vdash S'$ and $\Sigma' \vdash M' : \tau, (r', w') \Rightarrow \Delta$ where $r' \preceq r$ and $w \preceq w'$.

Proof. We prove this by induction on the typing derivation for M, and by cases according to the structure of M.

Case: $M \in$ Val. The statement is vacuously true.

Case: $M = x_{p,\tau}$. The reduction has the form $\langle \Sigma, x_{p,\tau}, S\rangle \rightarrow \langle \Sigma, S(x_{p,\tau}), S\rangle$, and the typing derivation is of the form $\Sigma \vdash x_{p,\tau} : \tau, (p(\Sigma), \top) \Rightarrow \Sigma$. Since $\vdash S$ this means that $\Sigma \vdash S(x_{p,\tau}) : \tau, (\bot, \top) \Rightarrow \Sigma$. We have $\bot \preceq p(\Sigma)$ and $\top \preceq \top$ as required.

Case: $M = \mathbf{ref}_{p'}\ x_{p,\tau}$. The reduction has the form

$$\langle \Sigma, \mathbf{ref}_{p'}\ x_{p,\tau}, S\rangle \rightarrow \langle \Sigma, \ell_{p',\tau}, S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]\rangle$$

and the typing derivation is of the form

$$\frac{p \preceq p'}{\Sigma \vdash \mathbf{ref}_{p'}\ x_{p,\tau} : \mathbf{ref}_{p'}\ \tau, (\bot, p') \Rightarrow \Sigma}.$$

Since $\vdash S$ we have that $\Sigma \vdash S(x_{p,\tau}) : \tau, (\bot, \top) \Rightarrow \Sigma$, so we have $\vdash S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]$ and $\Sigma \vdash \ell_{p',\tau} : \mathbf{ref}_{p'}\ \tau, (\bot, \top) \Rightarrow \Sigma$. We have $\bot \preceq \bot$ and $p' \preceq \top$ as required.

Case: $M = !x_{p,\mathbf{ref}_{p'}\ \tau}$. The reduction has the form

$$\langle \Sigma, !x_{p,\mathbf{ref}_{p'}\ \tau}, S\rangle \rightarrow \langle \Sigma, S(S(x_{p,\tau})), S\rangle$$

and the typing derivation is of the form $\Sigma \vdash\ !x_{p,\mathbf{ref}_{p'}\ \tau} : \tau, (p(\Sigma) \sqcup p'(\Sigma), \top) \Rightarrow \Sigma$. Since $\vdash S$ this means that $\Sigma \vdash S(x_{p,\tau}) : \mathbf{ref}_{p'}\ \tau, (\bot, \top) \Rightarrow \Sigma$, and thus that $\Sigma \vdash S(S(x_{p,\tau})) : \tau, (\bot, \top) \Rightarrow \Sigma$. We have $\bot \preceq p(\Sigma) \sqcup p'(\Sigma)$ and $\top \preceq \top$ as required.

$\boxed{\textbf{Case: } M = x_{p,\mathbf{ref}_{p''}\ \tau} := y_{p',\tau'}.}$ The reduction has the form

$$\langle \Sigma, x_{p,\mathbf{ref}_{p''}\ \tau} := y_{p',\tau'}, S \rangle \rightarrow \langle \Sigma, (), S[S(x_{p,\mathbf{ref}_{p''}\ \tau} \mapsto S(y_{p',\tau'})] \rangle$$

and the typing derivation has the form

$$\frac{p(\Sigma) \sqcup p'(\Sigma) \preceq p''}{\Sigma \vdash x_{p,\mathbf{ref}_{p''}\ \tau} := y_{p',\tau'} : \mathit{unit}, (\bot, p'') \Rightarrow \Sigma}.$$

Since $\vdash S$ this means that $\Sigma \vdash S(x_{p,ref_{p''}\ \tau}) : \mathbf{ref}_{p''}\ \tau, (\bot, \top) \Rightarrow \Sigma$, and that $\Sigma \vdash S(y_{p',\tau'}) : \tau, (\bot, \top) \Rightarrow \Sigma$, and thus we have that $\vdash S[S(x_{p,\mathbf{ref}_{p''}\ \tau} \mapsto S(y_{p',\tau'})]$. We have $\Sigma \vdash () : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma$, and $\bot \preceq \bot$ and $p'' \preceq \top$ as required.

$\boxed{\textbf{Case: } M = \mathbf{if}\ x_{p,bool}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1.}$ The reduction has the form

$$\langle \Sigma, \mathbf{if}\ x_{p,bool}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, S \rangle \rightarrow \langle \Sigma, M_i, S \rangle$$

and the typing derivation is of the form

$$\frac{\Sigma \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Delta \quad p(\Sigma) \preceq w_0 \sqcap w_1}{\Sigma \vdash \mathbf{if}\ x_{p,\tau}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1 : \tau, (p(\Sigma) \sqcup r_0 \sqcup r_1, w_0 \sqcap w_1) \Rightarrow \Delta}.$$

We have $r_i \preceq p(\Sigma) \sqcup r_0 \sqcup r_1$ and $w_0 \sqcap w_1 \preceq w_i$ as required.

$\boxed{\textbf{Case: } M = x_{p,\tau_f}\ y_{p',\tau'}\ \textbf{where}\ \tau_f = (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau.}$ The reduction has the form

$$\langle \Sigma, x_{p,\tau_f}\ y_{p',\tau'}, S \rangle \rightarrow \langle \Sigma, M, S[z_{p',\tau'} \mapsto S(y_{p',\tau'})) \rangle,$$

where $S(x_{p,\tau_f}) = \lambda z_{p',\tau'}.M$. The typing derivation is of the form

$$\frac{p(\Sigma) \preceq w_f}{\Sigma \vdash x_{p,\tau_f}\ y_{p',\tau'} : \tau, (p(\Sigma) \sqcup r_f, w_f) \Rightarrow \Sigma'}.$$

Since $\vdash S$ this means that

$$\frac{\Sigma \vdash M : \tau, (r_f, w_f) \Rightarrow \Sigma'}{\Sigma \vdash \lambda z_{p',\tau'}.M : (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau, (\bot, \top) \Rightarrow \Sigma}$$

and that $\Sigma \vdash S(y_{p',\tau'}) : \tau', (\bot, \top) \Rightarrow \Sigma$, so we can show that $\vdash S[z_{p',\tau'} \mapsto S(y_{p',\tau'})]$. We have $r_f \preceq p(\Sigma) \sqcup r_f$ and $w_f \preceq w_f$ as required.

$\boxed{\textbf{Case: } M = \mathbf{open}\ \sigma.}$ The reduction has the form $\langle \Sigma, \mathbf{open}\ \sigma, S \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, (), S \rangle$. By typing we know $\Sigma \vdash \mathbf{open}\ \sigma : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma \cup \{\sigma\}$, and we can show $\Sigma \cup \{\sigma\} \vdash () : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma \cup \{\sigma\}$. We have $\bot \preceq \bot$ and $\top \preceq \top$ as required.

$\boxed{\textbf{Case: } M = \mathbf{close}\ \sigma.}$ Similar to the previous case.

$\boxed{\textbf{Case: } M = \textbf{bind } x_{p,\tau} = M' \textbf{ in } N.}$ Here we have two cases: either $M' \in \text{Val}$, or we can do a reduction in M'.

$\boxed{\textbf{Subcase: } M' = v.}$ The reduction and type derivations respectively have the form

$$\langle \Sigma, \textbf{bind } x_{p,\tau} = v \textbf{ in } N, S \rangle \to \langle \Sigma, N, S[x_{p,\tau} \mapsto v] \rangle$$

$$\frac{\Sigma \vdash v : \tau, (\bot, \top) \Rightarrow \Sigma \quad \Sigma \vdash N : \tau', (r, w) \Rightarrow \Delta}{\Sigma \vdash \textbf{bind } x_{p,\tau} = v \textbf{ in } N : \tau', (r, w) \Rightarrow \Delta}.$$

We thus have $\vdash S[x_{p,\tau} \mapsto v]$, and $r \preceq r$ and $w \preceq w$ as required.

$\boxed{\textbf{Subcase: } M' \notin \text{Val.}}$ The type derivation for M must have the form

$$\frac{\Sigma \vdash M' : \tau, (r', w') \Rightarrow \Sigma'' \quad \Sigma'' \vdash N : \tau', (r_N, w_N) \Rightarrow \Delta \quad r'(\Sigma'') \preceq p}{\Sigma \vdash \textbf{bind } x_{p,\tau} = M' \textbf{ in } N : \tau', (r_N, w' \sqcap w_N) \Rightarrow \Delta}.$$

and by progress that we can reduce M'. We can thus perform the reduction $\langle \Sigma, M', S \rangle \to \langle \Sigma', M'', S' \rangle$, and by the induction hypothesis we know $\Sigma' \vdash M'' : \tau, (r'', w'') \Rightarrow \Sigma''$ where $r'' \preceq r'$, $w' \preceq w''$ and $\vdash S'$. We can thus show that

$$\frac{\Sigma' \vdash M'' : \tau, (r'', w'') \Rightarrow \Sigma'' \quad \Sigma'' \vdash N : \tau', (r_N, w_N) \Rightarrow \Delta \quad r''(\Sigma'') \preceq p}{\Sigma \vdash \textbf{bind } x_{p,\tau} = M'' \textbf{ in } N : \tau', (r_N, w'' \sqcap w_N) \Rightarrow \Delta},$$

and we have $r_N \preceq r_N$ and $w' \sqcap w_N \preceq w'' \sqcap w_N$ as required.

□

A.2 Proof that Well-typed Programs are Flow-Lock Secure

In this section we prove our claim that all programs typeable with our type system are indeed secure.

The basic approach is to utilise the coinductive nature of the bisimulation definition. We show that for well-typed closed M, $\langle \emptyset, M \rangle \sim_\alpha \langle \emptyset, M \rangle$ by construction of a candidate relation R^Ω_α, that in particular contains the pair $(\langle \emptyset, M \rangle, \langle \emptyset, M \rangle)$, and which can be shown to be an α-bisimulation. This gives us that $(\langle \emptyset, M \rangle, \langle \emptyset, M \rangle) \in R^\emptyset_\alpha \subseteq \sim_\alpha$.

A.2.1 The candidate relation R^Ω_α

To be able to define the candidate relation R^Ω_α we need the notion of programs that are *high* with respect to some actor α. (A similar concept is introduced by Almeida Matos and Boudol [2]). We say that a program is α-Ω-high if it does not modify any locations that α could see while all the locks in Ω remain closed. However, this operational notion of being high is a bit awkward to work with, so instead we use a stronger, syntactic notion stating that a program is *syntactically* α-Ω-high if it does not *write* to any locations that α could see while the locks in Ω remain closed.

Definition 5 (Syntactically α-Ω-high programs: H^Ω_α). Let H^Ω_α be the set of all terms M such that $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ and $\alpha \not\preceq^\Omega w$.

Now we can define our candidate relation as follows:

Definition 6 (Candidate relation R^{Ω}_{α}). Let R^{Ω}_{α} be a symmetric relation on well-typed preconfigurations, inductively defined as follows:

$$1\,\frac{}{\langle\Sigma, M\rangle R^{\Omega}_{\alpha} \langle\Delta, M\rangle} \qquad 2\,\frac{M, N \in H^{\Omega}_{\alpha}}{\langle\Sigma, M\rangle R^{\Omega}_{\alpha} \langle\Delta, N\rangle}$$

$$3\,\frac{\langle\Sigma, M\rangle R^{\Omega}_{\alpha} \langle\Sigma, N\rangle \quad \alpha \not\preceq^{\Omega} p}{\langle\Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = M\ \mathbf{in}\ M']\rangle R^{\Omega}_{\alpha} \langle\Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N\ \mathbf{in}\ M']\rangle}$$

where $\mathbb{E}[\cdot]$ are the evaluation contexts for CORE$_{FL}$, given by

$$\mathbb{E}[\cdot] ::= [\cdot] \mid \mathbf{bind}\ \boldsymbol{x} = \mathbb{E}[\cdot]\ \mathbf{in}\ M$$

We can identify a useful property of this set, namely that if two preconfigurations are related and one of the programs is high, then so is the other.

Lemma 5 (R^{Ω}_{α} relates high terms to other high terms). *If $\langle\Sigma, M\rangle R^{\Omega}_{\alpha} \langle\Delta, N\rangle$ and $M \in H^{\Omega}_{\alpha}$, then $N \in H^{\Omega}_{\alpha}$.*

Proof. By induction on the size of the typing derivation of M.

If $\langle\Sigma, M\rangle R^{\Omega}_{\alpha} \langle\Delta, N\rangle$ by rule 1, then $M = N$ and we have $N \in H^{\Omega}_{\alpha}$.

If $\langle\Sigma, M\rangle R^{\Omega}_{\alpha} \langle\Delta, N\rangle$ by rule 2, then $N \in H^{\Omega}_{\alpha}$ by construction.

If $\langle\Sigma, M\rangle R^{\Omega}_{\alpha} \langle\Delta, N\rangle$ by rule 3, then we have

$$M = \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = M_0\ \mathbf{in}\ M_1] \text{ and } N = \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_0\ \mathbf{in}\ M_1],$$

where $\langle\Sigma, M_0\rangle R^{\Omega}_{\alpha} \langle\Delta, N_0\rangle$.

By typing we have $\dfrac{\Sigma \vdash M_0 : \tau, (r_0, w_0) \Rightarrow \Sigma' \quad \Sigma' \vdash M_1 : \tau', (r_1, w_1) \Rightarrow \Sigma''}{\Sigma \vdash \mathbf{bind}\ x_{p,\tau} = M_0\ \mathbf{in}\ M_1 : \tau', (r_1, w_0 \sqcap w_1) \Rightarrow \Sigma''}$.

Since $M \in H^{\Omega}_{\alpha}$ we have $\alpha \not\preceq^{\Omega} w_0 \sqcap w_1$, which means that $M_0, M_1 \in H^{\Omega}_{\alpha}$. We apply the induction hypothesis on M_0 to get that $N_0 \in H^{\Omega}_{\alpha}$, and thus that $\mathbf{bind}\ x_{p,\tau} = N_0\ \mathbf{in}\ M_1 \in H^{\Omega}_{\alpha}$. Continuing the same argument for all binds in $\mathbb{E}[]$, we get that $N \in H^{\Omega}_{\alpha}$ as required. □

A.2.2 Proof that R^{Ω}_{α} is a bisimulation

Now that we have our candidate relation, the final step is to prove that it is indeed a bisimulation. In order to do this, we first need to state a number of helper lemmas.

We begin by proving that syntactically high programs are also operationally high, i.e. that they never produce any α-observable changes to the store. We do this in three separate steps. First we prove that syntactically high terms reduce to syntactically high terms. Second, we prove that reducing a syntactically high term will not result in any α-observable changes to the store. Finally we put these two together to form a notion of uninterrupted high computation.

Lemma 6 (H^{Ω}_{α} is closed under reduction). *If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ and $\alpha \not\preceq^{\Omega} w$ and $\vdash S$ and $\langle\Sigma, M, S\rangle \rightarrow \langle\Sigma', M', S'\rangle$ then $\vdash S'$ and $\Sigma' \vdash M' : \tau, (r', w') \Rightarrow \Delta$ and $\alpha \not\preceq^{\Omega} w'$.*

Proof. Preservation gives us $w \preceq w'$, so if $\alpha \not\preceq^{\Omega} w$ then $\alpha \not\preceq^{\Omega} w'$. □

Lemma 7 (H_{α}^{Ω} is α-Ω-high).
If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ and $\alpha \not\preceq^{\Omega} w$ and $\vdash S$ and $\langle \Sigma, M, S \rangle \rightarrow \langle \Sigma', M', S' \rangle$ then $\forall \Theta. S =_{\alpha}^{\Theta \setminus \Omega} S'$.

Proof. By induction on the size of the typing derivation. For terms that do not update or create a location in the store when reduced, the above is trivially true. The remaining cases are reference creation, assignment and the recursive case of bind:

$\boxed{\textbf{Case: } M = \mathbf{ref}_{p'}\ x_{p,\tau}.}$ In this case the reduction and typing derivation are of the following form:

$$\langle \Sigma, \mathbf{ref}_{p'}\ x_{p,\tau}, S \rangle \rightarrow \langle \Sigma, \ell_{p',\tau}, S[\ell_{p',\tau} \mapsto S(x_{p,\tau})] \rangle$$

$$\frac{p \preceq p'}{\Sigma \vdash \mathbf{ref}_{p'}\ x_{p,\tau} : \mathit{ref}_{p'}\ \tau, (\bot, p') \Rightarrow \Delta}$$

and we know $\alpha \not\preceq^{\Omega} p'$. Thus the newly created location is secret to α, and we have $\forall \Theta. S =_{\alpha}^{\Theta \setminus \Omega} S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]$.

$\boxed{\textbf{Case: } M = x_{p,\mathit{ref}_{p''}\ \tau} := y_{p',\tau}.}$ The reduction and typing derivation have the form

$$\langle \Sigma, x_{p,\mathit{ref}_{p''}\ \tau} := y_{p',\tau}, S \rangle \rightarrow \langle \Sigma, (), S[S(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto S(y_{p',\tau})] \rangle$$

$$\frac{p(\Sigma) \sqcup p'(\Sigma) \preceq p''}{\Sigma \vdash x_{p,\mathit{ref}_{p''}\ \tau} := y_{p',\tau} : \mathit{unit}, (\bot, p'') \Rightarrow \Sigma}$$

and we know $\alpha \not\preceq^{\Omega} p''$. Since $\vdash S$ we know that $\Sigma \vdash S(x_{p,\mathit{ref}_{p''}\ \tau}) : \mathit{ref}_{p''}\ \tau, (\bot, \top) \Rightarrow \Sigma$ and thus $S =_{\alpha}^{\Theta \setminus \Omega} S[S(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto S(y_{p',\tau})]$.

$\boxed{\textbf{Case: } M = \mathbf{bind}\ x_{p,\tau} = M_0\ \mathbf{in}\ N.}$ If $M_0 \in \mathrm{Val}$ the reduction step will not change the value of any memory location so the conclusion trivially holds. Otherwise we can apply the induction hypothesis on M_0 to get that if $\langle \Sigma, M_0, S \rangle \rightarrow \langle \Sigma', M_0', S' \rangle$ then $\forall \Theta. S =_{\alpha}^{\Theta \setminus \Omega} S'$. From this we can conclude that if $\langle \Sigma, \mathbf{bind}\ x_{p,\tau} = M_0\ \mathbf{in}\ N, S \rangle \rightarrow \langle \Sigma', \mathbf{bind}\ x_{p,\tau} = M_0'\ \mathbf{in}\ N, S' \rangle$ then $\forall \Theta. S =_{\alpha}^{\Theta \setminus \Omega} S'$.

□

Lemma 8 (Uninterrupted high evaluation). *If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ and $\alpha \not\preceq^{\Omega} w$ and $\vdash S$ and $\langle \Sigma, M, S \rangle \rightarrow^{*} \langle \Sigma', v, S' \rangle$ then $\forall \Theta. S =_{\alpha}^{\Theta \setminus \Omega} S'$.*

Proof. We prove this by induction on the length of the derivation. We have two cases: Either (1) M is a value, or (2) we can reduce M.

$\boxed{\textbf{Case: } 1.}$ If M is a value, then by typing we have $\Sigma' = \Sigma$, so we can take 0 steps to get $S = S'$ and the conclusion holds.

$\boxed{\textbf{Case: 2.}}$ M is not a value, so by progress we can reduce it further. By preservation and α-Ω-high, since $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ and $\vdash S$ and $\langle \Sigma, M, S \rangle \to \langle \Sigma'', M', S'' \rangle$ then $\forall \Theta . S =_{\alpha}^{\Theta \setminus \Omega} S''$ and $M' \in H_{\alpha}^{\Omega}$. We can then do $\langle \Sigma'', M', S'' \rangle \to^* \langle \Sigma', v, S' \rangle$ and the induction hypothesis gives us $\forall \Theta . S'' =_{\alpha}^{\Theta \setminus \Omega} S'$. By transitivity we can conclude that $\forall \Theta . S =_{\alpha}^{\Theta \setminus \Omega} S'$. □

Apart from these lemmas pertaining to high programs, we need to prove the lemma from section 4 that connects the visibility of a policy to its guards. We first give a helper lemma that gives an alternative interpretation of visibility:

Lemma 9. $\alpha \lessdot p(\Theta) \Leftrightarrow \exists (\Phi \Rightarrow \alpha) \in p . \Phi \subseteq \Theta$

Proof. If $\alpha \lessdot p(\Theta)$ then by definition we have $\{\} \Rightarrow \alpha \in p(\Theta)$, which in turn means that we have $\{\} \Rightarrow \alpha \in \{\Phi \setminus \Theta \Rightarrow \beta \mid \Phi \Rightarrow \beta \in p\}$. For this to be true we must have $\exists \Phi \Rightarrow \alpha . \Phi \setminus \Theta = \{\}$, which means $\Phi \subseteq \Theta$. □

Now we can prove the guard lemma:

Lemma 3 (Guard lemma). If $\alpha \not\lessdot p$, then $\alpha \not\lessdot^{\Omega} p$ where $\Omega = \bigcup guards_{\alpha}(p)$.

Proof. By contradiction. Assume that $\exists \Theta . \alpha \lessdot p(\Theta \setminus \Omega)$. This means that $\exists \Phi \Rightarrow \alpha \in p . \Phi \neq \{\}$ and $\Phi \subseteq \Theta \setminus \Omega$. But if $\Phi \Rightarrow \alpha \in p$ then $\Phi \subseteq \Omega$, so we have a contradiction. □

With these lemmas in hand, we can finally move on to prove the main lemma, that our candidate relation is a bisimulation.

Lemma 4 ($\bigcup\limits_{\Omega} R_{\alpha}^{\Omega}$ is a bisimulation). If $\langle \Sigma, M \rangle R_{\alpha}^{\Omega} \langle \Delta, N \rangle$ and $\vdash S$ and

$$\langle \Sigma, M, S \rangle \xrightarrow{p} \langle \Sigma', M', S' \rangle \;\&\; \Theta \supseteq \Sigma \;\&\; S =_{\alpha}^{\Theta} T \;\&\; \vdash T$$

then $\vdash S'$, and there exists Δ', N', T' such that

either $\langle \Delta, N, T \rangle \to^* \langle \Delta', N', T' \rangle \;\&\; \vdash T' \;\&\; S' =_{\alpha}^{\Theta \setminus \Omega} T' \;\&\; \langle \Sigma', M' \rangle R_{\alpha}^{\Omega'} \langle \Delta', N' \rangle$,

or $\langle \Delta, N, T \rangle \Uparrow$,

where $\Omega' = \Omega \cup \bigcup guards_{\alpha}(p(\Theta))$

Proof. We will conduct the proof by induction on the size of the typing derivation of $\langle \Sigma, M \rangle$.

By preservation, we already know that if we reduce a well-typed term in the presence of a well-typed store, the resulting term and store are going to be well-typed as well, so we will not bother about controlling either of those facts in the rest of this proof.

In many cases we will implicitly make use of properties of the relations $=_{\alpha}^{\Theta}$ and $\preceq$, such as transitivity and monotonicity, which we discussed in the main body of the paper.

$\boxed{\textbf{Case: } \langle \Sigma, M \rangle R_{\alpha}^{\Omega} \langle \Delta, N \rangle \textbf{ by rule 1.}}$ We have that $M = N$. Without loss of generality we will assume that $M, N \notin H_{\alpha}^{\Omega}$, since we will cover that when considering rule 2. This means that

M cannot be a variable, a value, a dereferencing, a recursion, an open or a close. Remains a reference creation, an assignment, a conditional, an application or a bind.

Subcase: $M \equiv \mathbf{ref}_{p'}\ x_{p,\tau}$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbf{ref}_{p'}\ x_{p,\tau}, S\rangle \xrightarrow{\top} \langle \Sigma, \ell_{p',\tau}, S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]\rangle \quad \text{and}$$
$$\langle \Delta, \mathbf{ref}_{p'}\ x_{p,\tau}, T\rangle \xrightarrow{\top} \langle \Delta, \ell_{p',\tau}, T[\ell_{p',\tau} \mapsto T(x_{p,\tau})]\rangle$$

By typing we have

$$\frac{p(\Sigma) \preceq p'}{\Sigma \vdash \mathbf{ref}_{p'}\ x_{p,\tau} : \mathit{ref}_{p'}\ \tau, (\bot, p') \Rightarrow \Sigma}$$

Suppose that $\alpha \lessdot p(\Theta)$. Then $S(x_{p,\tau}) = T(x_{p,\tau})$ and we have $S[\ell_{p',\tau} \mapsto S(x_{p,\tau})] =^{\Theta\setminus\Omega}_{\alpha} T[\ell_{p',\tau} \mapsto T(x_{p,\tau})]$. If on the other hand we have that $\alpha \not\lessdot p(\Theta)$, then $S(x_{p,\tau}) = T(x_{p,\tau})$ is not guaranteed, so to assure that $S[\ell_{p',\tau} \mapsto S(x_{p,\tau})] =^{\Theta\setminus\Omega}_{\alpha} T[\ell_{p',\tau} \mapsto T(x_{p,\tau})]$ we require that $\alpha \not\lessdot p'(\Theta \setminus \Omega)$. This follows from $p(\Sigma) \preceq p'$. We can finally conclude $\langle \Sigma, \ell_{p',\tau}\rangle R^{\Omega}_{\alpha} \langle \Delta, \ell_{p',\tau}\rangle$ by rule 1.

Subcase: $M \equiv x_{p,\mathit{ref}_{p''}\ \tau} := y_{p',\tau}$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, x_{p,\mathit{ref}_{p''}\ \tau} := y_{p',\tau}, S\rangle \xrightarrow{\top} \langle \Sigma, (), S[S(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto S(y_{p',\tau})]\rangle \quad \text{and}$$
$$\langle \Delta, x_{p,\mathit{ref}_{p''}\ \tau} := y_{p',\tau}, T\rangle \xrightarrow{\top} \langle \Delta, (), T[T(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto T(y_{p',\tau})]\rangle$$

By typing we have

$$\frac{p(\Sigma) \sqcup p'(\Sigma) \preceq p''}{\Sigma \vdash x_{p,\mathit{ref}_{p''}\ \tau} := y_{p',\tau} : \mathit{unit}, (\bot, p'') \Rightarrow \Sigma}$$

Now we reason by cases according to the following three exhaustive conditions: (i) $\alpha \lessdot p(\Theta)$ and $\alpha \lessdot p'(\Theta)$, (ii) $\alpha \not\lessdot p(\Theta)$, and (iii) $\alpha \not\lessdot p'(\Theta)$.
In case (i) we have that $S(x_{p,\mathit{ref}_{p''}\ \tau}) = T(x_{p,\mathit{ref}_{p''}\ \tau})$ and $S(y_{p',\tau}) = T(y_{p',\tau})$ and thus we have

$$S[S(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto S(y_{p',\tau})] =^{\Theta\setminus\Omega}_{\alpha} T[T(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto T(y_{p',\tau})]$$

as required.
In case (ii) then $S(x_{p,\mathit{ref}_{p''}\ \tau}) = T(x_{p,\mathit{ref}_{p''}\ \tau})$ is not guaranteed, so to assure that

$$S[S(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto S(y_{p',\tau})] =^{\Theta\setminus\Omega}_{\alpha} T[T(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto T(y_{p',\tau})]$$

we require that $\alpha \not\lessdot p''(\Theta \setminus \Omega)$. This follows from $p(\Sigma) \preceq p''$.
In case (iii) then $S(y_{p',\tau}) = T(y_{p',\tau})$ is not guaranteed, so to assure that

$$S[S(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto S(y_{p',\tau})] =^{\Theta\setminus\Omega}_{\alpha} T[T(x_{p,\mathit{ref}_{p''}\ \tau}) \mapsto T(y_{p',\tau})]$$

we again require that $\alpha \not\lessdot p''(\Theta \setminus \Omega)$. This follows from $p'(\Sigma) \preceq p''$.

We can finally conclude $\langle \Sigma, ()\rangle R^{\Omega}_{\alpha} \langle \Delta, ()\rangle$ by rule 1.

Subcase: $M \equiv \mathbf{if}\ x_{p,\mathit{bool}}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbf{if}\ x_{p,\mathit{bool}}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, S\rangle \xrightarrow{p} \langle \Sigma, M_i, S\rangle \quad \text{and}$$
$$\langle \Delta, \mathbf{if}\ x_{p,\mathit{bool}}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, T\rangle \xrightarrow{p} \langle \Delta, M_j, T\rangle$$

where $i, j \in \{0, 1\}$. By typing we have

$$\frac{\Sigma \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Sigma' \quad p(\Sigma) \preceq w_0 \sqcap w_1}{\Sigma \vdash \textbf{if } x_{p,bool} \textbf{ then } M_0 \textbf{ else } M_1 : \tau, (p(\Sigma) \sqcup r_0 \sqcup r_1, w_0 \sqcap w_1) \Rightarrow \Sigma'}$$

Assume that $\alpha \vartriangleleft p(\Theta)$. Then $S(x_{p,bool}) = T(x_{p,bool})$ which means that $i = j$ and $\bigcup guards_\alpha(p(\Theta)) = \{\}$. We have $M_i = M_j$ and we can conclude $\langle \Sigma, M_i \rangle R_\alpha^\Omega \langle \Delta, M_j \rangle$ by rule 1.

Assume now instead that $\alpha \ntriangleleft p(\Theta)$. Then $S(x_{p,bool}) = T(x_{p,bool})$ is not guaranteed, and thus possibly $M_i \neq M_j$. But since $\alpha \ntriangleleft p(\Theta)$, by the guard lemma we have that $\alpha \ntriangleleft^{\Omega'} p(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p(\Theta))$, and further since $p(\Sigma) \preceq w_i$ and $\Theta \supseteq \Sigma$ we have that $\alpha \ntriangleleft^{\Omega'} (w_i)$. This means that $M_i, M_j \in H_\alpha^{\Omega \cup \Omega'}$ and we can conclude $\langle \Sigma, M_i \rangle R_\alpha^{\Omega \cup \Omega'} \langle \Delta, M_j \rangle$ by rule 2.

Subcase: $M \equiv x_{p,\tau_f}\ y_{p',\tau'}$ **where** $\tau_f = (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau.$ For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, x_{p,\tau_f}\ y_{p',\tau'}, S \rangle \xrightarrow{p} \langle \Sigma, M_0, S[z_{p',\tau'} \mapsto S(y_{p',\tau'})] \rangle, \quad \text{and}$$
$$\langle \Delta, x_{p,\tau_f}\ y_{p',\tau'}, T \rangle \xrightarrow{p} \langle \Delta, M_1, T[w_{p',\tau'} \mapsto S(y_{p',\tau'})] \rangle$$

where $S(x_{p,\tau_f}) = \lambda z_{p',\tau'}.M_0$ and $T(x_{p,\tau_f}) = \lambda w_{p',\tau'}.M_1$. By typing we have

$$\frac{p(\Sigma) \preceq w_f}{\Sigma \vdash x_{p,\tau_f}\ y_{p',\tau'} : \tau, (p(\Sigma) \sqcup r_f, w_f) \Rightarrow \Sigma'}$$

Assume that $\alpha \vartriangleleft p(\Theta)$. Then $S(x_{p,\tau_f}) = T(x_{p,\tau_f})$, which in turn means that $z_{p',\tau'} = w_{p',\tau'}$ and $M_0 = M_1$. It also means that $\bigcup guards_\alpha(p(\Theta)) = \{\}$, and we can conclude $\langle \Sigma, M_0 \rangle R_\alpha^\Omega \langle \Delta, M_1 \rangle$ by rule 1. We have $S[z_{p',\tau'} \mapsto S(y_{p',\tau'})] =_\alpha^{\Theta \setminus \Omega} T[z_{p',\tau'} \mapsto T(y_{p',\tau'})]$ since if $\alpha \vartriangleleft p'(\Theta)$ then $S(y_{p',\tau'}) = T(y_{p',\tau'})$.

Assume now instead that $\alpha \ntriangleleft p(\Theta)$. Then $S(x_{p,\tau_f}) = T(x_{p,\tau_f})$ is not guaranteed, and thus possibly $M_0 \neq M_1$. But since $\alpha \ntriangleleft p(\Theta)$, by the guard lemma we have that $\alpha \ntriangleleft^{\Omega'} p(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p(\Theta))$, and further since $p(\Sigma) \preceq w_f$ and $\Theta \supseteq \Sigma$ we have that $\alpha \ntriangleleft^{\Omega'} (w_f)$. This means that $M_0, M_1 \in H_\alpha^{\Omega \cup \Omega'}$ and we can conclude $\langle \Sigma, M_0 \rangle R_\alpha^{\Omega \cup \Omega'} \langle \Delta, M_1 \rangle$ by rule 2. We have $S[z_{p',\tau'} \mapsto S(y_{p',\tau'})] =_\alpha^{\Theta \setminus \Omega} T[w_{p',\tau'} \mapsto S(y_{p',\tau'})]$ since the equivalence relation doesn't care about variables not in the intersection of the domains of the two stores.

Subcase: $M \equiv \mathbb{E}[\textbf{bind } x_{p,\tau} = M_0 \textbf{ in } M_1].$ Here we must proceed by inspection of M_0. For all terms we have by typing of the inner term that

$$\frac{\Sigma \vdash M_0 : \tau, (r_0, w_0) \Rightarrow \Sigma' \quad \Sigma' \vdash M_1 : \tau_1, (r_1, w_1) \Rightarrow \Sigma'' \quad r_0(\Sigma') \preceq p}{\Sigma \vdash \textbf{bind } x_{p,\tau} = M_0 \textbf{ in } M_1 : \tau_1, (r_1, w_0 \sqcap w_1) \Rightarrow \Sigma''}$$

Subsubcase: $M_0 \equiv v.$ For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1], S \rangle \xrightarrow{\top} \langle \Sigma, \mathbb{E}[M_1], S[x_{p,\tau} \mapsto v] \rangle \quad \text{and}$$
$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1], T \rangle \xrightarrow{\top} \langle \Delta, \mathbb{E}[M_1], T[x_{p,\tau} \mapsto v] \rangle$$

We have that $S[x_{p,\tau} \mapsto v] =_\alpha^{\Theta \setminus \Omega} T[x_{p,\tau} \mapsto v]$ and we can conclude $\langle \Sigma, \mathbb{E}[M_1] \rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[M_1] \rangle$ by rule 1.

Subsubcase: $M_0 \equiv y_{p',\tau}$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = y_{p',\tau} \textbf{ in } M_1], S\rangle \xrightarrow{p'} \langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(y_{p',\tau}) \textbf{ in } M_1], S\rangle \quad \text{and}$$
$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = y_{p',\tau} \textbf{ in } M_1], T\rangle \xrightarrow{p'} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(y_{p',\tau}) \textbf{ in } M_1], T\rangle$$

Assume $\alpha \lhd p'(\Theta)$. Then $S(y_{p',\tau}) = T(y_{p',\tau})$ and we can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(y_{p',\tau}) \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(y_{p',\tau}) \textbf{ in } M_1]\rangle.$$

Assume now instead that $\alpha \ntriangleleft p'(\Theta)$. Then $S(y_{p',\tau}) = T(y_{p',\tau})$ is not guaranteed, so we could end up with two different values. By the guard lemma we know that $\alpha \ntriangleleft^{\Omega'} p'(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p'(\Theta))$. By typing we know that $p'(\Sigma) \preceq p$, so this means that $\alpha \ntriangleleft^{\Omega'} p(\Theta)$, and we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(y_{p',\tau}) \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(y_{p',\tau}) \textbf{ in } M_1]\rangle.$$

Subsubcase: $M_0 \equiv \textbf{ref}_{p''}\ y_{p',\tau'}$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\ \tau'} = \textbf{ref}_{p''}\ y_{p',\tau'} \textbf{ in } M_1], S\rangle \xrightarrow{p'}$$
$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\ \tau'} = \ell_{p'',\tau'} \textbf{ in } M_1], S[\ell_{p'',\tau'} \mapsto S(y_{p',\tau'})]\rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\ \tau'} = \textbf{ref}_{p''}\ y_{p',\tau'} \textbf{ in } M_1], T\rangle \xrightarrow{p'}$$
$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\ \tau'} = \ell_{p'',\tau'} \textbf{ in } M_1], T[\ell_{p'',\tau'} \mapsto T(y_{p',\tau'})]\rangle$$

We reason that we have the required equality on the memories like we did for the reference creation at top level. We can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\ \tau'} = \ell_{p'',\tau'} \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\ \tau'} = \ell_{p'',\tau'} \textbf{ in } M_1]\rangle.$$

Subsubcase: $M_0 \equiv !y_{p',ref_{p''}\ \tau}$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = !y_{p',ref_{p''}\ \tau} \textbf{ in } M_1], S\rangle \xrightarrow{p' \sqcap p''} \langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(S(y_{p',ref_{p''}\ \tau})) \textbf{ in } M_1], S\rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = !y_{p',ref_{p''}\ \tau} \textbf{ in } M_1], T\rangle \xrightarrow{p' \sqcap p''} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(T(y_{p',ref_{p''}\ \tau})) \textbf{ in } M_1], T\rangle$$

By typing we know that

$$\frac{}{\Sigma \vdash\ !y_{p',ref_{p''}\ \tau} : \tau, (p'(\Sigma) \sqcup p''(\Sigma), \top) \Rightarrow \Sigma}$$

Assume $\alpha \lhd p'(\Theta)$ and $\alpha \lhd p''(\Theta)$. Then $S(S(y_{p',\mathit{ref}_{p''}\,\tau})) = T(T(y_{p',\mathit{ref}_{p''}\,\tau}))$ and we can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = S(S(y_{p',\mathit{ref}_{p''}\,\tau}))\ \mathbf{in}\ M_1] \rangle R^{\Omega}_{\alpha} \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = T(T(y_{p',\mathit{ref}_{p''}\,\tau}))\ \mathbf{in}\ M_1] \rangle.$$

Assume now instead that $\alpha \ntriangleleft p'(\Theta)$ or $\alpha \ntriangleleft p''(\Theta)$. Then we cannot guarantee
$S(S(y_{p',\mathit{ref}_{p''}\,\tau})) = T(T(y_{p',\mathit{ref}_{p''}\,\tau}))$, so we could end up with two different values. By the guard lemma we know that $\alpha \ntriangleleft^{\Omega'} p'(\Theta)$ and $\alpha \ntriangleleft^{\Omega''} p''(\Theta)$ where $\Omega' = \mathit{guards}_{\alpha}(p'(\Theta))$ and $\Omega'' = \mathit{guards}_{\alpha}(p''(\Theta))$. But since $p'(\Sigma) \sqcup p''(\Sigma) \preceq p$ and $\Theta \supset \Sigma$ we know that $\alpha \ntriangleleft^{\Omega' \cup \Omega''} p$ and thus we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = S(S(y_{p',\mathit{ref}_{p''}\,\tau}))\ \mathbf{in}\ M_1] \rangle R^{\Omega \cup \Omega' \cup \Omega''}_{\alpha} \\ \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = T(T(y_{p',\mathit{ref}_{p''}\,\tau}))\ \mathbf{in}\ M_1] \rangle.$$

Subsubcase: $M_0 \equiv y_{p',\mathit{ref}_{p''}\,\tau} := z_{p',\tau}$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\mathit{unit}} = y_{p',\mathit{ref}_{p''}\,\tau} := z_{p',\tau}\ \mathbf{in}\ M_1], S \rangle \xrightarrow{\top} \\ \langle \Sigma, \mathbf{bind}\ x_{p,\mathit{unit}} = ()\ \mathbf{in}\ M_1, S[S(y_{p',\mathit{ref}_{p''}\,\tau}) \mapsto S(z_{p',\tau})] \rangle$$

and

$$\langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\mathit{unit}} = y_{p',\mathit{ref}_{p''}\,\tau} := z_{p',\tau}\ \mathbf{in}\ M_1], T \rangle \xrightarrow{\top} \\ \langle \Delta, \mathbf{bind}\ x_{p,\mathit{unit}} = ()\ \mathbf{in}\ M_1, T[T(x_{p',\mathit{ref}_{p''}\,\tau}) \mapsto T(z_{p',\tau})] \rangle$$

We reason that we have the required equality on the memories like we did for the assignment at top level. We can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\mathit{unit}} = ()\ \mathbf{in}\ M_1] \rangle R^{\Omega}_{\alpha} \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\mathit{unit}} = ()\ \mathbf{in}\ M_1] \rangle.$$

Subsubcase: $M_0 \equiv \mathbf{if}\ y_{p',\mathit{bool}}\ \mathbf{then}\ N_0\ \mathbf{else}\ N_1$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = \mathbf{if}\ y_{p',\mathit{bool}}\ \mathbf{then}\ N_0\ \mathbf{else}\ N_1\ \mathbf{in}\ M_1], S \rangle \xrightarrow{p'} \langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_i\ \mathbf{in}\ M_1], S \rangle$$

and

$$\langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = \mathbf{if}\ y_{p',\mathit{bool}}\ \mathbf{then}\ N_0\ \mathbf{else}\ N_1\ \mathbf{in}\ M_1], T \rangle \xrightarrow{p'} \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_j\ \mathbf{in}\ M_1], T \rangle$$

Assume $\alpha \lhd p'(\Theta)$. Then $N_i = N_j$ and we conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_i\ \mathbf{in}\ M_1] \rangle R^{\Omega}_{\alpha} \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_j\ \mathbf{in}\ M_1] \rangle$$

Assume now instead that $\alpha \ntriangleleft p'(\Theta)$, then possibly $N_i \neq N_j$. By the guard lemma we know $\alpha \ntriangleleft^{\Omega'} p'(\Theta)$ where $\Omega' = \bigcup \mathit{guards}_{\alpha}(p'(\Theta))$. By the same reasoning as for a conditional at top level we know that $\langle \Sigma, N_i \rangle R^{\Omega \cup \Omega'}_{\alpha} \langle \Delta, N_j \rangle$.

By $p'(\Sigma) \preceq p$ and $\Theta \supseteq \Sigma$ we know $\alpha \not\lessdot^{\Omega'} p$, and we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_i\ \mathbf{in}\ M_1] \rangle R_\alpha^{\Omega \cup \Omega'} \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_j\ \mathbf{in}\ M_1] \rangle.$$

Subsubcase: $M_0 \equiv y_{p',\tau_f}\ z_{p'',\tau'}$ **where** $\tau_f = (\tau', p'') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, y_{p',\tau_f}\ z_{p'',\tau'}, S \rangle \xrightarrow{p'} \langle \Sigma, N_0, S[w_{p'',\tau'} \mapsto S(z_{p'',\tau'})] \rangle \quad \text{and}$$
$$\langle \Delta, y_{p',\tau_f}\ z_{p'',\tau'}, T \rangle \xrightarrow{p} \langle \Delta, N_1, T[w'_{p'',\tau'} \mapsto T(z_{p'',\tau'})] \rangle$$

where $S(y_{p',\tau_f}) = \lambda w_{p'',\tau'}.N_0$ and $T(y_{p',\tau_f}) = \lambda w'_{p'',\tau'}.N_1$.

We reason that we have the required equality on memories like we did for application at top level.

Assume $\alpha \lessdot p'(\Theta)$. Then $N_0 = N_1$ and we conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_0\ \mathbf{in}\ M_1] \rangle R_\alpha^{\Omega} \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_1\ \mathbf{in}\ M_1] \rangle.$$

Assume now instead that $\alpha \not\lessdot p'(\Theta)$, then possibly $N_0 \neq N_1$. By the guard lemma we know $\alpha \not\lessdot^{\Omega'} p'(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p'(\Theta))$. By the same reasoning as for an assignment at top level we know that $\langle \Sigma, N_0 \rangle R_\alpha^{\Omega \cup \Omega'} \langle \Delta, N_1 \rangle$.

By $p'(\Sigma) \preceq p$ and $\Theta \supseteq \Sigma$ we know $\alpha \not\lessdot^{\Omega'} p$, and we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_0\ \mathbf{in}\ M_1] \rangle R_\alpha^{\Omega \cup \Omega'} \langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = N_1\ \mathbf{in}\ M_1] \rangle.$$

Subsubcase: $M_0 \equiv \mathbf{open}\ \sigma$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = \mathbf{open}\ \sigma\ \mathbf{in}\ M_1], S \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, \mathbf{bind}\ x_{p,\tau} = ()\ \mathbf{in}\ M_1, S \rangle \quad \text{and}$$
$$\langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = \mathbf{open}\ \sigma\ \mathbf{in}\ M_1], T \rangle \rightarrow \langle \Delta \cup \{\sigma\}, \mathbf{bind}\ x_{p,\tau} = ()\ \mathbf{in}\ M_1, T \rangle$$

We can conclude (by rule 1) that

$$\langle \Sigma \cup \{\sigma\}, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = ()\ \mathbf{in}\ M_1] \rangle R_\alpha^{\Omega} \langle \Delta \cup \{\sigma\}, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = ()\ \mathbf{in}\ M_1] \rangle.$$

Subsubcase: $M_0 \equiv \mathbf{close}\ \sigma$. Similar to the previous case.

Subsubcase: $M_0 \equiv \mathbf{rec}\ y_{\perp,\tau}.v$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = \mathbf{rec}\ y_{\perp,\tau}.v\ \mathbf{in}\ M_1], S \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, \mathbf{bind}\ x_{p,\tau} = v\ \mathbf{in}\ M_1, S[y_{\perp,\tau} \mapsto v] \rangle$$

and

$$\langle \Delta, \mathbb{E}[\mathbf{bind}\ x_{p,\tau} = \mathbf{rec}\ y_{\perp,\tau}.v\ \mathbf{in}\ M_1], T \rangle \rightarrow \langle \Delta \cup \{\sigma\}, \mathbf{bind}\ x_{p,\tau} = v\ \mathbf{in}\ M_1, T[y_{\perp,\tau} \mapsto v] \rangle$$

By typing of M_0 we know

$$\frac{\Sigma \vdash v : \tau, (\perp, \top) \Rightarrow \Sigma}{\Sigma \vdash \mathbf{rec}\ y_{\perp,\tau}.v : \tau, (\perp, \top) \Rightarrow \Sigma}$$

and we have $S[y_{\perp,\tau} \mapsto v] =_{\alpha}^{\Theta\backslash\Omega} T[y_{\perp,\tau} \mapsto v]$. We can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1] \rangle R_{\alpha}^{\Omega} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1] \rangle.$$

Case: $\langle \Sigma, M \rangle R_{\alpha}^{\Omega} \langle \Delta, N \rangle$ by rule 2. We have that $M, N \in H_{\alpha}^{\Omega}$, For $\Theta \supseteq \Sigma$ we have $S =_{\alpha}^{\Theta} T$ and $\langle \Sigma, M, S \rangle \xrightarrow{p} \langle \Sigma', M', S' \rangle$. By the highness lemma we know that $S' =_{\alpha}^{\Theta\backslash\Omega} S$, and so we can choose to match this by taking 0 steps for N, i.e. $\langle \Delta, N, T \rangle \to^{0} \langle \Delta, N, T \rangle$, and since $S =_{\alpha}^{\Theta\backslash\Omega} T$, by transitivity we have $S' =_{\alpha}^{\Theta\backslash\Omega} T$ as required. By lemma ? we know that subject reduction preserves the highness property, so we have $M' \in H_{\alpha}^{\Omega}$, and thus $M', N \in H_{\alpha}^{\Omega'}$ where $\Omega' = \Omega \cup \bigcup guards_{\alpha}(p(\Theta))$ and we conclude $\langle \Sigma', M' \rangle R_{\alpha}^{\Omega'} \langle \Delta, N \rangle$ by rule 2.

Case: $\langle \Sigma, M \rangle R_{\alpha}^{\Omega} \langle \Delta, N \rangle$ by rule 3. We have that $M = \mathbb{E}[\textbf{bind } x_{p,\tau} = M_0 \textbf{ in } M_1]$ and $N = \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1]$ and that $\langle \Sigma, M_0 \rangle R_{\alpha}^{\Omega} \langle \Delta, N_0 \rangle$ and $\alpha \not\preccurlyeq^{\Omega} p$.

Here we can separate two cases — either M_0 is a value, or we can reduce in M_0.

Subcase: $M_0 \equiv v$. For $\Theta \supseteq \Sigma$ we have $S =_{\alpha}^{\Theta} T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1], S \rangle \xrightarrow{\top} \langle \Sigma, \mathbb{E}[M_1], S[x_{p,\tau} \mapsto v] \rangle$$

If M_0 is a value, then by lemma 5 we have N_0 is high. We have that either $\langle \Delta, N_0, T \rangle \Uparrow$, in which case we have $\langle \Delta, N, T \rangle \Uparrow$, or $\langle \Delta, N_0, T \rangle \to^{*} \langle \Delta', v', T' \rangle$. In the latter case we have $\forall \Theta . T =_{\alpha}^{\Theta\backslash\Omega} T'$ by lemma bigstep high and by transitivity that $S =_{\alpha}^{\Theta\backslash\Omega} T'$. This means we can match the step in M by

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1], T \rangle \to^{*} \langle \Delta', \mathbb{E}[\textbf{bind } x_{p,\tau} = v' \textbf{ in } M_1], T' \rangle \to \\ \langle \Delta', \mathbb{E}[M_1], T'[x_{p,\tau} \mapsto v'] \rangle$$

Since we know $\alpha \not\preccurlyeq^{\Omega} p$ we have $S[x_{p,\tau} \mapsto v] =_{\alpha}^{\Theta\backslash\Omega} T'[x_{p,\tau} \mapsto v']$. We can conclude (by rule 1) that $\langle \Sigma, \mathbb{E}[M_1] \rangle R_{\alpha}^{\Omega} \langle \Delta', \mathbb{E}[M_1] \rangle$. **Subcase: $M_0 \notin$ Val.** By the progress lemma this means we can reduce M_0, so for $\Theta \supseteq \Sigma$ we have $S =_{\alpha}^{\Theta} T$ and $\langle \Sigma, M_0, S \rangle \xrightarrow{p'} \langle \Sigma', M_0', S' \rangle$. By the induction hypothesis we have that either $\exists \Delta', N_0', T' . \langle \Delta, N_0, T \rangle \to^{*} \langle \Delta', N_0', T' \rangle$ and $S' =_{\alpha}^{\Theta\backslash\Omega} T'$ and $\langle \Sigma', M_0' \rangle R_{\alpha}^{\Omega'} \langle \Delta', N_0' \rangle$ where $\Omega' = \Omega \cup guards_{\alpha}(p'(\Theta))$, or $\langle \Delta, N_0, T \rangle \Uparrow$. In the latter case we have

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1], T \rangle \Uparrow$$

For the former case we can choose to match the reduction in M by

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1], T \rangle \to^{*} \langle \Delta', \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0' \textbf{ in } M_1], T' \rangle$$

and we conclude

$$\langle \Sigma', \mathbb{E}[\textbf{bind } x_{p,\tau} = M_0' \textbf{ in } M_1] \rangle R_{\alpha}^{\Omega'} \langle \Delta', \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0' \textbf{ in } M_1] \rangle$$

by rule 3.

□

A.3 Proofs that Flow Lock Security Implies Noninterference

In this appendix we provide details of the proof that flow lock security implies noninterference. The strategy is to

- Strengthen the definition of location indistinguishability at a given level to include variables;
- Generalise the definition of noninterference to a binary relation between pairs of programs, and strengthen to include variables in the store;
- Specialise the definition of α indistinguishability to lock-free policies.
- Specialise the definition of flow lock bisimulation to lock-free programs and stores.

Recall that we consider a lattice of security levels $\langle \mathcal{L}, \sqsubseteq, \sqcup \rangle$, and a policy $\mathrm{level} : \mathsf{Loc} \to \mathcal{L}$ that fixes the intended security level of the storage locations in the program.

We assume that these locations are typed, but we will elide typing issues in the following discussion. Programs P and Q operate over these locations, and are assumed to be of unit type, and are assumed not to perform any location allocation.

To be precise we need to define the lock-free semantics for configurations of the form $\langle P, S \rangle$. But it is easy to see that if P is lock free then the lock part of the state can simply be ignored since it neither influences computation nor does it change, so transitions for $\langle P, S \rangle$ re derived by simply projecting out the lock state in the transition system.

Definition 8 (Noninterference (Generalised)). Given two stores S and T, and a level $k \in \mathcal{L}$, define S and T to be *indistinguishable at level k*, written $S \equiv_k T$, iff the location domains of S and T are the same, and for all $\ell \in \mathrm{dom}(S)$ and for all $x \in \mathrm{dom} S \cap \mathrm{dom} T$ such that $\mathrm{level}(\ell) \sqsubseteq k$ we have $S(\ell) = T(\ell)$. and $S(x) = T(x)$.

Now define, for each level k, the binary relation $\sim_k^{\mathrm{NI}}$ on lock-free programs as follows: $P \sim_k^{\mathrm{NI}} Q$ if for all S and T such that $S \equiv_k T$, whenever $\langle P, S \rangle$ and $\langle Q, T \rangle$ are terminating configurations, $\langle P, S \rangle \to^* \langle (), S' \rangle$ and $\mathrm{dom}(S') \backslash \mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$ and $S \equiv_k T$, then there exists a T' such that $\langle Q, T \rangle \to^* \langle (), T' \rangle$, and $S' \equiv_k T'$.

The following lemma states that these definitions are indeed generalisations, and can be seen by inspection of the definitions:

Lemma 10. 1. *For all lock free stores S and T, $S \equiv_k T$ implies $S =_k T$.*

2. *For all closed (i.e. variable free) lock free programs P, if $P \sim_k^{\mathrm{NI}} P$ for all k, then P is noninterfering.*

Now we build a bridge from the opposite side, by specialising the definition of flow lock security to lock-free programs. Firstly we note that for lock free stores, level indistinguishability $\equiv_k$ given above coincides with the indistinguishability relation $=_k^{\Theta}$ for any Θ, i.e.

Lemma 11. $S =_k^{\Theta} T \iff S \equiv_k T$

The proof is again just by inspection of the definition, so we omit a detailed argument. Now we turn to the definition of bisimulation for lock-free programs.

Lemma 12. *Define the largest symetric relation between lock-free programs, $\approx_k$, such that whenever*

$$P \approx_k Q \ \&\ S \equiv_k T \ \&\ \mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\} \ \&\ \langle P, S\rangle \to \langle P', S'\rangle$$

then there exits Q', T' such that

either $\langle Q, T\rangle \to \langle Q', T'\rangle$ & $S \equiv_k T$ & $P' \approx_k Q'$,

or $\langle Q, T\rangle \Uparrow$,

Then we have that $\langle \Sigma, P\rangle \sim_\alpha^\Omega \langle \Delta, Q\rangle$ implies $P \approx_k Q$.

The proof is again straightforward by specialisation of the bisimulation definition, and using the preceeding lemma.

Now we can provide the proof that if P is flow lock secure then P is noninterfering.

Proof. (Theorem 1) Suppose that closed program P is flow lock secure. I.e. for all levels k $\langle\{\}, P\rangle \sim_k \langle\{\}, P\rangle$. By lemma 12 this implies that $P \approx_k P$. We will prove that $P \approx_k P$ implies $P \equiv_k P$, from which it follows that P is noninterferring by lemma 10.

In order to prove that $P \approx_k P$ implies $P \equiv_k P$ we will prove the more general statement, namely that

$$\forall P, Q. P \approx_k Q \Longrightarrow P \equiv_k Q$$

i.e. we prove this for open P and Q.

Assume that $P \approx_k Q$ and that $\langle P, S\rangle \to^n \langle (), S'\rangle$, $\langle Q, T\rangle$ is terminating, $S \equiv_k T$ and $\mathrm{dom}(S')\backslash\mathrm{dom}(S)\cap\mathrm{dom}(T) = \{\}$. We are then required to show that $\langle Q, T\rangle \to^n \langle (), T'\rangle$ for some T' such that $S' \equiv_k T'$, and we do so by induction by induction on n

$\boxed{\textbf{Base case: } n = 0.}$ In this case $P = ()$ and hence $S = S'$. By the convergence assumption we know that $\langle Q, T\rangle \to \cdots \to \langle Q_i, T_i\rangle \to \cdots \to \langle (), T_m\rangle$ for some stores T_i, and since we are free to choose store variable names, we can assume that $\mathrm{dom}(T_i) \setminus \mathrm{dom}(T) \cap \mathrm{dom}(S) = \{\}$. By symmetry $Q \approx_k P$, and thus from the definition of $\approx_k$, each of these computation steps from can only be matched by taking zero steps from $\langle P, S\rangle$, and hence $S \equiv_k T_m$ as required.

$\boxed{\textbf{Inductive case: } \langle P, S\rangle \to \langle P_1, S_1\rangle \to^* \langle (), S'\rangle.}$ Since $\mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$, and since computation only increases the domain of the store, $\mathrm{dom}(S') \subseteq \mathrm{dom}(S_1)$, and hence $\mathrm{dom}(S_1)\backslash\mathrm{dom}(S)\cap\mathrm{dom}(T) = \{\}$. Given this, by assumption that $P \approx_k Q$ and from the fact that $\langle Q, T\rangle$ does not diverge, we know that $\langle Q, T\rangle \to^* \langle Q_1, T_1\rangle$ for some T_1 such that $S_1 \equiv_k T_1$. Since $\mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$ and $\mathrm{dom}(S_1) \supseteq \mathrm{dom}(S)$ it follows that $\mathrm{dom}(S')\backslash\mathrm{dom}(S_1) \cap \mathrm{dom}(T) = \{\}$. Now we know that $\mathrm{dom}(T_1) = \mathrm{dom}(T) \cup X$ for some set of variables X. We can assume that X is chosen to be disjoint from $\mathrm{dom}(S')\backslash\mathrm{dom}(S_1)$, and hence we have that $\mathrm{dom}(S')\backslash\mathrm{dom}(S_1) \cap \mathrm{dom}(T_1) = \{\}$. Now we can apply the induction hypothesis to obtain the existance of a T' such that $\langle Q_1, T_1\rangle \to^* \langle (), T'\rangle$ $S_1 \equiv_k T_1$ as required. □