Noninterference through Bisimulation

April Tune, Wendy Yang, and G. A. Kavvos

School of Computer Science, University of Bristol, United Kingdom

Abstract. Noninterference properties state that data does not flow in an undesirable manner, e.g. from a high-security to a low-security setting. Within programming languages noninterference is often enforced through the use of modal type systems. Proving the property often requires non-trivial techniques, such as denotational semantics or logical relations. We show that a simple bisimulation technique (due to Choudhury, Eades, and Weirich) can be adapted to show noninterference for a wide variety of modal type systems.

1 Introduction

This paper aims to showcase a simple technique for proving *information flow properties* of programming languages.

Information flow is a well-studied concept in computer science at large [10]. In programming languages information flow can be readily controlled using type systems—in particular *modal type systems*. Such type systems are sometimes used to separate data into classified and unclassified; in this case, ensuring secure information flow implies an important security property, viz. that non-classified data cannot leak any information about classified data. Information flow properties can also be used to ensure referential transparency, e.g. in programming calculi with effects [28].

Information flow properties are usually proven by analysing the semantics of a language. One approach goes through using a *denotational semantics* that encodes information flow [1, 21]. This entails a large amount of mathematical overhead, but it exposes the mathematical and conceptual reasons that enforce noninterference. A more common approach uses the technique of *logical relations* [36, 31, 16]. Finally, a third approach proceeds by an adequate translation to a parametric calculus [37], such as the simply-typed λ-calculus [33], System F_ω [6], or the Calculus of Constructions [2].

In this paper we show that a much simpler, bisimulation technique, which is due to Choudhury, Eades, and Weirich [7], can be adapted to show noninterference for a wide range of modal type systems. This makes it just as powerful as denotational techniques [21] while also being simple—much simpler than both logical relations and parametricity translations, which require tricky syntactic lemmas. Because of its syntactic nature it is possible that this technique will struggle to prove properties of languages that are drastically more complex, e.g. languages that feature higher-order state [31, 16]. Still, this technique is effective and simple, while covering all but the most complex of language features.

To substantiate this argument we prove two noninterference results for two small programming languages with modal types, namely Moggi's *monadic metalanguage* [28] and Clouston's *Fitch-style modal λ-calculus* [8]. These two languages are very different. The former is the minimal typed λ-calculus with a modal type constructor $T(A)$, corresponding to a strong monad. The latter is the simplest possible λ-calculus with a type constructor $\Box A$, corresponding to the minimal modal logic K under the Curry-Howard isomorphism. Categorically, it amounts to a right adjoint. While a noninterference theorem has been shown before for the monadic metalanguage [21], a noninterference result for the Fitch-style λ-calculus is presented here for the first time. Remarkably, the bisimulation technique applies to both with equal ease.

To show the power of the technique we augment both of these languages with (equi-)*recursive types*. The point is that the various techniques in the literature *cannot* easily handle this extension. The denotational models can be made domain-theoretic [1], but solving recursive domain equations over them appears non-trivial. The logical relations technique is difficult to adapt to recursive types, as it is not easy to argue that the required relation exists—due to the fact they are no longer given by induction on types. One solution is to use Pitts' *minimal invariants* to show that necessarily relations exist [30]. Another is to use *step-indexing* [4], perhaps by constructing the the logical relation within a logic with guarded recursion Dreyer, Ahmed, and Birkedal [11]. The latter technique has recently been used to prove a noninterference result [16]. Both of these solutions are relatively sophisticated.

In contrast, the bisimulation technique we demonstrate here requires none of that sophistication, and works with recursive types directly out of the box. the simplicity of this technique means that is readily amenable to formalisation in a proof assistant, requiring neither the baggage of denotational models nor the issues with defining logical relations. To substantiate this we provide a formalisation of our results in Agda.

2 The Monadic Metalanguage

The programming languages that are closest to mathematics, such as Haskell, are often called *purely functional*. It is difficult to precisely define what purity is; some researchers identify it with *referential transparency* [35, §3.2.1], which itself has many explications: the value of a program is determined by the value of its parts; calling a function on the same input always produces the same output; any term can be replaced by its value without changing the overall outcome. Of course, most programming languages are *not* of this sort, as they allow *effects*, such as a mutable store, nondeterminism, and exceptions.

Moggi [28] proposed that the semantics of languages with effects can be unified using the categorical abstraction of a *strong monad*. It is possible to present how this abstraction works entirely type-theoretically. Assume that we have a special type constructor $T(-)$. If A is a type of values, then $T(A)$ is the type of *computations* that return a value of type A. Only terms of $T(-)$ type are

allowed to have effects. This allows us to encapsulate effects under a particular kind of type, keeping the rest of the language pure.

In this setting it is reasonable to have a function $\texttt{return} : A \to T(A)$ that maps a value $a : A$ to a 'pure' computation $\texttt{return}(a) : T(A)$ which performs no effects and returns $a : A$ immediately. However, it would be ill-advised to have a function $T(A) \to A$ that extracts the value returned by a computation. The reason is that this value could depend on effects, thereby causing impurity.

How can we then use the value returned by a computation $M : T(A)$? The answer is that we can bind this value to a free variable $x : A$, but only in a term $x : A \vdash N : T(B)$ which is also of $T(-)$ type. Thus, values produced by a potentially side-effectful computation can only influence terms of $T(-)$ type, i.e. other side-effectful computations. Under these conditions we can form a *sequencing* term

$$\texttt{let}\ x \Leftarrow M\ \texttt{in}\ N : T(B)$$

This program first runs the computation M, performing all its side-effects and obtaining a value V. It then substitutes this value into N to obtain $N[V/x]$, and continues by running that computation. The simply-typed λ-calculus equipped with these two primitives is known as the *monadic metalanguage* [28]. However, similar techniques can also be used as a kind of design pattern that models effectful computation within a purely functional language [5].

The discussion above has a strong flavour of information flow: the type constructor $T(-)$ is a *modality* that isolates a region of the language. Data can flow from the *pure region* into the *impure region* ($A \to T(A)$), but never backwards. We may call this the *Hotel California invariant*: *'you can check out any time you like, but you can never leave.'*

However, all we have done is give some intuitions and some design criteria. Our ultimate objective is to *prove* that the language with these typing rules abides by the desired information flow restrictions. The key property in this setting is that a term $c : T(A) \vdash M : \texttt{Bool}$ of base type with a free computation variable $c : T(A)$ should evalute the same v alue, irrespective of which computation $M : T(A)$ we might substitute for c. More rigorously, the requisite noninterference property is that, for any closed $C_1, C_2 : T(A)$, the term $M[C_1/x]$ evaluates to the same value as the term $M[C_2/x]$.

A denotational proof of this fact was given by Kavvos [21], based on the model of Abadi *et al.* [1]. In this section we will present a much simpler, bisimulation-based proof.

2.1 Syntax and semantics

The syntax of the monadic metalanguage is given in Fig. 1. This version of the metalanguage also includes *recursive types*, in the style of the fixed point calculus **FPC** [17, §7.4] [18, §20]. Such an extension of the monadic metalanguage has been previously considered by Filinski [12], who shows that it admits a straightforward domain-theoretic semantics.

Fig. 1: Types and terms of the monadic metalanguage.

$$\textsc{MM/Type/Var}\ \frac{}{\Delta, \alpha, \Delta' \vdash \alpha\ \mathsf{type}} \qquad \textsc{MM/Type/UnitT}\ \frac{}{\Delta \vdash \mathbf{1}\ \mathsf{type}} \qquad \textsc{MM/Type/Prod}\ \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A \times B\ \mathsf{type}}$$

$$\textsc{MM/Type/Sum}\ \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A + B\ \mathsf{type}} \qquad \textsc{MM/Type/Fun}\ \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A \to B\ \mathsf{type}}$$

$$\textsc{MM/Type/Rec}\ \frac{\Delta, \alpha \vdash A\ \mathsf{type}}{\Delta \vdash \mathsf{rec}\ \alpha.\ A\ \mathsf{type}} \qquad \textsc{MM/Type/Monad}\ \frac{\Delta \vdash A\ \mathsf{type}}{\Delta \vdash TA\ \mathsf{type}}$$

$$\textsc{MM/Ctx/Empty}\ \frac{}{\vdash \cdot\ \mathsf{ctx}} \qquad \textsc{MM/Ctx/Extend}\ \frac{\vdash \Gamma\ \mathsf{ctx} \qquad \vdash A\ \mathsf{type}}{\vdash \Gamma, x : A\ \mathsf{ctx}}$$

$$\textsc{MM/Term/Var}\ \frac{\vdash \Gamma, x : A, \Gamma'\ \mathsf{ctx}}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \textsc{MM/Term/Unit}\ \frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \langle\rangle : \mathbf{1}} \qquad \textsc{MM/Term/In}\ \frac{\Gamma \vdash M : A_i}{\Gamma \vdash \mathsf{in}_i(M) : A_1 + A_2}$$

$$\textsc{MM/Term/Case}\ \frac{\Gamma \vdash M : A_1 + A_2 \qquad \Gamma, x : A_1 \vdash P : B \qquad \Gamma, y : A_2 \vdash Q : B}{\Gamma \vdash \mathsf{case}(M; x.\ P; y.\ Q) : B}$$

$$\textsc{MM/Term/Tuple}\ \frac{\Gamma \vdash M : A_1 \qquad \Gamma \vdash N : A_2}{\Gamma \vdash \langle M, N\rangle : A_1 \times A_2} \qquad \textsc{MM/Term/Proj}\ \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i(M) : A_i} \qquad \textsc{MM/Term/Lam}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\ M : A \to B}$$

$$\textsc{MM/Term/App}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B} \qquad \textsc{MM/Term/Return}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{return}(M) : TA}$$

$$\textsc{MM/Term/Bind}\ \frac{\Gamma \vdash M : TA \qquad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : TB} \qquad \textsc{MM/Term/Fold}\ \frac{\Gamma \vdash M : A[\mathsf{rec}\ \alpha.\ A/\alpha]}{\Gamma \vdash \mathsf{fold}(M) : \mathsf{rec}\ \alpha.\ A}$$

$$\textsc{MM/Term/Unfold}\ \frac{\Gamma \vdash M : \mathsf{rec}\ \alpha.\ A}{\Gamma \vdash \mathsf{unfold}(M) : A[\mathsf{rec}\ \alpha.\ A/\alpha]}$$

The rules of the calculus itself are given in Fig. 1. Types are constructed by the typing judgement $\Delta \vdash A\ \mathtt{type}$, where $\Delta = \alpha_1, \ldots, \alpha_n$ is a list of *type variables*. The meaning of this judgement is that A is a type that may use the free type variables in Δ. The available type constructors include sums, products, monadic types, and the recursive type $\mu\alpha.A$. The recursive type is meant to be isomorphic to its *unfolding* $A[\mu\alpha.A/\alpha]$. A *closed type* is a type A such that $\Delta \vdash A\ \mathtt{type}$. Terms over a context are given using the typing judgement $\Gamma \vdash M : A$. Contexts are of the form $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, where all the A_i are limited to closed types. The typing rules combine the usual rules for recursive types with the term constructors for the monadic metalanguage.

The combination of products, sums, and recursive types is rather expressive. For example, recursive types and function types suffice to define terms by recursion, in the style of PCF—see *loc. cit.* Furthermore, we can define a type of Booleans by defining the shorthands

$$\mathtt{Bool} \stackrel{\text{def}}{=} \mathbf{1} + \mathbf{1} \qquad \mathtt{true} \stackrel{\text{def}}{=} \mathtt{in}_1(\langle\rangle) \qquad \mathtt{false} \stackrel{\text{def}}{=} \mathtt{in}_2(\langle\rangle)$$
$$\mathtt{if}\ M\ \mathtt{then}\ P\ \mathtt{else}\ Q \stackrel{\text{def}}{=} \mathtt{case}(M; \_.\, P; \_.\, Q)$$

where _ is any fresh variable. The 'lazy' natural numbers can be defined in a similar manner [18, §20.2].

A small-step operational semantics is given in Fig. 2. This is essentially the standard small-step semantics of recursive types [18, §20.1], extended with the rule MM/Opsem/Bind for the primitives of the monadic metalanguage. It evidently satisfies the expected progress and preservation results, and it is deterministic. Letting $\longmapsto^*$ denote the reflexive transitive closure, we define

$$M \Downarrow V \stackrel{\text{def}}{\equiv} M \longmapsto^* V \text{ and } V\ \mathtt{val}$$

to mean that a term M evaluates to a (necessarily unique) value V $\mathtt{val}$.

2.2 Indistinguishability

We now define a typed *indistinguishability relation*

$$\Gamma \vdash M \sim N : A$$

for terms of type A in context Γ. The meaning of this relation is that the terms M and N cannot be distinguished by an *unprivileged observer*. The meaning of 'privilege' will change depending on the information flow property of interest. In the monadic metalanguage, to be privileged is to be impure, i.e. to have access to the values returned by effectful computations. Thus, an unprivileged observer is one who observes only the pure regions of the calculus.

These intuitions are reflected in Fig. 3, which gives the clauses defining the indistinguishability relation. The majority of the clauses are *congruence rules*, which ensure that the relation is preserved under all constructs of the language. The only non-trivial rule is MM/Sim/Prot which ensures that $M \sim N : TA$ for

Fig. 2: Small-step operational semantics for the monadic metalanguage.

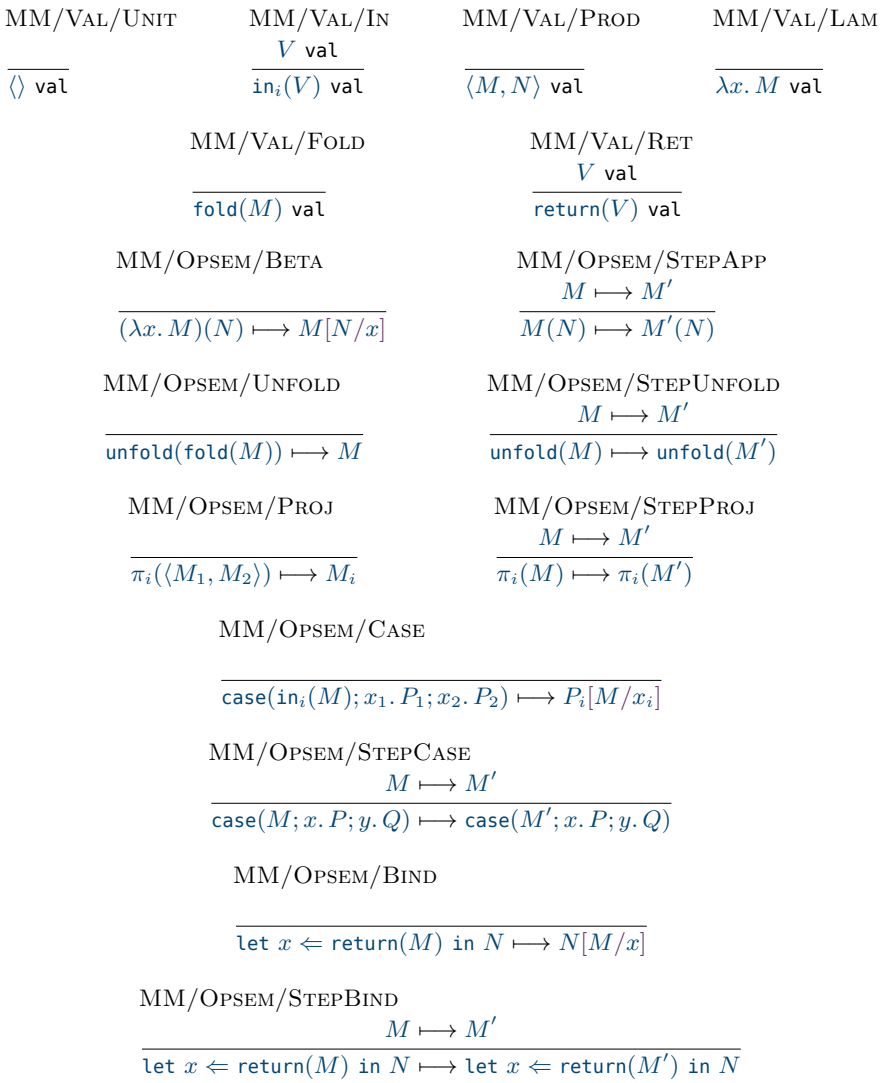


Fig. 3: Indistinguishability relation $\sim$ for the monadic metalanguage.

$$\textsc{MM/Sim/Prot}\quad \frac{\Gamma \vdash M : TA \qquad \Gamma \vdash M' : TA}{\Gamma \vdash M \sim M' : TA}$$

$$\textsc{MM/Sim/Var}\quad \frac{}{\Gamma, x : A, \Gamma' \vdash x \sim x : A}$$

$$\textsc{MM/Sim/App}\quad \frac{\Gamma \vdash M \sim M' : A \to B \qquad \Gamma \vdash N \sim N' : A}{\Gamma \vdash M(N) \sim M'(N') : B}$$

$$\textsc{MM/Sim/Lam}\quad \frac{\Gamma, x : A \vdash M \sim M' : B}{\Gamma \vdash \lambda x.\, M \sim \lambda x.\, M' : A \to B}$$

$$\textsc{MM/Sim/Unit}\quad \frac{}{\Gamma \vdash \langle\rangle \sim \langle\rangle : \mathbf{1}}$$

$$\textsc{MM/Sim/In}\quad \frac{\Gamma \vdash M \sim M' : A_i}{\Gamma \vdash \mathsf{in}_i(M) \sim \mathsf{in}_i(M') : A_1 + A_2}$$

$$\textsc{MM/Sim/Case}\quad \frac{\Gamma \vdash M \sim M' : A_1 + A_2 \qquad \Gamma, x : A_1 \vdash P \sim P' : B \qquad \Gamma, y : A_2 \vdash Q \sim Q' : B}{\Gamma \vdash \mathsf{case}(M; x.\, P; y.\, Q) \sim \mathsf{case}(M'; x.\, P'; y.\, Q') :}$$

$$\textsc{MM/Sim/Proj}\quad \frac{\Gamma \vdash M \sim M' : A_1 \times A_2}{\Gamma \vdash \pi_i(M) \sim \pi_i(M') : A_i}$$

$$\textsc{MM/Sim/Tuple}\quad \frac{\Gamma \vdash M \sim M' : A_1 \qquad \Gamma \vdash N \sim N' : A_2}{\Gamma \vdash \langle M, N\rangle \sim \langle M', N'\rangle : A_1 \times A_2}$$

$$\textsc{MM/Sim/Fold}\quad \frac{\Gamma \vdash M \sim M' : A[\mathsf{rec}\ \alpha.\ A/\alpha]}{\Gamma \vdash \mathsf{fold}(M) \sim \mathsf{fold}(M') : \mathsf{rec}\ \alpha.\ A}$$

$$\textsc{MM/Sim/Unfold}\quad \frac{\Gamma \vdash M \sim M' : \mathsf{rec}\ \alpha.\ A}{\Gamma \vdash \mathsf{fold}(M) \sim \mathsf{fold}(M') : A[\mathsf{rec}\ \alpha.\ A/\alpha]}$$

any two terms of monadic type. Thus, an observer that cannot distinguish terms up to $\sim$ sees pure terms as they are, but impure terms as opaque and cannot be distinguished. Finally, notice that this relation does not include any clauses relate terms up to computation (e.g. β-convertibility). It is easy to show that it is also reflexive:

Lemma 1. *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash M \sim M : A$.

The interesting property of this relation is that it is a *bisimulation* [32] for the small-step operational semantics.

Definition 1 (Bisimulation). *A relation R is a bisimulation with respect to a transition system $\longmapsto$ just if the following conditions hold:*

1. *If xRy and $x \longmapsto x'$, then there exists a y' such that $y \longmapsto y'$ and $x'Ry'$.*
2. *If xRy and $y \longmapsto y'$, then there exists an x' such that $x \longmapsto x'$ and $x'Ry'$.*

This captures the idea that related terms must evaluate in 'lock-step,' with the relation being preserved by every step of reduction. Proving that $\sim$ is a bisimulation is relatively simple, but requires two additional lemmata.

Lemma 2. $\Gamma \vdash t_1 \sim t_2 : A$ *implies* $\Gamma \vdash t_1 : A$ *and* $\Gamma \vdash t_2 : A$.

A (simultaneous) *substitution* is a partial function γ from variables to terms. We write $\gamma(x) \downarrow$ to mean that the substitution is defined for variable x, and $M[\gamma]$ for the resulting from applying the substitution γ to M. The definition is the usual structural one, with base case $(x)[\gamma] \simeq \gamma(x)$ (if the right hand side is defined). We write $\gamma : \Delta \to \Gamma$ whenever γ is a substitution such that

$$\forall (x : A) \in \Gamma.\ \gamma(x) \downarrow \wedge\ \Gamma \vdash \gamma(x) : A$$

The substitution lemma then amounts to the fact the following rule is admissible.

$$\frac{\Gamma \vdash M : A \qquad \gamma : \Delta \to \Gamma}{\Delta \vdash M[\gamma] : A}$$

We may similarly extend the definition of $\sim$ to substitutions:

$$\Delta \vdash \gamma_1 \sim \gamma_2 : \Gamma \stackrel{\text{def}}{\equiv} \forall (x : A) \in \Gamma.\ \gamma(x) \downarrow\ \wedge\ \delta(x) \downarrow\ \wedge\ \Gamma \vdash \gamma(x) \sim \delta(x) : A$$

We can then prove that

Lemma 3. *The following rule is admissible:*

$$\frac{\Delta \vdash \gamma_1 \sim \gamma_2 : \Gamma \qquad \Gamma \vdash M_1 \sim M_2 : A}{\Gamma \vdash M_1[\gamma_1] \sim M_2[\gamma_2] : A}$$

Proof. By induction on the evidence that $\Gamma \vdash M_1 \sim M_2 : A$. Note that if this holds via MM/Sim/Prot, then $M_i[\gamma_i]$ will also have a type TA and so the conclusion holds by the same rule.

We can now show that $\sim$ is a bisimulation for $\longmapsto$ on *non-monadic types*, i.e. types that are not of the form $T(-)$.

Theorem 1 (Bisimulation I). *If A is not a monadic type, $\Gamma \vdash M \sim N : A$ and $M \longmapsto M'$, then there exists a term $\Gamma \vdash N' : A$ such that $N \longmapsto N'$ and $\Gamma \vdash M' \sim N' : A$.*

Proof. By induction on the evidence that $M \longmapsto M'$. The cases for return and bind can immediately be discharged, as any reduction involving these terms must have monadic type. The other of β-reduction is a simple application of Lemma 3. All other cases are simple congruences.

One subtle point is that the result *does not hold* over monadic types: being unable to peer into the structure of their terms, the unprivileged observer cannot guarantee that impure terms reduce in lockstep. As a simple example, consider the two indistinguishable terms $(\lambda x.\, x)(\mathtt{return}(M))$ and $\mathtt{return}(M)$ for some closed M. Clearly the left term can reduce, but we will be unable to show the same for the right term. However, this is no skin off our backs, as this restricted form of the lemma is all we need to prove for our main result:

Theorem 2 (Noninterference I). *If $x : TA \vdash M : \mathtt{Bool}$ and $\vdash N_1, N_2 : TA$ then $M[N_1/x] \Downarrow V$ if and only if $M[N_2/x] \Downarrow V$.*

Proof. We have that $\vdash N_1 \sim N_2 : T(A)$ via MM/Sim/Prot. Hence, by Lemmas 1 and 3 we have that $\vdash M[N_1/x] \sim M[N_2/x] : \mathtt{Bool}$.

If $M[N_1/x] \Downarrow V$ we obtain a sequence of reductions

$$M[N_1/x] = P_0 \longmapsto \ \ldots \ \longmapsto P_k = V$$

Applying Theorem 1 to every step of this reduction we obtain a sequence of reductions

$$M[N_2/x] = P'_0 \longmapsto \ \ldots \ \longmapsto P'_k$$

with $\vdash P_i \sim P'_i$: for every $0 \leq i \leq k$. Hence $\vdash V \sim P'_k : \mathtt{Bool}$. But the only rule of Fig. 3 that could apply to the closed value V of $\mathtt{Bool} = \mathbf{1}+\mathbf{1}$ type is MM/Sim/In, followed by MM/Sim/Unit. Hence $V = \mathtt{in}_i(\langle\rangle) = P'_k$ for some $i \in \{1, 2\}$. Hence $M[N_2/x] \Downarrow V$. The other direction is entirely symmetric.

3 Fitch-style λ-calculus

Another class of calculi with modal types are the *Fitch-style calculi* [8]. These calculi have a type former $\Box A$, which has its origins in the Curry-Howard isomorphism for modal logic.

It is informally understood that the use of these modalities enforces some type of information flow control [21]. For modalities of $\Box$-type the resultant information control property says that *modal data may only depend on modal data*. This invariant is especially useful in settings that model intensionality (where source code must only depend on source code) [20, 24, 23], globality

(where globally-available data must only depend on globally-available resources) [25], or metaprogramming (where constructed programs may only depend on statically-available data, and not dynamically-computed data) [9, 19].

However, most of the aforementioned applications use a calculus based on the dual-context syntax of Davies and Pfenning [9, 29, 22]. Instead, more recent work on modal type theory increasingly relies on the Fitch-style calculus of Clouston [8] [14, 15, 13]. While noninterference theorems have indeed been shown for variants of the dual-context systems [27, 21], no noninterference results have been shown for the Fitch-style calculus so far. In this section we show that the bisimulation proof of [7] directly adapts to yield one.

3.1 Syntax and semantics

While the monadic metalanguage was a simple variation on the simply-typed λ-calculus, the syntax of modal λ-calculi with a $\Box$-style modality is more complicated. The reason is that the monadic metalanguage models a *strong* monad, which makes the $T(-)$ type constructor pleasantly interact with the context. This is not so for calculi with a $\Box$ modality: expressing such modalities requires fundamental changes to the judgemental structure of the context [34, §4.1] [13].

λ-calculi with a $\Box$ modality have a type $\Box A$ for every type A. However, unlike the monadic metalanguage, the information flow invariant here is more restrictive *data should not flow either into or out of a box*. Thus, it must not be possible to define neither a function of type $A \to \Box A$ nor a function of type $\Box A \to A$. In a sense, the $\Box$ separates the language into two strata: the data 'within the box,' and the data 'outside the box.'

To prohibit the definition of a function $A \to \Box A$ we must have a way to *protect* the context so that data cannot flow into a boxed type. This is achieved by introducing a *lock* ($🔒$) on contexts. This is an operation that forbids access to variables to its left. This limitation is encoded by the variable rule:

$$\frac{🔒 \notin \Delta}{\Gamma, x : A, \Delta \vdash x : A}$$

Locks enable us to *introduce* a term of boxed type: if we have a term $\Gamma, 🔒 \vdash M : A$ in a locked context, then we can promote it to a term of boxed type:

$$\frac{\Gamma, 🔒 \vdash M : A}{\Gamma \vdash \mathsf{box}(M) : \Box A}$$

It might appear that the lock prohibits access to *all* variables behind it. This is not true: the *elimination* rule for terms of boxed type allows us to 'unbox' a term of boxed type, at the price of introducing another lock in the context:[1]

$$\frac{\Gamma \vdash M : \Box A}{\Gamma, 🔒, \Delta \vdash \mathsf{unbox}(M) : A}$$

[1] The additional variables in Δ are required for weakening to be admissible.

Thus, if a variable is behind a lock, it can only be used if it is of modal type. This maintains the information flow invariant: things in the box can only be used within the box, while the box remains sealed. Inspired by the relevant piece of laboratory equipment, we may call this the *glovebox invariant*.

We can hence write like

$$\vdash \lambda f.\, \lambda x.\, \mathtt{box}(\mathtt{unbox}(f)(\mathtt{unbox}(x))) : \Box(A \to B) \to \Box A \to \Box B$$

which applies a function $A \to B$ (within the glovebox) to an argument A (within the glovebox) to obtain a result B (within the glovebox).

The syntax of the Fitch-style λ-calculus with recursive types is given in Fig. 4. The type judgement $\Delta \vdash A\ \mathtt{type}$ is similar to that of the monadic metalanguage, but replaces $T(-)$ with $\Box(-)$. The context judgement $\vdash \Gamma\ \mathtt{ctx}$ allows extending the context in two ways: by binding a variable x to a closed type $\vdash A\ \mathtt{type}$ (Fitch/Ctx/Extend); or by adding a new lock (Fitch/Ctx/Lock). The term typing rules are similar to the simply-typed λ-calculus and **FPC**, with the exception the variable rule (Fitch/Term/App) and the modal rules that we discussed above (Fitch/Term/Box, Fitch/Term/Unbox).

The small-step semantics is presented in Fig. 5. As before, this is the usual small-step semantics for recursive types, with the addition of the reduction $\mathtt{unbox}(\mathtt{box}(M)) \longmapsto M$. Note that terms of the form $\mathtt{box}(M)$ are values (by Fitch/Val/Prod). The usual progress and preservation theorems are satisfied.

3.2 Indistinguishability

Much like before, we define an indistinguishability relation in Fig. 6. In this setting the observer is *outside* the box, and cannot see inside.

As for the metalanguage, the relation will consist almost entirely of congruence rules. The exception of terms which have modal type, which will always be related. Thus, all $\mathtt{box}(-)$ terms are related (Fitch/Term/Box). We do *not* have a congruence rule for $\mathtt{unbox}(-)$, but instead have a rule that relates all well-typed terms of the form $\mathtt{unbox}(M)$ with each other. Such a rule would be derivable from Fitch/Term/Box and a congruence rule. However, this choice is inessential for the noninterference result, but makes some of the proofs technically easier.

We can then show results analogous to Lemma 3 and Theorem 1. We say that a closed type A is *pure*, denoted $A\ \mathtt{pure}$, just if it does not contain a $\Box$.

Lemma 4. *The following rule is admissible:*

$$\frac{🔒 \notin \Gamma \qquad A\ \mathtt{pure} \qquad \Delta \vdash \gamma_1 \sim \gamma_2 : \Gamma \qquad \Gamma \vdash M_1 \sim M_2 : A}{\Gamma \vdash M_1[\gamma_1] \sim M_2[\gamma_2] : A}$$

Theorem 3 (Bisimulation II). *Let* $🔒 \notin \Gamma$ *and* $A\ \mathtt{pure}$. *If* $\Gamma \vdash M \sim N : A$ *and* $M \longmapsto M'$, *then there exists a term* $\Gamma \vdash N' : A$ *such that* $N \longmapsto N'$ *and* $\Gamma \vdash M' \sim N' : A$.

Fig. 4: Types and terms of the Fitch calculus.

$$\textsc{Fitch/Type/Var}\quad \frac{}{\Delta, \alpha, \Delta' \vdash \alpha\ \mathsf{type}} \qquad \textsc{Fitch/Type/UnitT}\quad \frac{}{\Delta \vdash \mathbf{1}\ \mathsf{type}} \qquad \textsc{Fitch/Type/Prod}\quad \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A \times B\ \mathsf{type}}$$

$$\textsc{Fitch/Type/Sum}\quad \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A + B\ \mathsf{type}} \qquad \textsc{Fitch/Type/Fun}\quad \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A \to B\ \mathsf{type}}$$

$$\textsc{Fitch/Type/Rec}\quad \frac{\Delta, \alpha \vdash A\ \mathsf{type}}{\Delta \vdash \mathsf{rec}\ \alpha.\ A\ \mathsf{type}} \qquad \textsc{Fitch/Type/Box}\quad \frac{\Delta \vdash A\ \mathsf{type}}{\Delta \vdash \Box A\ \mathsf{type}}$$

$$\textsc{Fitch/Ctx/Empty}\quad \frac{}{\vdash \cdot\ \mathsf{ctx}} \qquad \textsc{Fitch/Ctx/Extend}\quad \frac{\vdash \Gamma\ \mathsf{ctx} \qquad \vdash A\ \mathsf{type}}{\vdash \Gamma, x : A\ \mathsf{ctx}} \qquad \textsc{Fitch/Ctx/Lock}\quad \frac{\vdash \Gamma\ \mathsf{ctx}}{\vdash \Gamma, 🔒\ \mathsf{ctx}}$$

$$\textsc{Fitch/Term/Var}\quad \frac{\vdash \Gamma, x : A, \Gamma'\ \mathsf{ctx} \qquad 🔒 \notin \Gamma'}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \textsc{Fitch/Term/Unit}\quad \frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \langle\rangle : \mathbf{1}} \qquad \textsc{Fitch/Term/In}\quad \frac{\Gamma \vdash M : A_i}{\Gamma \vdash \mathsf{in}_i(M) : A_1 + A_2}$$

$$\textsc{Fitch/Term/Case}\quad \frac{\Gamma \vdash M : A_1 + A_2 \qquad \Gamma, x : A_1 \vdash P : B \qquad \Gamma, y : A_2 \vdash Q, y : A_2 : B}{\Gamma \vdash \mathsf{case}(M; x.\ P; y.\ Q) : B}$$

$$\textsc{Fitch/Term/Tuple}\quad \frac{\Gamma \vdash M : A_1 \qquad \Gamma \vdash N : A_2}{\Gamma \vdash \langle M, N\rangle : A_1 \times A_2} \qquad \textsc{Fitch/Term/Proj}\quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i(M) : A_i} \qquad \textsc{Fitch/Term/Lam}\quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\ M : A \to B}$$

$$\textsc{Fitch/Term/App}\quad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B} \qquad \textsc{Fitch/Term/Box}\quad \frac{\Gamma, 🔒 \vdash M : A}{\Gamma \vdash \mathsf{box}(M) : \Box A} \qquad \textsc{Fitch/Term/Unbox}\quad \frac{\Gamma \vdash M : \Box A}{\Gamma, 🔒, \Delta \vdash \mathsf{unbox}(M) : A}$$

$$\textsc{Fitch/Term/Fold}\quad \frac{\Gamma \vdash M : A[\mathsf{rec}\ \alpha.\ A/\alpha]}{\Gamma \vdash \mathsf{fold}(M) : \mathsf{rec}\ \alpha.\ A} \qquad \textsc{Fitch/Term/Unfold}\quad \frac{\Gamma \vdash M : \mathsf{rec}\ \alpha.\ A}{\Gamma \vdash \mathsf{unfold}(M) : A[\mathsf{rec}\ \alpha.\ A/\alpha]}$$

Fig. 5: Small-step operational semantics for the Fitch-style calculus.

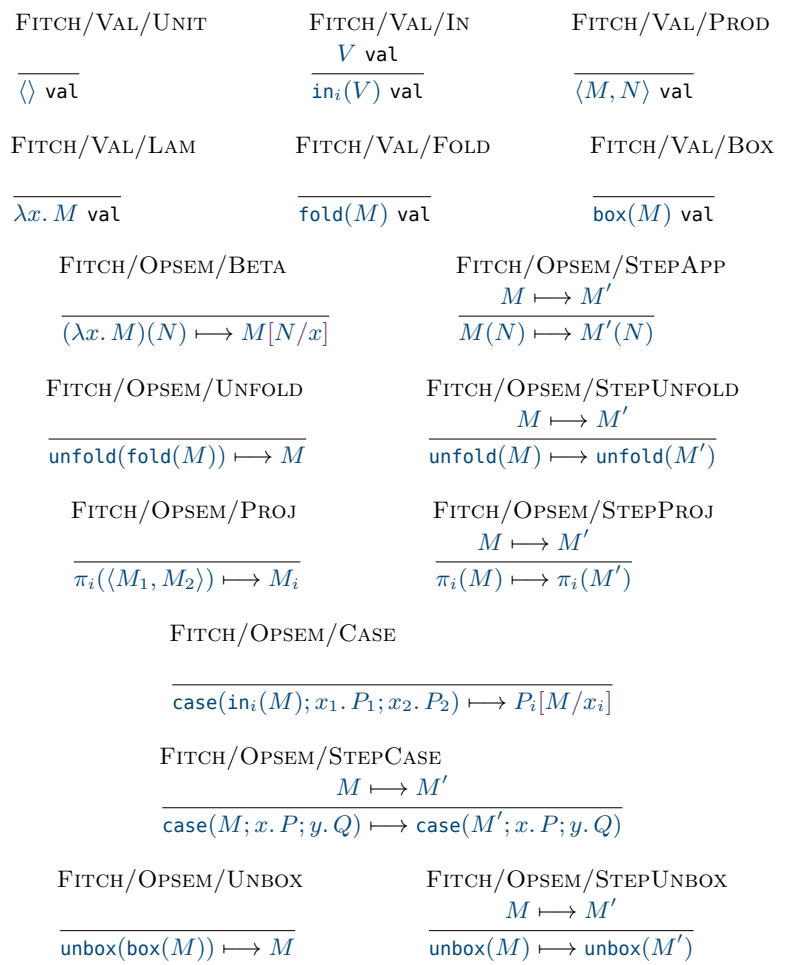


Fig. 6: Indistinguishability relation $\sim$ for the Fitch calculus.

$$\textsc{Fitch/Sim/Prot}\quad \frac{}{\Gamma \vdash M \sim M' : \Box A}$$

$$\textsc{Fitch/Sim/Unbox}\quad \frac{\Gamma \vdash M : \Box A \qquad \Gamma \vdash M' : \Box A}{\Gamma, \locked, \Delta \vdash \mathsf{unbox}(M) \sim \mathsf{unbox}(M') : A}$$

$$\textsc{Fitch/Sim/Var}\quad \frac{\locked \notin \Gamma'}{\Gamma, x : A, \Gamma' \vdash x \sim x : A}$$

$$\textsc{Fitch/Sim/App}\quad \frac{\Gamma \vdash M \sim M' : A \to B \qquad \Gamma \vdash N \sim N' : A}{\Gamma \vdash M(N) \sim M'(N') : B}$$

$$\textsc{Fitch/Sim/Lam}\quad \frac{\Gamma, x : A \vdash M \sim M' : B}{\Gamma \vdash \lambda x.\, M \sim \lambda x.\, M' : A \to B}$$

$$\textsc{Fitch/Sim/Unit}\quad \frac{}{\Gamma \vdash \langle\rangle \sim \langle\rangle : \mathbf{1}}$$

$$\textsc{Fitch/Sim/In}\quad \frac{\Gamma \vdash M \sim M' : A_i}{\Gamma \vdash \mathsf{in}_i(M) \sim \mathsf{in}_i(M') : A_1 + A_2}$$

$$\textsc{Fitch/Sim/Case}\quad \frac{\Gamma \vdash M \sim M' : A_1 + A_2 \qquad \Gamma, x : A_1 \vdash P \sim P' : B \qquad \Gamma, y : A_2 \vdash Q \sim Q' : B}{\Gamma \vdash \mathsf{case}(M; x.\, P; y.\, Q) \sim \mathsf{case}(M'; x.\, P'; y.\, Q') : B}$$

$$\textsc{Fitch/Sim/Proj}\quad \frac{\Gamma \vdash M \sim M' : A_1 \times A_2}{\Gamma \vdash \pi_i(M) \sim \pi_i(M') : A_i}$$

$$\textsc{Fitch/Sim/Tuple}\quad \frac{\Gamma \vdash M \sim M' : A_1 \qquad \Gamma \vdash N \sim N' : A_2}{\Gamma \vdash \langle M, N\rangle \sim \langle M', N'\rangle : A_1 \times A_2}$$

$$\textsc{Fitch/Sim/Fold}\quad \frac{\Gamma \vdash M \sim M' : A[\mathsf{rec}\ \alpha.\ A/\alpha]}{\Gamma \vdash \mathsf{fold}(M) \sim \mathsf{fold}(M') : \mathsf{rec}\ \alpha.\ A}$$

$$\textsc{Fitch/Sim/Unfold}\quad \frac{\Gamma \vdash M \sim M' : \mathsf{rec}\ \alpha.\ A}{\Gamma \vdash \mathsf{unfold}(M) \sim \mathsf{unfold}(M') : A[\mathsf{rec}\ \alpha.\ A/\alpha]}$$

The reason for restricting this result to contexts without a lock is similar to the restriction to non-monadic types in Theorem 1. As all $\mathtt{unbox}(-)$ terms are related we may have a term $\mathtt{unbox}((\lambda x.\, x)(M))$, which may take a reduction step. However, this term is related to $\mathtt{unbox}(M)$, which may not be able to a match this reduction for an appropriate M. However, this situation can only occur in locked contexts, which we ultimately do not need to prove non-interference.

The main theorem is then proved in the same manner as Theorem 2.

Theorem 4 (Noninterference II). *If* $x : \Box A \vdash M : \mathtt{Bool}$ *and* $\vdash N_1, N_2 : \Box A$ *then* $M[N_1/x] \Downarrow V$ *if and only if* $M[N_2/x] \Downarrow V$.

4 Agda formalisation

We have argued that the technique above is simple and adaptable, and hence can be used in the setting of a proof assistant. To substantiate that we have formalised them in Agda. In this section we give a brief overview of the formalisation of the monadic metalanguage. The full code can be found at

`https://github.com/april-pl/modal-agda`

To encode the terms of the monadic metalanguage we use an intrinsically-typed deep embedding, with terms are indexed by both contexts and types [3]. For the monadic metalanguage this definition is given in Fig. 7, and looks essentially identical to our typing rules. The only deviation is that we parameterise the `unfold` constructor, which constructs a term of type `B` when given a proof of equality `B ≡ A [ Rec A ]`. We do this to avoid an instance of the *green slime*, a unification problem which arises when the type of a constructor mentions a defined function (in our case: substitution on types) [26, Principle 1].

We can then directly encode the transition relation (Fig. 8), as well as its indistinguishability relation. As terms are intrinsically typed, the relation must also be indexed by the type and context of the terms it relates; this allows us to project individual terms out of a proof of relation. It is then possible to show that the relation is a bisimulation:

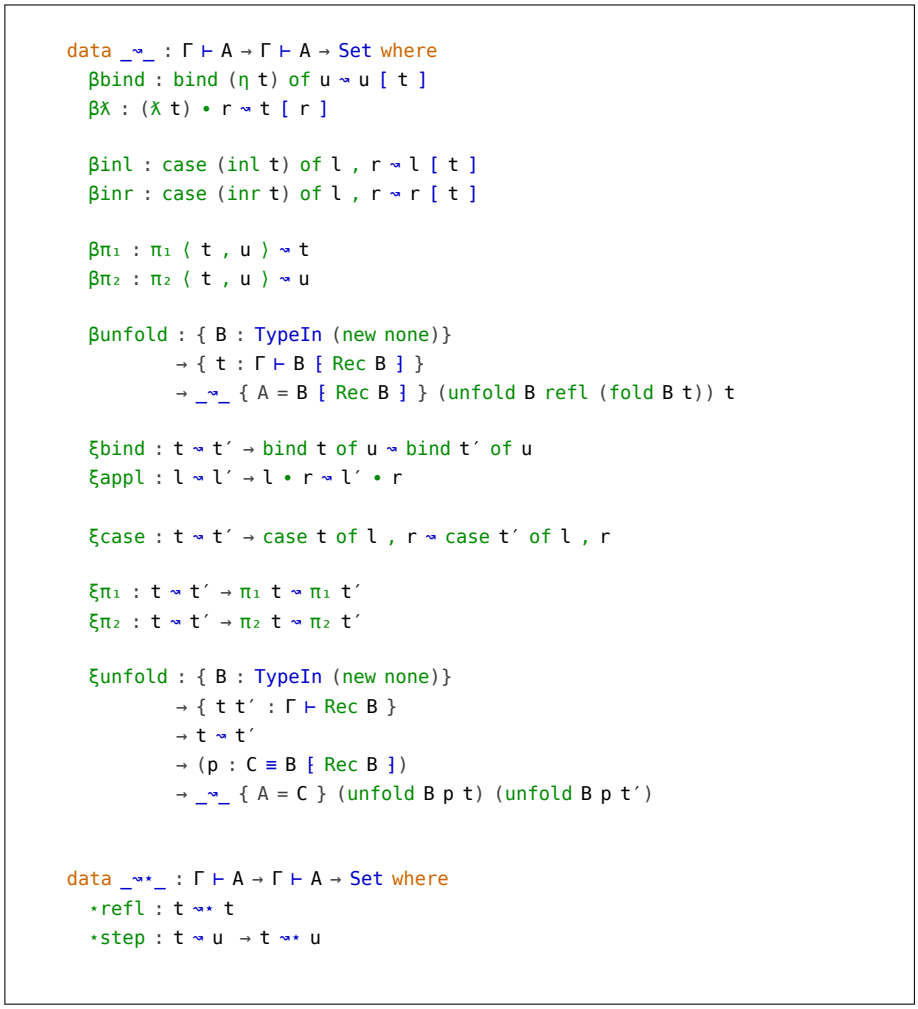
```
bisim : (t₁ t₂ : Γ ⊢ A)
      → pure A
      → Γ ⊢ t₁ ~ t₂ : A
      → t₁ ⇝ t₁′
      → Σ[ t₂′ ∈ Γ ⊢ A ] ((t₂ ⇝ t₂′) ×′ (Γ ⊢ t₁′ ~ t₂′ : A))

bisim* : pure A
       → Γ ⊢ t₁ ~ t₂ : A
       → t₁ ⇝* t₁′
       → Σ[ t₂′ ∈ Γ ⊢ A ] ((t₂ ⇝* t₂′) ×′ (Γ ⊢ t₁′ ~ t₂′ : A))
```

We may now prove Theorem 2:

Fig. 7: Agda contexts, types and terms for the monadic metalanguage.

```
data TyContext : Set where
  none : TyContext
  new  : TyContext → TyContext

data TypeIn : TyContext → Set where
  TyVar : α∈ θ → TypeIn θ
  Unit : TypeIn θ
  T    : TypeIn θ → TypeIn θ
  _⇒_ : TypeIn θ → TypeIn θ → TypeIn θ
  _×_  : TypeIn θ → TypeIn θ → TypeIn θ
  _+_  : TypeIn θ → TypeIn θ → TypeIn θ
  Rec : TypeIn (new θ) → TypeIn θ

Type : Set
Type = TypeIn none

Bool : Type
Bool = Unit + Unit

data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context

data _⊢_ : Context → Type → Set where
  ⋆ : Γ ⊢ Unit

  var : A ∈ Γ → Γ ⊢ A
  η_  : Γ ⊢ A → Γ ⊢ T A
  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B

  _•_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B

  bind_of_ : Γ ⊢ T A → Γ , A ⊢ T B → Γ ⊢ T B

  case_of_,_ : Γ ⊢ A + B → Γ , A ⊢ C → Γ , B ⊢ C → Γ ⊢ C
  inl : Γ ⊢ A → Γ ⊢ A + B
  inr : Γ ⊢ B → Γ ⊢ A + B

  ⟨_,_⟩ : Γ ⊢ A → Γ ⊢ B → Γ ⊢ A × B
  π₁ : Γ ⊢ A × B → Γ ⊢ A
  π₂ : Γ ⊢ A × B → Γ ⊢ B

  fold : (A : TypeIn (new none)) → Γ ⊢ (A [ Rec A ]) → Γ ⊢ Rec A
  unfold : (A : TypeIn (new none)) → (B ≡ A [ Rec A ]) → Γ ⊢ Rec A → Γ ⊢ B
```

Fig. 8: Agda transition relation for the monadic metalanguage.



```
data _⇝_ : Γ ⊢ A → Γ ⊢ A → Set where
  βbind : bind (η t) of u ⇝ u [ t ]
  βɤ : (ɤ t) • r ⇝ t [ r ]

  βinl : case (inl t) of l , r ⇝ l [ t ]
  βinr : case (inr t) of l , r ⇝ r [ t ]

  βπ₁ : π₁ ⟨ t , u ⟩ ⇝ t
  βπ₂ : π₂ ⟨ t , u ⟩ ⇝ u

  βunfold : { B : TypeIn (new none)}
          → { t : Γ ⊢ B [ Rec B ] }
          → _⇝_ { A = B [ Rec B ] } (unfold B refl (fold B t)) t

  ξbind : t ⇝ t′ → bind t of u ⇝ bind t′ of u
  ξappl : l ⇝ l′ → l • r ⇝ l′ • r

  ξcase : t ⇝ t′ → case t of l , r ⇝ case t′ of l , r

  ξπ₁ : t ⇝ t′ → π₁ t ⇝ π₁ t′
  ξπ₂ : t ⇝ t′ → π₂ t ⇝ π₂ t′

  ξunfold : { B : TypeIn (new none)}
          → { t t′ : Γ ⊢ Rec B }
          → t ⇝ t′
          → (p : C ≡ B [ Rec B ])
          → _⇝_ { A = C } (unfold B p t) (unfold B p t′)


data _⇝*_ : Γ ⊢ A → Γ ⊢ A → Set where
  *refl : t ⇝* t
  *step : t ⇝ u → t ⇝* u
```

```
non-interference : (v : ∅ ⊢ Bool)
                 → (M : ∅ , T A ⊢ Bool)
                 → (t : ∅ ⊢ T A)
                 → (u : ∅ ⊢ T A)
                 → M [ t ] ⇓ v
                 → M [ u ] ⇓ v
```

Fig. 9: Agda contexts and terms for the Fitch-style λ-calculus.

```
data Context : Set where
  ∅  : Context
  _,_ : Context → Type → Context
  _■ : Context → Context

data _⊢_ : Context → Type → Set where
  ⋆   : Γ ⊢ Unit
  var : A ∈ Γ → Γ ⊢ A

  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  box : Γ ■ ⊢ A   → Γ ⊢ □ A
  unbox : {ext : Γ is Γ₁ ■ :: Γ₂} → Γ₁ ⊢ □ A → Γ ⊢ A

  _•_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B

  case_of_,_ : Γ ⊢ A + B → Γ , A ⊢ C → Γ , B ⊢ C → Γ ⊢ C
  inl : Γ ⊢ A → Γ ⊢ A + B
  inr : Γ ⊢ B → Γ ⊢ A + B

  ⟨_,_⟩ : Γ ⊢ A → Γ ⊢ B → Γ ⊢ A × B
  π₁ : Γ ⊢ A × B → Γ ⊢ A
  π₂ : Γ ⊢ A × B → Γ ⊢ B

  fold : (A : TypeIn (new none)) → Γ ⊢ (A [ Rec A ]) → Γ ⊢ Rec A
  unfold : (A : TypeIn (new none)) → (B ≡ A [ Rec A ]) → Γ ⊢ Rec A → Γ ⊢ B
```

The formalisation of the Fitch-style calculus proceeds in a similar manner. The terms are given in Fig. 9. Note that the term constructor `unbox` builds in weakening by taking an (implicit) proof of type `Γ is Γ₁ ■ :: Γ₂`, which intuitively means that Γ is of the form $\Gamma_1, \blacksquare, \Gamma_2$ where $\blacksquare \notin \Gamma_2$. This trick, as well as our

definition of substitution at `unbox`, is due to Valliappan, Ruch, and Tomé Cortiñas [38].[2]

After embedding the language inside Agda, the proof statement and proof terms themselves are essentially identical to the ones given for monadic metalanguage; the main difference is that in some cases we carry around an extra proof term showing that the context is free of locks; we can use this to immediately discharge any case including the `unbox` constructor.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J.G. Riecke: A Core Calculus of Dependency. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '99, pp. 147–160. Association for Computing Machinery, San Antonio, Texas, USA (1999). DOI: 10.1145/292540.292555
2. M. Algehed and J.-P. Bernardy: Simple noninterference from parametricity. Proceedings of the ACM on Programming Languages 3(ICFP), 1–22 (2019). DOI: 10.1145/3341693
3. T. Altenkirch and B. Reus: Monadic presentations of lambda terms using generalized inductive types. In: Computer Science Logic, pp. 453–468. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
4. A.W. Appel and D. McAllester: An indexed model of recursive types for foundational proof-carrying code. ACM Transactions on Programming Languages and Systems 23(5), 657–683 (2001). DOI: 10.1145/504709.504712
5. N. Benton, J. Hughes, and E. Moggi: Monads and Effects. In: Applied Semantics. Ed. by G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, pp. 42–122. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
6. W.J. Bowman and A. Ahmed: Noninterference for free. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming - ICFP 2015, pp. 101–113. Association for Computing Machinery (2015). DOI: 10.1145/2784731.2784733
7. P. Choudhury, H. Eades, and S. Weirich: A Dependent Dependency Calculus. In: I. Sergey. Programming Languages and Systems, pp. 403–430. Springer International Publishing, Cham (2022)
8. R. Clouston: "Fitch-Style Modal Lambda Calculi". In: Springer International Publishing, Apr. 2018, pp. 258–275. ISBN: 978-3-319-89365-5. DOI: 10.1007/978-3-319-89366-2_14.
9. R. Davies and F. Pfenning: A modal analysis of staged computation. Journal of the ACM 48(3), 555–604 (2001). DOI: 10.1145/382780.382785
10. D.E. Denning: A Lattice Model of Secure Information Flow. Commun. ACM 19(5), 236–243 (1976). DOI: 10.1145/360051.360056
11. D. Dreyer, A. Ahmed, and L. Birkedal: Logical Step-Indexed Logical Relations. Logical Methods in Computer Science 7(2), 1–37 (2011). DOI: 10.2168/LMCS-7(2:16)2011
12. A. Filinski: On the relations between monadic semantics. Theoretical Computer Science 375(1-3), 41–75 (2007). DOI: 10.1016/j.tcs.2006.12.027

[2] We remark that their definition of substitution arises from the structure of a parametric right adjoint: see [13].

13. D. Gratzer, E. Cavallo, G.A. Kavvos, A. Guatto, and L. Birkedal: Modalities and Parametric Adjoints. ACM Transactions on Computational Logic 23(3), 1–29 (2022). DOI: 10.1145/3514241
14. D. Gratzer, G.A. Kavvos, A. Nuyts, and L. Birkedal: Multimodal Dependent Type Theory. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 492–506. Association for Computing Machinery (2020). DOI: 10.1145/3373718.3394736
15. D. Gratzer, G.A. Kavvos, A. Nuyts, and L. Birkedal: Multimodal Dependent Type Theory. Logical Methods in Computer Science 17(3) (2021). DOI: 10.46298/lmcs-17(3:11)2021
16. S.O. Gregersen, J. Bay, A. Timany, and L. Birkedal: Mechanized logical relations for termination-insensitive noninterference. Proceedings of the ACM on Programming Languages 5(POPL) (2021). DOI: 10.1145/3434291
17. C.A. Gunter: Semantics of Programming Languages: Structures and Techniques. MIT Press (1992)
18. R. Harper: Practical Foundations for Programming Languages. Cambridge University Press (2016)
19. J. Jang, S. Gélineau, S. Monnier, and B. Pientka: Moebius: metaprogramming using contextual types: the stage where system F can pattern match on itself. Proceedings of the ACM on Programming Languages 6(POPL) (2022). DOI: 10.1145/3498700
20. G.A. Kavvos: On the Semantics of Intensionality. In: J. Esparza and A.S. Murawski. Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, pp. 550–566. Springer Berlin Heidelberg, Berlin, Heidelberg (2017). DOI: 10.1007/978-3-662-54458-7_32
21. G.A. Kavvos: Modalities, Cohesion, and Information Flow. Proceedings of the ACM on Programming Languages 3(POPL) (2019). DOI: 10.1145/3290333
22. G.A. Kavvos: Dual-Context Calculi for Modal Logic. Logical Methods in Computer Science 16(3) (2020). DOI: 10.23638/LMCS-16(3:10)2020
23. G.A. Kavvos: Intensionality, Intensional Recursion, and the Gödel-Löb axiom. Journal of Applied Logics - IfCoLog Journal of Logics and their Applications 8(8), 2287–2311 (2021)
24. G.A. Kavvos: On the Semantics of Intensionality and Intensional Recursion. DPhil thesis, University of Oxford (2017).
25. D.R. Licata, I. Orton, A.M. Pitts, and B. Spitters: Internal Universes in Models of Homotopy Type Theory. In: H. Kirchner. 3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018). Leibniz International Proceedings in Informatics (LIPIcs), 22:1–22:17. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018). DOI: 10.4230/LIPIcs.FSCD.2018.22
26. C. McBride: A polynomial testing principle
27. K. Miyamoto and A. Igarashi: A Modal Foundation for Secure Information Flow. In: Proceedings of the Workshop on Foundations of Computer Security (FCS'04), pp. 187–203 (2004)
28. E. Moggi: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991). DOI: 10.1016/0890-5401(91)90052-4
29. F. Pfenning and R. Davies: A Judgmental Reconstruction of Modal Logic. Mathematical Structures in Computer Science 11(4), 511–540 (2001). DOI: 10.1017/S0960129501003322
30. A.M. Pitts: Relational Properties of Domains. Information and Computation 127(2), 66–90 (1996). DOI: 10.1006/inco.1996.0052

31. V. Rajani and D. Garg: On the expressiveness and semantics of information flow types. Journal of Computer Security 28(1), 129–156 (2020). DOI: 10.3233/JCS-191382
32. D. Sangiorgi: Introduction to Bisimulation and Coinduction. Cambridge University Press (2011)
33. N. Shikuma and A. Igarashi: Proving Noninterference by a Fully Complete Translation to the Simply Typed lambda-calculus. Logical Methods in Computer Science 4(3), 10 (2008). DOI: 10.2168/LMCS-4(3:10)2008
34. M. Shulman: Brouwer's fixed-point theorem in real-cohesive homotopy type theory. Mathematical Structures in Computer Science 28(6), 856–941 (2018). DOI: 10.1017/S0960129517000147
35. C. Strachey: Fundamental Concepts in Programming Languages. Higher-Order and Symbolic Computation 13, 11–49 (2000). DOI: 10.1023/A:1010000313106
36. E. Sumii and B.C. Pierce: Logical relations for encryption. Journal of Computer Security 11(4), 521–554 (2003). DOI: 10.3233/JCS-2003-11403
37. S. Tse and S. Zdancewic: Translating dependency into parametricity. In: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming - ICFP '04. Association for Computing Machinery (2004). DOI: 10.1145/1016850.1016868
38. N. Valliappan, F. Ruch, and C. Tomé Cortiñas: Normalization for Fitch-Style Modal Calculi. Proceedings of the ACM on Programming Languages 6(ICFP) (2022). DOI: 10.1145/3547649