Information Flow Aware Languages
Kuifje emerges from the Shadows

Jack Drury

University of New South Wales, Australia

Abstract. Communicating secrets has always been challenging. For all applications, from traditional ciphers to modern encrypted messaging apps, it is important to understand in detail how secret information is being treated. Seemingly benign actions within a process can lead to partial or full leakage of secrets. Performing an information flow analysis allows us to see how a process handles secrets and determine whether any leaks take place and if so where they occur. A simple model, referred to as the Shadow semantics [1], was introduced to track non-quantitative information flow through programs. Extending this model with probabilities leads to a quantitative information flow model which has been named Kuifje [2]. Both have rigorously defined semantics and proofs of their desirable algebraic properties. The problem is that these definitions and proofs exist only on paper. Here we formalise both in the proof assistant Isabelle/HOL and construct proofs about the properties of these formalisations. This work intends to be the first steps towards an environment where the models can be reasoned about algebraically with (formal-)confidence and without the need to dive into the semantics.

1 Introduction

In this report we look at two languages that were developed to analyse information flow, formalise them in the proof assistant Isabelle/HOL [3] and prove properties about the formalisations. The first is the Shadow Semantics [1] which is non-quantitative and deals with sets of possible hidden variable values. If there is an attacker that knows the source code and is watching the execution of a program, the Shadow Semantics tracks the possible values of the hidden variables based on what an attacker can see at each point in a program.

The second language is called Kuifje [2] and it was developed with the intention of using it for analysing quantitative information flow. So instead of dealing with sets of values it deals with probability distributions of values. It tracks the probability of different hidden variable values as a program progresses.

Both languages have been formalised and have had their important algebraic properties proven, but these proofs exist only on paper. The work in this report represents the first steps towards the creation of an environment where people can reason about Kuifje and Shadow programs formally by using their algebraic properties without resorting to the semantics.

This report first defines the abstract syntax and semantics of Shadow before looking at some case studies that helped formulate a set of useful algebraic

properties. In the following sections we define the abstract syntax and semantics of Kuifje and prove that it respects the monad laws. The report explains most of the proofs and the appendix contains the full Isabelle/HOL proofs for the interested reader.

1.1 Information Flow

For as long as there has been communication, there have been people who want to communicate secrets. For a secret to remain secret, the information that defines it must be inaccessible to those deemed unworthy. We can imagine various situations in which access to certain information should be restricted:

- Spies encrypting messages to ensure they cannot be read if intercepted.
- Databases that only reveal specific information to privileged users/agents.
- Firewalls that block packets containing sensitive information from reaching unsavoury destinations.

We could say that the systems above are *secure* if they achieve their stated goals. The problem with this view is that the above systems only work to prevent information from being released; they do not control its propagation afterwards. In such situations the spy must trust his allies not to spread the secret after decryption, the database must trust privileged users not to share sensitive information with other users and the firewall must trust the savoury destinations not to leak its secrets. Such trust is risky, but is there a way to guarantee that the accessor of private information will not misuse the data? If the accessor is a program, such questions can be answered by *information flow* analysis of the source code of the program. This involves looking at how the program uses the sensitive information and ensuring it does not *flow* to an undesirable (publicly visible) location.

1.2 Non-interference

To analyse the flow of sensitive information we need a way to differentiate private information from public information. This is done by declaring two types of variables:

- Hidden variables ($h \in H$) that should remain private and are traditionally referred to as high-security variables.
- Visible variables ($v \in V$) that are public and are traditionally referred to as low-security variables.

If a program runs and is able to keep all visible variables independent of the hidden variables it is said to exhibit *non-interference*.

However, non-interference is often too restrictive as it says that regardless of the value of the hidden variables, the same initial visible variables will always result in the same observable program behaviour [4]. This means that a user that can only access visible variables will not be able to learn anything about

the hidden variables. The problem with this is that many programs can not respect non-interference as they need to reveal a small amount of information about the secret to perform useful work.

Take, for example, a program that requires a password to allow access to a system. If the password is a hidden variable, the user will learn what the password is *not* by providing an incorrect password as a visible input.

Such coarse reasoning is not always helpful. What if we want to know the extent of the hidden information revealed by the visible variables and circumstances under which this holds? Such questions are important and have been heavily studied, see the review by Sabelfeld and Myers [5] which surveys almost 150 relevant papers. One particular technique that addresses this question involves the Shadow Semantics which was introduced by Morgan [1].

If we imagine a program acting over a state space of hidden variables h and visible variables v. The Shadow Semantics introduce a third variable H. The *shadow* of h that represents all possible values of h at a certain point in the program. This is equivalent to the *ignorance* of an attacker. It is the set of possible values for h based on what they have been able to glean from the visible variables and its source code. For example if H is $\{0, 1\}$, it implies that the program is in a state where h is 1 or in a state where h is 0. By following how H changes through a program we are able to monitor what an attacker knows for certain. If H is a singleton, then the attacker knows exactly the value of the hidden variables, otherwise they have an increasing amount of ignorance.

The Shadow Semantics deal with the application of a program to a shadow in order to produce a *hyper-shadow*. The hyper-shadow is a set of sets, each of the inner sets represents an outcome of *visible nondeterminism*. Visible nondeterminism is a nondeterministic fork in the execution of the program that can be observed by the attacker. If a hyper-shadow is a singleton such as $\{\{1, 2\}\}$, then the program did not contain any visible nondeterminism and the attacker knows that h is either 1 or 2. On the other hand if the hyper-shadow is $\{\{1, 2\}, \{0, 1\}\}$, it means there was a visible nondeterministic fork in the program and therefore (depending on which path was taken) the attacker knows one of two things: they either know that h is a value in $\{1, 2\}$ or they know that h is a value in $\{0, 1\}$.

1.3 Quantitative Information Flow

The Shadow Semantics treat each possible state within the shadow equally. Certain states may be removed or added but no state is more or less likely than any other. We can easily imagine scenarios where some outcomes or program branches are more likely than others and therefore the attacker may know that some values within the shadow have a higher probability of being the true value than others. This means that the program is actually more vulnerable than you would expect based on the shadow alone.

Quantitative theories of information flow (QIF) were introduced to deal with this. They aim not only to determine whether a program leaks information, but also to figure out *how much* information has leaked.

A theory of QIF was first introduced in the early 1980's by Denning [6] and a number of others have followed since [7, 8, 9, 4]. The early theories used measurements based on Shannon Entropy to calculate the amount of uncertainty an attacker would have regarding the values of hidden variables.

Before Smith pointed out the flaws in models based on Shannon Entropy, the consensus was that mutual information should be used to quantify uncertainty [4]. Mutual information, written $I(X;Y)$, is the amount of information shared between random variables X and Y. It is equal to the reduction in the Shannon Entropy of one random variable by learning the value of the other.

$$I(X;Y) = H(X) - H(X|Y)$$

Where $H(X)$ is the Shannon Entropy of the random variable X and $H(X|Y)$ is the conditional entropy which represents the uncertainty about X given full knowledge of Y. The mutual information is symmetric and non-negative. To see where this breaks down we consider a deterministic program that takes the mod 16 of a hidden $16k$ bit variable, if it evaluates to zero it will assign a visible variable to the value of the hidden variable (thus revealing the hidden variable), otherwise it will assign the visible variable to 1. If the attacker guesses that hidden variable has the same value as the visible variable they will be correct 1/16th of the time (assuming a uniform distribution over the values of the hidden variable).

Since the program is deterministic the visible variable (V) is completely determined by the value of the hidden variable (W) therefore $H(V|W) = 0$ and the information leakage is $I(V;W) = H(V) \approx k + 0.087$. Using the symmetry of mutual information we can calculate the remaining uncertainty in W which is $H(W|V) = H(W) - I(V;W) \approx 15k - 0.087$.

If instead we have a program that leaks the last $k+1$ bits of W into V we have $H(V) = k+1$ since $k+1$ bits are copied from the hidden variable and the rest of the bits are certain. We also have $H(W|V) = 15k - 1$ since we learn about the other $k+1$ bits of H from V. This means that $15k-1$ bits of uncertainty remain for the hidden W after execution of the program. This is less uncertainty than for the first program which had $15k - 0.087$ bits of uncertainty remaining after execution. This suggests that an attacker learns more about the hidden variable by watching the execution of the second program than they do by watching the second program.

Smith identified that while an attacker can correctly guess the value of W with a probability of 1/16 after the first program, the attacker has only a $1/2^{15k-1}$ chance of guessing the value after the execution of the second program ($15k - 1$ being the number of unknown bits).

Smith claims that a useful QIF theory should be able to provide security guarantees. The above examples show that Shannon Entropy based methods can be unhelpful, claiming that the first program is *better* than the second even though the second program makes it much more difficult to guess the hidden value.

As implied by this analysis, Smith suggests that a better measurement of the *vulnerability* of a program would be Bayes Vulnerability which is the maximum probability that a secret could be guessed in one go. This is because a clever attacker would guess the value which has the highest probability of being the secret.

After Smith suggested this improved metric, it was pointed out by Alvim et al. that both Bayes Vulnerability and Shannon Entropy are instances of an infinite family of *gain functions* [10]. These generalised gain functions allow modelling of more complex scenarios such as what is it worth if an attacker guesses a value close to the true secret or a part of the secret or a property of the secret or even guessing the secret within a certain number of attempts.

After this, a further relevant generalisation was made by McIver et al. noting that entropies that are determined by gain functions are characterised by being concave and continuous over distributions [11].

These generalisations showed that no matter which entropy is being used there is a common structure to the QIF analyses. The chosen entropy is applied to the prior distribution before the program is executed and it is applied conditionally to the joint distribution that is produced after the program has been executed. The post-execution joint distribution has been named a *hyper-distribution* [12, 13] and it is a probability distribution over posterior distributions, where each posterior distribution is the result of the input distribution conditioned on a particular outcome of the (probabilistic) program.

Hyper-distributions unify QIF analyses that use the class of gain functions identified above.

1.4 Kuifje

The use of hyper-distributions allows QIF to be given a monadic semantics. The distribution monad ($\mathbb{D}$) takes some type $\mathcal{A}$ to the discrete distributions on $\mathcal{A}$ (this would be $\mathbb{D}\mathcal{A}$). So computations of type $\mathcal{A} \to \mathbb{D}\mathcal{A}$ are computations taking some type to distributions over that type. A compositional language that takes distributions as inputs and returns hyper-distributions (as mentioned above) would need to represent computations of type $\mathbb{D}\mathcal{A} \to \mathbb{D}^2\mathcal{A}$.

Recently a QIF extension of The Shadow Semantics has been introduced and it does just that. The extension is called Kuifje and it has been given a monadic representation in Haskell [2]. Kuifje is what you get if you extend the Shadow Semantics with probabilities. Where the Shadow Semantics has a *set* of possible values for the hidden variable, Kuifje has a *probability distribution* of possible values, matching each possible value to a probability. Where the hyper-shadows are a *set of sets* of possible values, Kuifje has hyper-distributions which are a *distribution of distributions* of possible values. The hyper-distributions can be thought of intuitively as an outer distribution that assigns probabilities to each visible outcome of the program and the inner distributions assign probabilities to the values of the hidden variables given that visible outcome.

Due to the similarity of the structures in both Kuifje and Shadow, the algebraic behaviour of the two is nearly identical. Despite these similarities, the

semantics are quite different as dealing with probability distributions is more challenging than dealing with sets.

1.5 Isabelle/HOL Formalisation

Both Kuifje and the Shadow Semantics have rigorous formal definitions and proofs of their important properties, but these definitions and proofs exist only on paper. The purpose of Kuifje and the Shadow Semantics is to provide guarantees about the security of programs, therefore we should be certain that an analysis performed using either of them is actually correct.

Pen and paper proofs are fallible and so the goal of this project is to formalise both the Shadow Semantics and Kuifje in the proof assistant Isabelle/HOL [3] and prove algebraic properties about both, such that many of the pen and paper proofs can be machine checked. The algebraic properties that will be targeted are those that act as the building blocks of many of the larger proofs.

Isabelle/HOL provides an environment in which we can define mathematical constructs using the semantics of its higher order logic. We can then use Isabelle/HOL to create machine-checked proofs of certain properties of our constructs.

In addition to the usual predicate logic symbols (∃, ¬, ∧...) with the expected semantics, Isabelle/HOL has a rich syntax for defining and manipulating new objects. Below we briefly outline parts of Isabelle/HOL that are most important for our Shadow and Kuifje constructions. We present some examples to illustrate the syntax.

```
type-synonym 'h prog = 'h set ⇒ ('h set) set


definition
    CHOOSE :: ('h ⇒ 'h set) ⇒ 'h prog
where
    (CHOOSE f) hs ≡ {⋃y∈{f h | h. h∈hs}. y}
```

Fig. 1. Examples of Isabelle/HOL syntax. First a type alias is made where `'h prog` takes a `'h` set to a set of sets of `'h`. Then a function called `CHOOSE` is defined. The type is declared with the double colons and then a set theoretic definition is provided.

The **type-synonym** command enables creating an alias for a type. This alias is only used to enhance readability and Isabelle/HOL will automatically translate the alias when dealing with it. In the above example we create an alias called `prog` and we say that the type `'h prog` takes a set of objects of type `'h` and returns a set of sets of type `'h`. In this case `'h` is a type variable as it can be replaced by any type (e.g. integers) and the definition will hold.

The **definition** command allows us create a constant nonrecursive definition. In the above example (Fig. 1) we define `CHOOSE`, we declare its type with the two colons. Following the **where** keyword we then provide a constant set theoretic definition for `CHOOSE` in terms of an input function f and an input set hs, where the inputs must obey the type declaration. `CHOOSE` is a function that uses a function `f` to assign new values to variables.

We can define a recursive function using the **fun** command. The following syntax is then very similar to that of **definition** except we are able to use pattern matching in the definition.

Isabelle/HOL uses $\equiv$ to denote equality and curly braces represent sets. The right hand side of the $\equiv$ in Fig. 1 is the singleton set made up of the result of a union iteration. The notation is telling us to take the union over all y's (this is the y to the right of the period) such that y is an element of the inner set comprehension. The set comprehension syntax says: return the set containing every $f\ h$ such that there exists an h and h is in hs.

That covers most of the definitions, but another very important part of Isabelle/HOL is the syntax used to state and prove lemmas. We breakdown the example in Fig. 2.

```
lemma bool_encryption_lemma:
  fixes   t :: bool
    and H :: bool set
  shows
    SKIP H =
    NewVar t (
               CHOOSE (λ(_, h). {(True, h), (False, h)}) ;;
               REVEAL {
                       {
                        (g, h) | g h. g∈{True, False} ∧ h∈H ∧
                                                        XOR g h  = k

                       } | k. k∈{True, False}
                      }
             ) H
```

Fig. 2. Example of Isabelle/HOL lemma syntax. First the name of the lemma (`bool_encryption_lemma`) so it can be referred to by other lemmas. The **fixes** keyword sets a property (in this case `t` is a boolean and `H` is a set of booleans). The **shows** keyword is followed by the statement to be proved.

The keyword **lemma** indicates that we want to prove the statement that follows. Following **lemma** is the name of the lemma and then a colon. **fixes** is used to set a property of some statement, in this case we are setting the type of the variable t to booleans and the type of H to sets of booleans.

shows is followed by the statement we want to prove, in this case it is the equality between *SKIP H* and more complicated statement that introduces a local variable randomly assigns it a boolean value and then reveals the xor of that variable with an existing hidden variable.

This statement would then be followed by a sequence of commands usually of the form **apply** x where x is a rule or a tactic that transforms the statement in some valid way. These transformations would continue until the statement can be transformed to *True*.

2 Abstract Syntax and Semantics of Shadows

A Shadow program is a mathematical model that takes in a set of some type $\mathcal{H}$ and returns a set of sets of $\mathcal{H}$ (i.e. something of type $\mathbb{P}\mathcal{H} \to \mathbb{P}^2\mathcal{H}$).

2.1 The Abstract Syntax of Shadows

The language takes the form of a simple imperative language. It is similar to Dijkstra's Guarded Command Language (GCL) [14] which was designed to be compact, clear and to make reasoning about programs easier. There are three important differences between the Shadow language and GCL:

- The language adds a `reveal` command for revealing the value of some expression to an attacker.
- Assignment becomes an internal nondeterministic choice between alternatives. With ordinary assignment being a degenerate case.
- Demonic choice is explicit here and is visible to an attacker, whereas GCL would represent it as a choice between expressions whose guards evaluate to true.

We lay out the abstract syntax below in extended Backus-Naur form, where $\texttt{PROG}[\mathcal{H}]$ is a Shadow program of type $\mathbb{P}\mathcal{H} \to \mathbb{P}^2\mathcal{H}$ for some base type $\mathcal{H}$.

$$
\begin{array}{lll}
\texttt{PROG}[\,\mathcal{H}\,] & ::= & \texttt{SKIP} \\
& \mid & \texttt{REVEAL rs} \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{rs} :: \mathbb{P}\mathbb{P}\mathcal{H} \\
& \mid & \texttt{CHOOSE f} \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{f} :: \mathcal{H} \to \mathbb{P}\mathcal{H} \\
& \mid & \texttt{DEMONIC PROG}[\mathcal{H}]\ \texttt{PROG}[\mathcal{H}] \\
& \mid & \texttt{COMPOSE PROG}[\mathcal{H}]\ \texttt{PROG}[\mathcal{H}] \\
& \mid & \texttt{IfThenElse b PROG}[\mathcal{H}]\ \texttt{PROG}[\mathcal{H}] \qquad\qquad \texttt{b} :: \mathrm{Bool} \\
& \mid & \texttt{NewVar t PROG}[\tau \times \mathcal{H}] \qquad\qquad\qquad\qquad\quad \texttt{t} :: \tau
\end{array}
$$

Where `SKIP` is the program that does nothing (except to type check it creates a singleton set out of its input). `REVEAL` takes an expression (in the form of a set of sets `rs`) and reveals it (for a deeper explanation see the semantics in the section below). `CHOOSE` is hidden (from an attacker) nondeterministic assignment, `DEMONIC` is a nondeterministic choice that is visible to an attacker.

`COMPOSE` is program composition, `IfThenElse` is the usual construct where if `B` evaluates to true the first `PROG` path is taken, otherwise the second is taken. `NewVar` allows a local variable to be introduced into the scope of the following `PROG` therefore its type grows by τ (where τ is the type of the new variable being declared).

2.2 The Semantics of Shadows

Shadow programs keep track of what an attacker knows about the variables of a program at different stages of execution. It achieves this by being fed a set of type $\mathcal{H}$ representing what an attacker knows for certain at this point in the execution of the program. If it is at the start of the program and the attacker knows nothing, it will be the set of all possible combinations of the variables in the state space. As parts of the program are executed and information leaks out to the attacker, the Shadow semantics will produce a set of sets of type $\mathcal{H}$ known as a hyper-shadow (which is of type $\mathbb{P}^2\mathcal{H}$). Each inner set of $\mathcal{H}$ represents a possible execution path that the attacker knows about (due to the visible nondeterminism of the program).

Within each possible execution path, the number of elements will decrease as the attacker learns more about the variables or increase as they are obfuscated by hidden nondeterministic assignments. If one of these execution paths becomes a singleton set, it implies that if this path is taken, the attacker has full knowledge of all variables in the system.

This method assumes that the attacker can see the source code and has infinite computing power.

To represent this language in Isabelle/HOL we created a shallow embedding. This means that we directly implemented each syntactic element from the previous section as a function in Isabelle/HOL. Therefore the semantics are embedded in Isabelle's Higher Order Logic. To produce this embedding, we started with a denotational semantics for each term and created an Isabelle/HOL definition, hence each term is directly translated into a function that takes a set of type $\mathcal{H}$ as an input and returns a set of sets of $\mathcal{H}$ (i.e. they are of the type $\mathbb{P}\mathcal{H} \to \mathbb{P}^2\mathcal{H}$).

The denotational semantics are given below, where the brackets $[\![\cdot]\!]$ are an evaluation function that takes the syntactic elements to their mathematical definition ($[\![\cdot]\!] :: \texttt{PROG}[\mathcal{H}] \to (\mathbb{P}\mathcal{H} \to \mathbb{P}^2\mathcal{H})$).

$$
\begin{aligned}
[\![\texttt{SKIP}]\!]\ \texttt{H} &= \{\texttt{H}\} \\
[\![\texttt{REVEAL rs}]\!]\ \texttt{H} &= \{\texttt{H} \cap \texttt{S} \mid \texttt{S} \in \texttt{rs}\} \\
[\![\texttt{CHOOSE f }]\!]\ \texttt{H} &= \{\bigcup_{\texttt{h}\in\texttt{H}} \texttt{f(h)}\} \\
[\![\texttt{DEMONIC P Q}]\!]\ \texttt{H} &= (\ [\![\texttt{P}]\!]\ \texttt{H}\ ) \cup (\ [\![\texttt{Q}]\!]\ \texttt{H}\ ) \\
[\![\texttt{COMPOSE P Q}]\!]\ \texttt{H} &= \bigcup_{\texttt{H'}\in\ [\![\texttt{P}]\!]\ \texttt{H}} [\![\texttt{Q}]\!]\ \texttt{H'} \\
[\![\texttt{IfThenElse B P Q}]\!]\ \texttt{H} &= (\ [\![\texttt{P}]\!]\ \{\texttt{h} \in \texttt{H} \mid \texttt{B(h)}\}\ ) \cup (\ [\![\texttt{Q}]\!]\ \{\texttt{h} \in \texttt{H} \mid \neg\texttt{B(h)}\}\ ) \\
[\![\texttt{NewVar t P}]\!]\ \texttt{H} &= \{\{\texttt{h} \mid (\_,\texttt{h}) \in \texttt{S}\} \mid \texttt{S} \in\ [\![\texttt{P}]\!]\ (\{\texttt{t}\} \times \texttt{H})\}
\end{aligned}
$$

Here SKIP returns the singleton set that contains what was passed to it. This is because SKIP does nothing and so the attackers knowledge is unchanged.

REVEAL rs returns the intersection of H with each set inside rs. This is because REVEAL is given a set which contains sets that are separated based on some expression. For example, if we have the boolean expression $x = y$, the REVEAL statement would return a set of two sets, one of which contains all states in which the expression is true and the other contains all states in which the expression is false. The intuition behind this is that each of the two inner sets represents a possible path the program could take and the attacker knows which path is taken, therefore each inner set is the attackers knowledge given that a certain execution path is taken.

CHOOSE is a hidden, nondeterministic assignment, so it is interpreted as the singleton set of the union of function f applied to every state in the statespace. It is the singleton of the union because the amount of uncertainty the attacker has is increasing. This means there is no partitioning of the statespace into known paths and the number of possible elements is increasing.

DEMONIC P Q returns the union of program P applied to H with the result of program Q applied to H. This is because demonic choice is a visible nondeterministic choice between program Q and program P, so we need the result of each to be visible to the attacker, hence the union of their results is taken, leaving their inner sets separate.

COMPOSE P Q, is just program composition. It applies P to H and then applies Q to each path within that result.

IfThenElse B P Q is as expected, P is applied to the states in which the boolean function B evaluates to true and Q is applied to the states in which B evaluates to false and their union is taken.

NewVar t P, adds the variable t to the statespace by taking the cross product $\{\texttt{t}\}\times$ H, then program P is applied to this new statespace (in the definition this is represented by $[\![\texttt{P}]\!]\ (\{\texttt{t}\} \times \texttt{H})$). This will result in a

The Each path in the result is then selected and the new variable is subsequently removed. The purpose of NewVar is to create a local variable, therefore it is important that the type is correct, which can only be done by adding the

variable to the statespace and then removing it again. In general the program `P` will begin with a `CHOOSE` statement to assign some values to the new variable `t`.

2.3 Isabelle's Shadow

Based on the above denotational semantics we construct definitions in Isabelle/HOL and present them below in Fig. 3.

Note that the `COMPOSE` definition also creates an infix version of the function represented by two semicolons.

3 Shadowing the Dining Cryptographers

The dining Cryptographers is a problem first introduced by Chaum in the 1980's [15]. Chaum used it to demonstrate a method for sending anonymous messages. It considers the case of three cryptographers sitting around a table having just finished dinner at a restaurant. They are informed by the waiter that the bill has been paid anonymously. The cryptographers want to know whether it was one of them who paid the bill (but not which one) or whether it was a third party (the NSA in Chaum's example). One cryptographer will flip a coin secretly with the cryptographer on their left and then do the same with the cryptographer on the right. The cryptographer will then announce aloud whether they paid the bill; if the two coins they flipped land on the same side they will tell the truth and otherwise the cryptographer will lie. The remaining two cryptographers will covertly flip a coin between themselves and then each make an announcement similar to the first cryptographer, but the statement will be based on the new coin and the coin they shared with the first cryptographer.

This means that if there are an odd number of cryptographers claiming to have paid then one of the cryptographers really did pay. Otherwise if there is an even number, it implies that the NSA paid.

McIver and Morgan [16] perform an algebraic analysis of the dining cryptographers using the Shadow semantics. They show that revealing whether or not one of the cryptographers paid is equivalent to introducing hidden coin flips between each cryptographer and having the cryptographer announce what they see based on the above protocol. We are interested in checking the correctness of their algebraic analysis. To do so, we must prove, based on the semantics, that each of the algebraic steps taken are valid. Before we dive into the whole analysis of the dining cryptographers we start with some smaller problems that are subsets of their analysis. First we look at the Encryption Lemma and then a simpler variant of the dining cryptographers with only two parties.

3.1 Encryption Lemma

An important building block for McIver and Morgans algebraic proofs is the the Encryption Lemma. It states that if you have a hidden boolean variable, `h`,

```
type-synonym 'h prog = 'h set ⇒ ('h set) set

definition
    SKIP ::'h prog
where
    SKIP hs ≡ {hs}

definition
    REVEAL ::('h set) set ⇒ 'h prog
where
    (REVEAL rs) hs ≡ {inter hs s | s. s∈rs}

definition
    CHOOSE ::('h ⇒ 'h set) ⇒ 'h prog
where
    (CHOOSE f) hs ≡ {⋃y∈{f h | h. h∈hs}. y}

definition
    DEMONIC ::'h prog ⇒ 'h prog ⇒ 'h prog
where
    (DEMONIC p q) hs ≡ union (p hs) (q hs)

definition
    COMPOSE ::'h prog ⇒ 'h prog ⇒ 'h prog (infixr ;; 60)
where
    (COMPOSE p q) hs ≡ ⋃hs'∈(p hs). (q hs')

definition
  IfThenElse :: ('h ⇒ bool) ⇒ 'h prog ⇒ 'h prog ⇒ 'h prog
where
  IfThenElse b p q hs ≡ union (p {h. h∈hs ∧ b h})
                              (q {h'. h'∈hs ∧ ¬(b h')})

definition
    NewVar :: 'tau ⇒ ('tau × 'h) prog ⇒ 'h prog
where
    (NewVar init p) hs ≡ {{snd h |h. h ∈ s} | s. s ∈(p ({init} × hs))}
```

Fig. 3. The embedding of the Shadow semantics into Isabelle/HOL.

and you initialise a new local hidden variable `h'` to a random boolean, revealing the XOR of `h` and `h'` is equivalent to `skip` (i.e. it leaks no information to the attacker). Intuitively this makes sense as the randomness of `h'` implies that nothing can be gained by looking at the result of a comparison between it and `h` . Having a statement that is equivalent to `skip` is useful because `skip` acts like the identity when composed with other programs and therefore it can be plucked out of thin air when trying to show program equivalences.

The Encryption Lemma is the basis for the "one-time pad", an (ideally) uncrackable cryptographic technique that was widely used throughout the 20th century. The caveats are that the two parties must have a completely random and secret shared key that is at least as long as the message being sent and they must never reuse the key.

Assuming we are within the context of a hidden variable `h`, McIver and Morgan represent the Encryption Lemma in concrete syntax as:

Expression 1. Encryption Lemma

```
[hid h' · h' :∈ {0,1}; reveal h' ⊕ h] = skip
```

The left hand side introduces a new variable `h'`, assigns it a random value from the set {0, 1} and then `reveal`s the XOR of `h'` and `h`.

We can present this in abstract syntax using our Isabelle/HOL definition with the following Lemma:

```
lemma bool_encryption_lemma:
  fixes   t :: bool
    and H :: bool set
  shows
    SKIP H =
    NewVar t (
              CHOOSE (λ(_, h). {(True, h), (False, h)}) ;;
              REVEAL {{ (g, h) | g h. g∈{True, False} ∧ h∈H ∧
                                      XOR g h  = k}|k. k∈{True, False}}
             ) H
```

Fig. 4. The encryption lemma in Isabelle/HOL.

This is exactly the same as the version McIver and Morgan present. We declare a new variable `t`, randomly assign it a value in {True, False} and then `REVEAL` the result of the XOR of the new variable and a hidden variable `h`. To prove this equality we expanded the semantics of each Shadow term and showed the equality for all cases of `H`. For the full Isabelle/HOL proof see the appendix.

The Encryption Lemma can be generalised to all surjective functions and is not just restricted to the type of Booleans, for the formalisation and Isabelle/HOL proof of this see the appendix.

3.2 Part-way with a Cream Cake

The dining cryptographers problem involves three parties, but is composed of individual coin flips between two parties. McIver and Morgan analyse such a scenario which I will dub the "Cream Cake Commotion".

Two cryptographers are looking forward to dessert. A waiter presents them with a single cream cake and a simple biscuit. Neither wishes to appear greedy in front of the waiter and reach for the cream cake. They decide to flip a coin between themselves so that the waiter can't see, if it is heads they write their preference (biscuit or cake) on a napkin, if it is tails they write the opposite of their preference. They hand the napkins to the waiter who will declare if they wrote the same thing on the napkins. If so, they skip dessert as they don't want to fight over who gets what. Otherwise, after the waiter leaves each cryptographer will safely grab their preferred dessert.

McIver and Morgan translate this into a form for Shadow analysis: given global hidden boolean variables `a` and `b`, we want to prove that revealing whether `a` equals `b` is equivalent to introducing a third hidden boolean variable, `c`, and revealing whether `a` equals `c` and then revealing whether `c` equals `b`. Intuitively this makes sense as `c` only exists briefly in its own local scope and its purpose is to use transitivity to check whether `a` equals `b`. In the style of McIver and Morgan we have:

Expression 2. Cream cake analysis

```
reveal a ≡ b
=
[ hid c · c :∈ {0,1}; reveal a ≡ c; reveal b ≡ c ]
```

To show this equality they begin with the left hand side and prepend a `skip`:

Expression 3. Cream cake step 1

```
reveal a ≡ b = skip ; reveal a ≡ b
```

This makes sense as a `skip` does nothing, but to make sure we prove the following in our Isabelle/HOL formalisation:

This is discharged by unfolding the semantic definition of `SKIP` and `COMPOSE` and simplifying. Next the Encryption Lemma is used to transform the `skip`, along with the fact that revealing the result of an XOR operation is the same as revealing the result of an equality test. This results in the following concrete expression:

```
lemma SKIP_then_p:
  p = SKIP ;; p
```

Fig. 5. Simple lemma stating that placing a `SKIP` in front of a program does nothing.

Expression 4. Cream cake step 2

```
skip ; reveal a ≡ b
=
[ hid c ·c :∈ {0,1}; reveal a ≡ c ] ; reveal a ≡ b
```

We have already described the Encryption Lemma and its proof, but we can represent the equality of revelations in Isabelle/HOL with the lemma:

```
lemma reveal_xor_is_reveal_equals:
  REVEAL {{(a,b)|a b. (a = b) = k}|k. k∈{True,False}}
  =
  REVEAL {{(a,b)|a b. (XOR a b) = k}|k. k∈{True,False}}
```

Fig. 6. Lemma to show that equality reveals the same information as an `XOR` operation

This is simply discharged by unfolding the definitions of `REVEAL` and `XOR` and applying Isabelle/HOL's tableau prover blast. The next and penultimate algebraic step pulls the reveal from outside `c`'s scope into scope:

Expression 5. Bring Reveal inside scope (step 3)

```
[ hid c ·c :∈ {0,1}; reveal a ≡ c] ; reveal a ≡ b
=
[ hid c ·c :∈ {0,1}; reveal a ≡c ; reveal a ≡ b ]
```

The scope of a new variable in the Isabelle/HOL definition is limited to the second input of `NewVar t P`, i.e. the new variable only exists within `P`. Therefore pulling a program from outside the scope and bringing it in is equivalent to bringing a program inside of a NewVar statement. For now it will suffice to show that we can bring the `reveal` statement inside the scope, ensuring that it does not impact the newly declared variable. To do this we need to perform a kind of lifting on the reveal statement from the statespace outside the scope (consisting of only Bool×Bool) to the statespace inside the scope with the extra variable (Bool×Bool×Bool). In Isabelle/HOL this looks like the lemma in Fig. 7:

```
lemma reveal_in_scope:
  fixes t::'a
  shows
     (
        NewVar t p  ;;
        REVEAL E
     ) H
   =
     (
       NewVar t (
                   p ;;
                   REVEAL {(UNIV::'a set)×e |e. e∈E}
                 )
     ) H
```

Fig. 7. Lemma bringing a `REVEAL` statement inside a new variable scope. The `UNIV` is required to ensure that the statement does not affect the statespace of the new variable.

`UNIV` is the set of all values of type `'a`. The form of the `REVEAL` expression is motivated by its semantics. `REVEAL` takes the intersection of the attackers knowledge with whatever is inside the `REVEAL` expression. Therefore, since our `REVEAL` is entering the scope of a variable that it should not know about, we must leave it untouched, hence we use the cartesian product of `UNIV` with our actual expression. The above Isabelle/HOL lemma is proved by unfolding the `NewVar`, `REVEAL` and `COMPOSE` definitions and then showing that the sets that are produced are equal (see appendix for the full Isabelle/HOL proof).

The final step uses the fact that, for boolean `a`, `b` and `c`, learning whether `a = b` and whether `a = c` determines the results of `a = c` and `b = c`. McIver and Morgan represent the final step in concrete syntax as:

Expression 6. Cream cake step 4

```
[ hid c · c :∈ {0,1}; reveal a ≡c ; reveal a ≡ b ]
=
[ hid c · c :∈ {0,1}; reveal a ≡c ; reveal b ≡c ]
```

In the Isabelle/HOL abstract syntax, it would appear as in Fig. 8.

This is proven using extensionality and showing that the resulting sets are equal (see appendix). This concludes the cream cakes example, formally proving that the scope adjustments, equalities and other algebraic manipulations made by McIver and Morgan are valid. This will prove a useful stepping stone as we go onto the analysis of the three dining cryptographers.

```
lemma equiv_reveals:
    REVEAL {{(a,b,c)|a b c. (a = c) = k ∧ b∈{True,False}}|k.
k∈{True,False}}
   ;;
    REVEAL {{(a,b,c)|a b c. (a = b) = k ∧ c∈{True,False}}|k.
k∈{True,False}}
   =
    REVEAL {{(a,b,c)|a b c. (a = c) = k ∧ b∈{True,False}}|k.
k∈{True,False}}
   ;;
    REVEAL {{(a,b,c)|a b c. (c = b) = k ∧ a∈{True,False}}|k.
k∈{True,False}}
```

Fig. 8. Lemma to show that equality reveals the same information as an `XOR` operation

3.3 Three Dining Cryptographers

So at last we get back to our protocol of interest, the three dining cryptographers. The ultimate goal of the analysis is to show (in the style of McIver and Morgan):

Expression 7. Three Cryptographers

```
reveal a ⊕ b ⊕ c
=
[
  hid l, m, r ·
    l :∈ {0,1};
    m :∈ {0,1};
    reveal l ⊕ a ⊕ m;
    r :∈ {0,1};
    reveal m ⊕ b ⊕ r;
    reveal l ⊕ c ⊕ r
]
```

Assuming a global declaration for the hidden booleans `a`, `b` and `c`.

To get to this equality they start with the result from the Cream Cake example, but with some minor differences. Instead of revealing the result of an equality test, an XOR is used and `a` is instantiated as `l⊕a` and `b` is instantiated as `b⊕r` and a nondeterministic assignment must be brought into a local variable scope. They show that within a global declaration of variables `a`, `b`, `l` and `r` we have the following equality:

Expression 8. Three Cryptographers step 1

```
r :∈ {0,1};
reveal l ⊕ (a ⊕ b) ⊕ r
```

```
=
[
  hid m ·
    m :∈ {0,1};
    reveal l ⊕ a ⊕ m;
    r :∈ {0,1};
    reveal m ⊕ b ⊕ r
]
```

If we introduce a SKIP to the LHS, then instantiate the Encryption Lemma with h' = m and h = l ⊕ a, then we get the first 3 lines of the RHS. We then need to pull the CHOOSE program for r and the reveal into scope. We already know it is possible for REVEAL, for CHOOSE the proof is very similar, but the Isabelle/HOL lemma is shown in Fig. 9.

```
lemma choose_in_scope:
  fixes t::'a
  shows
  (
      NewVar t p ;;
      CHOOSE E
   ) H
   =
   (
     NewVar t (
                 p ;;
                 CHOOSE (λh. {(fst h, b)|b. b∈(E (snd h))})
               )
   ) H
```

Fig. 9. This lemma shows that bringing a CHOOSE inside a new variable declaration. The extra arguments to the CHOOSE function are selected so that it does not affect the newly declared variable.

The lambda function provided in the CHOOSE program is chosen so that it ignores the new element in the statespace and only applies the function E to the old components of the statespace. The whole proof can be found in the appendix. We are then left with the problem of turning the in-scope reveal l⊕(a⊕b)⊕r into reveal m ⊕ b ⊕ r.

This comes straight out of proofs we already have from Isabelle/HOL as we know that revealing an equality is the same as revealing an XOR and we know that revealing the results of a = b and a = c determines the results of revealing a = c and b = c. If we instantiate this with:

```
a = l ⊕ a
b = b ⊕ r
c = m
```

We get exactly the result we desire and end up with McIver and Morgan's equality. At the next stage they show:

Expression 9. Three Cryptographers step 2

```
[
  hid l, r ·
    l :∈ {0,1};
    r :∈ {0,1};
    reveal l ⊕ (a ⊕ b) ⊕ r;
    reveal l ⊕ c ⊕ r
]
=
reveal a ⊕ b ⊕ c
```

The middle two lines of the LHS are from the equality we just showed. By grouping (`l`⊕`r`) together and (`a`⊕`b`) together we see the LHS has the form of two reveal statements that we have come to expect (where two reveal statements are equivalent to another two reveal statements). This allows them to turn `reveal l`⊕`c`⊕`r` into `reveal a`⊕`b`⊕`c` which they can then shift outside the local scope as the reveal does not mention `l` or `r` (see lemma reveal_in_scope). After that it is an application of a variant of the Encryption Lemma to arrive at the RHS. The variant declares two local hidden variables and assigns them a random value and then takes their `XOR` with a globally declared hidden variable (in this case that is `a`⊕`b`). In Isabelle/HOL we define this as:

```
lemma double_encryption_lemma:
  assumes BB ={(True,True),(True,False),(False,True),(False,False)}
              hs ⊆ BB
  shows
  SKIP hs
  =
  NewVar t (
             (
               CHOOSE (λ(_,x). {(lr,x)|lr. lr∈BB})
             ) ;;
               REVEAL {{(lr,x)|lr x. lr∈BB ∧ x∈BB
                                   ∧ (XOR
                                           (XOR (fst lr) (snd lr))
                                           (XOR (fst x) (snd x))
                                      ) = k
                        }|k. k∈{True,False}
                       }
          )
  hs
```

Fig. 10. This is a kind of 'double encryption lemma', where .

This is discharged by instantiating the general encryption lemma (see appendix) with a pair of booleans and the surjective function is made up of three `XOR` operations.

From here McIver and Morgan consider their analysis complete as they can substitute Expression 8 into Expression 9, pull a `reveal` and a non-deterministic assignment into the innermost scope (which we have already proven in Isabelle/HOL) which returns Expression 7 as was desired.

4 Abstract Syntax and semantics of Kuifje

A Kuifje program is a mathematical model that takes in a probability distribution over some type $\mathcal{H}$ and returns a distribution of distributions over $\mathcal{H}$ (i.e. a program is of type $\mathbb{D}\mathcal{H} \to \mathbb{D}^2\mathcal{H}$).

4.1 Abstract Syntax of Kuifje

Kuifje's abstract syntax is similar to Shadow's, except:

- Composition is built in, so sequences of commands are terminated with a `SKIP`.
- `REVEAL` is renamed to `OBSERVE` and `CHOOSE` is renamed to `UPDATE` this reflects the names in its Haskell implementation.
- `DEMONIC` is removed as a similar effect can be acheived by combining `OBSERVE` with a uniform probabilistic choice.
- A `WHILE` loop construction is introduced.

```
PROG[H] ::= SKIP
          | UPDATE f PROG[H]                          f :: H → DH
          | IF b PROG[H] PROG[H] PROG[H]          b :: H → DBool
          | WHILE b PROG[H] PROG[H]               b :: H → DBool
          | OBSERVE f PROG[H]                         f :: H → DO
```

`SKIP` does nothing to the input distribution (therefore it just returns a hyper-distribution with only the input distribution inside). `UPDATE` is a probabilistic update to the state and then `PROG` is executed. `IF` is the usual conditional, if `b` evaluates to true the first `PROG` is evaluated, if not then the second is, after evaluating the first or second `PROG` then the third is evaluated. The `WHILE` construct continually evaluates the first `PROG` while `b` is true, if `b` is false then the second `PROG` is evaluated. `OBSERVE` acts like the `reveal` from Shadow, it reveals the information in expression `f` and then evaluates the following program. It should be noted that `SKIP` is used to terminate sequences of commands by placing it in the final `PROG` slot of any of the other commands.

4.2 The Semantics of Kuifje

Kuifje programs track the probabilities of different values being the true values of the hidden variables of a program. The input to a Kuifje program is a prior distribution which represents the initial knowledge of an attacker. If the attacker knows nothing then this will be a uniform distribution over all possible values in the statespace of the hidden variables of the program.

A Kuifje program will take this prior distribution and manipulate it as information is leaked. As different externally *observable* paths are taken, the distribution will be split in to many distributions and each distribution will be assigned

the probability that its observable path was taken. The outermost distribution is the hyper-distribution (of type $\mathbb{D}^2\mathcal{H}$), the elements it contains represents different execution paths that can be distinguished by an attacker. Each of those paths (the inner distributions of type $\mathbb{D}\mathcal{H}$) then represent the probability of different hidden values given that program path was taken.

As Kuifje was designed to be implemented in Haskell, the semantics are represented as state transformation functions that exist in Haskell. We present those semantics here, using *sem* to represent the evaluation from the Kuifje abstract syntax to the Haskell-style monadic semantics. Note that the chosen representation for the distribution type is a list of pairs. The first element of the pair is the value of some type $\mathcal{H}$ and the second element is the probability which is represented by a rational number. The semantics make use of *return* and *bind* ($\gg\!=$) of the distribution monand, which we define along with the semantics for clarity. The unit *return* of the monad takes an element and then gives the point distribution containing only that element. The bind operator takes a distribution on type a and a function from type a to distributions on type b and computes the application of the function to the elements of the initial distribution. The *join* and *fmap* operators are then defined in terms of *return* and *bind* in the usual way. Also note that ${}_p\oplus$ is an infix probabilistic choice which performs the statement on the left with probability p and the statement on the right with probability *1-p*.

$$
\begin{aligned}
sem &:: \texttt{PROG}[\mathcal{H}] \to (\mathbb{D}\mathcal{H} \to \mathbb{D}^2\mathcal{H}) \\
sem\ \texttt{SKIP} &= return \\
sem\ \texttt{UPDATE f p} &= \lambda x.\ (huplift\ \texttt{f})\ x \gg\!= (sem\ \texttt{p}) \\
sem\ \texttt{IF c p q r} &= \lambda x.\ conditional\ \texttt{c}\ (sem\ \texttt{p})\ (sem\ \texttt{q})\ x \gg\!= (sem\ \texttt{r}) \\
sem\ \texttt{WHILE c p q} &= let\ wh\ \texttt{=}\ conditional\ \texttt{c}\ (\lambda x.\ (sem\ \texttt{p})\ x \gg\!= wh)\ (sem\ \texttt{q}) \\
&\quad\ in\ wh \\
sem\ \texttt{OBSERVE f p} &= \lambda x.\ (hobsem\ \texttt{f})\ x \gg\!=\ (sem\ \texttt{p}) \\
return &:: \mathcal{H} \to \mathbb{D}\mathcal{H} \\
return\ x &= [(x,1)] \\
(\gg\!=) &:: \mathbb{D}\mathcal{H} \to (\mathcal{H} \to \mathbb{D}\mathcal{G}) \to \mathbb{D}\mathcal{G} \\
d \gg\!= f &= [\ (y,\ p \times q) \mid (x,\ p) \leftarrow d,\ (y,\ q) \leftarrow (f\ x)\ ] \\
huplift &:: (\mathcal{H} \to \mathbb{D}\mathcal{H}) \to (\mathbb{D}\mathcal{H} \to \mathbb{D}^2\mathcal{H}) \\
huplift\ f &= return \circ (\gg\!= f) \\
conditional &:: (\mathcal{H} \to \mathbb{D}Bool) \to (\mathbb{D}\mathcal{H} \to \mathbb{D}^2\mathcal{H}) \to (\mathbb{D}\mathcal{H} \to \mathbb{D}^2\mathcal{H}) \to (\mathbb{D}\mathcal{H} \to \mathbb{D}^2\mathcal{H}) \\
conditional\ c\ t\ e\ d &= \textbf{let}\ d' \texttt{=} d \gg\!= \lambda s.\ c\ s \gg\!= \lambda b.\ return\ (b,\ s) \\
&\qquad\quad w_1 \texttt{=} sum\ [\ p \mid ((b,s),p) \leftarrow d',\ b\ ] \\
&\qquad\quad w_2 \texttt{=} 1 - w_1 \\
&\qquad\quad d_1 \texttt{=} [\ (s,\ p/w_1) \mid ((b,\ s),\ p) \leftarrow d',\ b\ ] \\
&\qquad\quad d_2 \texttt{=} [\ (s,\ p/w_2) \mid ((b,\ s),\ p) \leftarrow d',\ \neg b\ ] \\
&\qquad\quad h_1 \texttt{=} t\ d_1 \\
&\qquad\quad h_2 \texttt{=} e\ d_2 \\
&\qquad \textbf{in if}\quad d_2 = [\ ]\ \textbf{then}\ h_1 \\
&\qquad\quad \textbf{else if}\ d_1 = [\ ]\ \textbf{then}\ h_2 \\
&\qquad\qquad\qquad \textbf{else}\ join\ (h_1\ {}_{w_1}\!\oplus\ h_2) \\
hobsem &:: (\mathcal{H} \to \mathbb{D}\mathcal{O}) \to (\mathbb{D}\mathcal{H} \to \mathbb{D}^2\mathcal{H}) \\
hobsem\ f &= multiply \circ toPair \circ (\gg\!= obsem\ f) \\
&\textbf{where} \\
&\quad obsem :: (\mathcal{H} \to \mathbb{D}\mathcal{O}) \to (\mathcal{H} \to \mathbb{D}(\mathcal{H} \times \mathcal{O})) \\
&\quad obsem\ f\ =\ \lambda w.\ (w,\ x)\ (f\ x) \\
&\quad toPair :: \mathbb{D}(\mathcal{O} \times \mathcal{H}) \to (\mathbb{D}\mathcal{O} \times (\mathcal{O} \to \mathbb{D}\mathcal{H})) \\
&\quad toPair\ dp\ =\ (d,\ f) \\
&\quad \textbf{where} \\
&\qquad d\ =\ fmap\ fst\ dp \\
&\qquad f\ ws\ =\ \textbf{let}\ dpws\ \texttt{=}\ [\ (s,\ p) \mid ((ws',\ s),\ p) \leftarrow dp,\ ws'\ =\ ws\ ] \\
&\qquad\qquad\ \textbf{in}\ [\ (s,\ p/weight\ dpws) \mid (s,\ p) \leftarrow dpws\ ] \\
&\quad multiply :: (\mathbb{D}\mathcal{O} \times (\mathcal{O} \to \mathbb{D}\mathcal{H})) \to \mathbb{D}^2\mathcal{H} \\
&\quad multiply\ (d,\ f)\ =\ fmap\ f\ d
\end{aligned}
$$

`SKIP` takes a distribution and returns the point hyper-distribution (i.e. the hyper-distribution with only a single entry). It achieves this using the monadic *return* as defined for the distribution monad. `UPDATE f p` updates the input distribution by lifting the function `f` so that it now takes distributions to distributions of distributions, it then feeds this result to program `p`. `IF` and `WHILE` both make use of the *conditional* function. The goal of *conditional c t e d* is to take in `c` (a function of type $\mathcal{H} \to \mathbb{D}Bool$) apply it to the state, leak the outcomes of the condition, feed this result into the appropriate program (`t` or `e`) and then continue on with program `d`. The reason that the definition of *conditional* appears much more complex here is that 'fold fusion' has been used to remove intermediate data structures (this is also true of the *hobsem* definition). Details of this will not be presented here but the specifics of this case can be found in the original Kuifje paper [2] and more information on fold fusion can be found in the following resources [17, 18, 19].

`IF` makes simple use of the *conditional* function, but `WHILE` uses it to recursively evaluate `p` and then continue with program `q` where `c` produces False. `OBSERVE f p` is an explicit leak where `f` is the expression to leak and `p` is the program to continue on with after the leak. The complexity of the function *hobsem* that is used to lift `f` to a function that takes distributions to hyper distributions is again a result of fold fusion.

4.3 Kuifje in Isabelle/HOL

When implementing Kuifje in Isabelle/HOL we are interested in being able to test for equality of distributions easily. The chosen representation for distributions was as a list of pairs (which is the same as presented in [2]). The problem with this is that there could be many different representations of the same distribution. The list could have different orderings, or an element could appear multiple times with its probability split over the instances, or there could be elements included with zero probability. We consider all those cases to represent the same probability distribution, we want some kind of canonical form to exist so that we can check distributions for equality easily. To do so we restrict the elements of the distribution to be in the type class of linear orders. We also introduce a function `reductionDist` that removes all elements with zero probability and combines repeated elements and we introduce a modified bind operator (`=>>`) that applies `reductionDist` to the input distribution. Based on these points and the above semantics we produced the following definitions in Isabelle/HOL:

type-synonym `prob = rat`

type-synonym `'a dist = ('a × prob) list`

type-synonym `'a hyper = ('a dist) dist`

type-synonym `'s prog = 's dist ⇒ 's hyper`

definition
  SKIP :: ('a::linorder) dist ⇒ 'a hyper
where
  SKIP ≡ return

definition
  UPDATE :: (('s::linorder)⇒'s dist) ⇒ 's prog ⇒ 's prog
where
  UPDATE f p ≡ λx. (huplift f) x ≫= p

definition
  IF :: (('s::linorder) ⇒ bool dist) ⇒ 's prog ⇒ 's prog ⇒ 's prog ⇒ 's prog
where
  IF c p q r ≡ (λx. conditional c p q x ≫= r)

definition
  OBSERVE :: (('s::linorder) ⇒ ('o::linorder) dist) ⇒ 's prog ⇒ 's prog
where
  OBSERVE f p ≡ λx. (hobsem f) x ≫= p

definition
  return :: ('a::linorder) ⇒ 'a dist
where
  return x ≡ [(x,1)]

definition
  bind ::'a dist ⇒ ('a ⇒ 'b dist) ⇒ ('b::linorder) dist
where
  bind d f ≡ [(y, p * q). (x,p) ← d, (y,q) ← (f x)]

definition
  dist_fmap :: ('a ⇒ 'b) ⇒ 'a dist ⇒('b::linorder) dist
where
  dist_fmap f dx = dx ≫= (return ∘ f)

definition
  join :: ('a::linorder) hyper ⇒ 'a dist
where
  join x = x ≫= id

definition
  huplift :: (('a::linorder) ⇒ 'a dist) ⇒ ('a dist ⇒ 'a hyper)
where
  huplift f ≡ return ∘ (λd. d =>> f)

```
definition
  conditional :: (('s::linorder) ⇒ bool dist) ⇒ 's prog ⇒ 's prog ⇒ 's prog
where
  conditional c t e d ≡
    let d' = d =>> (λs. c s =>> (λb. return (b, s)))
      ; w1 = ∑p∈{p | p s. ((True,s),p) ∈ (set d')}. p
      ; w2 = 1 − w1
      ; d1 = reductionDist ([(s,p/w1). ((True,s),p) ← d'])
      ; d2 = reductionDist ([(s,p/w1). ((False,s),p) ← d'])
      ; h1 = t d1
      ; h2 = e d2
      in (
          if d2 = [] then h1
                     else ( if d1 = [] then h2
                                       else join (prob_choice w1 h1 h2)
                          )
        )

definition
  hobsem :: (('s::linorder) ⇒ ('o::linorder) dist) ⇒ 's prog
where
  hobsem f ≡ multiply ∘ (toPair ∘ (λd. reductionBind d ( obsem f)))

definition
  obsem :: (('a::linorder) ⇒ ('o::linorder) dist) ⇒ ('a ⇒ ('o × 'a) dist)
where
  obsem f' x ≡ dist_fmap (λw. (w,x)) (f' x)

definition
  toPair :: (('o::linorder) × ('s::linorder)) dist ⇒ ('o dist × ('o ⇒ 's dist))
where
  toPair dp ≡ (dist_fmap fst dp,
               λws. let dpws = reductionDist [(s,p). ((ws',s),p) ← dp, ws' = ws]
                      in
                               reductionDist [(s,p/(weight dpws)). (s,p) ← dpws]

               )

definition
  multiply :: ('o dist  × ('o ⇒ 's dist)) ⇒ ('s::linorder) hyper
where
  multiply df ≡ dist_fmap (snd df) (fst df)
```

5 Functor and Monad Laws

Now that we have a monadic definition for our language, we want to ensure that the structures actually behave the way we expect.

Beginning with functors we have two laws, the identity law:

$$\texttt{fmap id} = \texttt{id}$$

Which is saying that if we map the identity over a functor we must get the functor back unchanged. The second functor law is that functor composition is associative:

$$\texttt{fmap (f} \circ \texttt{g)} = \texttt{fmap f} \circ \texttt{fmap g}$$

We present the first law in Isabelle/HOL as:

```
lemma functor_law_id:
dist_fmap (λx. x) dx = dx
```

This is discharged by simplifying with a few relevant definitions (see appendix).

The second law becomes:

```
lemma functor_law_comp_assoc:
  fixes g :: 'a ⇒ ('b::linorder)
    and f :: 'b ⇒ ('c::linorder)
  shows dist_fmap (f ∘ g) = (dist_fmap f) ∘ (dist-fmap g)
```

This is discharged by unfolding the definitions, using extensionality and induction (see appendix).

Next we address the three monad laws. The first is the left identity and it says that the result of `return x` fed into `f` is the same as `f x`. In Isabelle/HOL this is:

```
lemma monad_left_id:
  return x ≫= f = f x
```

This is discharged by unfolding the definitions and simplifying.

The second monad law says that binding on a `return` has no effect:

```
lemma monad_right_id:
  m ≫= return = m
```

This is discharged by unfolding the definitions, inducting over m and simplifying.

The third and final monad law is more complex than the previous two. It is an associative law:

```
lemma monad_assoc:
  fixes f :: 'a ⇒ ('b::linorder) dist
    and g :: 'b ⇒ ('c::linorder) dist
    and m :: 'a dist
  shows  (m ⤜ f) ⤜ g = m ⤜ (λx. (f x) ⤜ g)
```

We see that it is almost just shifting the brackets from `m` and `f` to `f` and `g`, except the lambda is introduced to make sure it type checks. This final law was much more challenging to prove, but after performing induction over `m` and creating two sub-lemmas this was proven (see appendix).

6 Future Work

Now that we have proof that our Kuifje construction obeys the monad and functor laws, the next steps would be to prove algebraic lemmas as we did for Shadow. A good place to start would be the encryption lemma, which needs a slight modification to move it from a non-deterministic statement to a probabilistic one.

After this the goal would be to use these proofs to create a machine-checked environment in which a user could manipulate Kuifje and Shadow programs algebraically, without the need to wrestle with the semantics and with the added benefit of automated reasoning.

References

1. C. Morgan, "The shadow knows: Refinement and security in sequential programs," *Science of Computer Programming*, vol. 74, no. 8, pp. 629–653, 2009.
2. J. Gibbons, A. McIver, T. Schrijvers, and C. Morgan, "Quantitative information flow with monads in haskell," in *Foundations of Probabilistic Programming*, G. Barthe, J.-P. Katoen, and A. Silva, Eds. Cambridge University Press, 2020.
3. T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2003, vol. 2283.
4. G. Smith, "On the foundations of quantitative information flow," in *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2009, pp. 288–302.
5. A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.
6. D. E. R. Denning, *Cryptography and data security*. Addison-Wesley, 1982, vol. 112.
7. J. W. Gray III, "Toward a mathematical foundation for information flow security," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 255–294, 1992.
8. J. K. Millen, "Covert channel capacity," in *1987 IEEE Symposium on Security and Privacy*. IEEE, 1987, pp. 60–60.
9. D. Clark, S. Hunt, and P. Malacaria, "Quantified interference for a while language," *Electronic Notes in Theoretical Computer Science*, vol. 112, pp. 149–166, 2005.

10. S. A. Mario, K. Chatzikokolakis, C. Palamidessi, and G. Smith, "Measuring information leakage using generalized gain functions," in *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 2012, pp. 265–279.
11. A. McIver, C. Morgan, and T. Rabehaja, "Abstract hidden markov models: a monadic account of quantitative information flow," in *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 2015, pp. 597–608.
12. A. McIver, L. Meinicke, and C. Morgan, "Compositional closure for bayes risk in probabilistic noninterference," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2010, pp. 223–235.
13. A. McIver, C. Morgan, G. Smith, B. Espinoza, and L. Meinicke, "Abstract channels and their robust information-leakage ordering," in *International Conference on Principles of Security and Trust*. Springer, 2014, pp. 83–102.
14. E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, E.-U. Informaticien, and E. W. Dijkstra, *A discipline of programming*. prentice-hall Englewood Cliffs, 1976, vol. 1.
15. D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of cryptology*, vol. 1, no. 1, pp. 65–75, 1988.
16. A. McIver and C. Morgan, "The thousand-and-one cryptographers," *Engineering Secure and Dependable Software Systems*, vol. 53, p. 137, 2019.
17. G. Hutton, "A tutorial on the universality and expressiveness of fold," *Journal of Functional Programming*, vol. 9, no. 4, pp. 355–372, 1999.
18. R. Hinze, T. Harper, and D. W. James, "Theory and practice of fusion," in *Symposium on Implementation and Application of Functional Languages*. Springer, 2010, pp. 19–37.
19. G. Malcolm, "Data structures and program transformation," *Science of computer programming*, vol. 14, no. 2-3, pp. 255–279, 1990.