TU Delft

Delft University of Technology

Modal μ-Calculus for Free in Agda

Todorov, Ivan; Poulsen, Casper Bach

DOI
10.1145/3678000.3678202

Publication date
2024

Document Version
Final published version

Published in
TyDe 2024

Citation (APA)
Todorov, I., & Poulsen, C. B. (2024). Modal μ-Calculus for Free in Agda. In S. Alves, & J. Cockx (Eds.), *TyDe 2024: Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development* (pp. 16-28). ACM. https://doi.org/10.1145/3678000.3678202

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Modal μ-Calculus for Free in Agda

Ivan Todorov
Delft University of Technology
Netherlands
i.todorov@student.tudelft.nl

Casper Bach Poulsen
Delft University of Technology
Netherlands
c.b.poulsen@tudelft.nl

Abstract

Expressive logics, such as the modal μ-calculus, can be used to specify and verify functional requirements of program models. While such verification is useful, a key challenge is to guarantee that the model being verified actually corresponds to the (typically effectful) program it is supposed to. We explore an approach that bridges this gap between effectful programming and functional requirement verification. Using dependently-typed programming in Agda, we develop an embedding of the modal μ-calculus for defining and verifying functional properties of possibly-non-terminating effectful programs which we represent in Agda using the coinductive free monad.

CCS Concepts: • **Theory of computation** → **Program verification**; **Modal and temporal logics**; **Logic and verification**.

Keywords: Formal Verification, Modal μ-Calculus, Algebraic Effects, Free Monad, Agda

ACM Reference Format:
Ivan Todorov and Casper Bach Poulsen. 2024. Modal μ-Calculus for Free in Agda. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '24), September 6, 2024, Milan, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3678000.3678202

1 Introduction

An important aspect of developing reliable software is to validate that it satisfies its functional requirements. A popular approach to validating functional requirements is via testing. However, as Dijkstra [12] famously put it, *program testing can be used to show the presence of bugs, but never to show their absence!* A safer approach is to formally verify that a program satisfies its functional requirements. In order to do this, we must decide on the following:

1. How are we going to model the given software?
2. How are we going to represent the functional requirements of the software as properties of the model?
3. Are we going to do the verification manually (e.g., in a proof assistant) or do we want to use an automatic verification technique (e.g., model checking)?

For model checking we can use tools such as TLA+ [16] or mCRL2 [13] to verify the functional requirements of our software. For example, mCRL2 lets us specify our software formally as a labelled transition system and provides a powerful logic – *the modal μ-calculus* [15] – for automatically verifying properties of that system. However, a downside of traditional model checking is that we are verifying a model instead of the actual software system.

Recent work by Dal Lago and Ghyselen [8] suggests a promising alternative direction, namely model-checking programs involving algebraic effects and handlers. The attraction of this approach is that it would allow us to state and prove high-level functional requirements about programs *directly*. However, as Dal Lago and Ghyselen [8] demonstrate, this model-checking problem is, in general, undecidable.

In this paper we explore a different approach: we embed the modal μ-calculus into the dependently-typed language Agda,[1] such that we can state and prove functional requirements of programs with algebraic effects and handlers directly. Our embedding makes it possible to formally verify that a given program satisfies its functional requirements. It should be noted that, since we work in Agda, our work has been carried out in an intuitionistic setting. Therefore, in order to maintain the traditional semantics of modal μ-calculus formulas, which are subject to the laws of classical logic, we use the law of excluded middle as a *postulate*[2] in Agda. Nevertheless, while our embedding currently requires programmers to manually prove that the functional requirements are met, our work demonstrates that dependently-typed languages, such as Agda, are suited for expressing and verifying functional requirements using the expressive logic of the modal μ-calculus. We believe the core ideas of our embedding are transferable to other dependently-typed languages, such as Idris[3], Coq[4], or Lean[5]. However, we do rely

This work is licensed under a Creative Commons Attribution 4.0 International License.

TyDe '24, September 6, 2024, Milan, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1103-9/24/09
https://doi.org/10.1145/3678000.3678202

[1] https://agda.readthedocs.io
[2] https://agda.readthedocs.io/en/v2.6.4.3-r1/language/postulates.html
[3] https://idris2.readthedocs.io/en/latest/
[4] https://coq.inria.fr/
[5] https://lean-lang.org/

extensively on Agda's support for *dependent pattern matching*, *guarded coinduction*, and *copattern matching* [4], which may necessitate alternative encodings in other languages.

The main contribution of this paper is that it demonstrates how to use *dynamic logic* – in particular, the *first-order modal μ-calculus* – to represent functional requirements of effectful programs within a proof assistant. More specifically, we make the following technical contributions:

- First (in § 4), we present a deep embedding in Agda of a simple dynamic logic known as *Hennessy-Milner logic*, which is the core of the first-order modal μ-calculus.
- Then (in § 5), we add support for *action formulas* – a way of representing a set of operations.
- Next (in § 6), we extend our embedding of Hennessy-Milner logic by introducing least- and greatest-fixed-point operators, thereby turning it into an embedding of the modal μ-calculus.
- Subsequently (in § 7), we add support for regular formulas – a way of representing a (possibly infinite) sequence of action formulas.
- After that (in § 8), we extend our embedding of the modal μ-calculus by introducing operators for existential and universal quantification and by adding parameters to the fixed-point operators, thereby turning it into an embedding of the first-order modal μ-calculus.

The paper is structured as follows: first (in § 2), we discuss examples of expressing simple functional requirements using the first-order modal μ-calculus; then (in § 3), we show how to define the *coinductive free monad* in Agda, which lets us represent possibly-non-terminating sequential effectful programs; after that (in §§ 4 to 8), we present our embedding of the first-order modal μ-calculus in Agda;[6] finally, we discuss related work (in § 9) and conclude and discuss future work (in § 10).

2 Using the First-Order Modal μ-Calculus

In this section we show how to use the first-order modal μ-calculus to express functional requirements of a simple effectful program. As our running example we will use the following contrived ATM which allows users to view the balance of their bank account:

1. To start using the ATM, a user must insert their card;
2. Then, the user has to provide their PIN code;
3. Next, the ATM checks the PIN code;
4. If the PIN code is correct, the bank account balance is displayed, after which the bank card is ejected and the ATM goes back to its initial state; otherwise, the ATM throws an exception and halts.

Below is a sequential program that implements this behavior using operations named according to their effects (*getPIN*, *correctPIN*, etc.). We can think of it as a sequence of operations which is executed a (possibly infinite) number of times, halting only if an incorrect PIN is entered.

```
1  ATM =
2    getPIN;
3    if correctPIN then
4      showBalance;
5      ATM;
6    else
7      throwException;
```

An intuitive functional requirement which our ATM program must satisfy is that, when the ATM's first user starts interacting with it, they must provide their PIN code:

> At the start of the program it must be possible to execute the *getPIN* operation.

The same must also be true for all subsequent users of the ATM. However, for now, let us only focus on the first user. We observe that our requirement is a *modal statement*, because it talks about what must be possible in a particular situation, namely when the program is first executed. Therefore, it can be represented using dynamic logic as follows:

$$\langle\ \mathit{getPIN}\ \rangle\ \mathsf{true}$$

For a given program (represented as a sequence of operations) the formula ⟨ A ⟩ F, where A is some operation and F is some formula, can be read as "the first operation is A and the remaining formula F holds for at least one possible continuation of the program". To see what we mean by *possible continuation* here, consider, for example, the *correctPIN* operation on line 3 which has two possible continuations: one for the case in which the operation returns true (the ***then*** branch) and one for false. The formula true holds for all programs. Our *ATM* program above satisfies the formula ⟨ *getPIN* ⟩ true, since its first operation is indeed *getPIN* and the formula true trivially holds for the remaining program.

Another functional requirement of our ATM is that, when the ATM's first user starts interacting with it, they must not be able to directly view their bank account balance:

> At the start of the program it must not be possible to execute the *showBalance* operation.

This property must also hold for all subsequent users of the ATM, but for simplicity we once again focus on the first user. Its representation in dynamic logic is:

$$[\ \mathit{showBalance}\ ]\ \mathsf{false}$$

For some program (some sequence of operations) the formula [A] F, where A is some operation and F is some formula, can be read as "if the first operation is A, then the remaining formula F holds for all possible continuations of the program". The formula false does not hold for any program. Therefore, for any operation A the only scenario in which the formula

[6] The full source code of our work can be found at https://github.com/ivanstodorov/modal-mu-calculus-for-free.

[A] false holds for some program is when the first operation in that program is not A, since, if the first operation in the program is A, then the formula false must hold for the remaining program, which is impossible. Our ATM program above satisfies the formula [*showBalance*] false, because its first operation is not *showBalance*.

The functional requirements discussed above all made the simplifying assumption that they only talk about the initial state of the ATM. The modal μ-calculus can also be used to express more complex requirements, such as the following:

> It is not possible to execute the *showBalance* operation before executing the *getPIN* operation.

A difference from the previous requirements is that this requirement should be true for all users of our ATM, not only the first one. We can express this requirement using *action formulas* and *parameterized fixed-point operators* [13]. An action formula represents a set of operations. For example, the action formula true represents the set of all possible operations, whereas false represents the empty set of operations. Furthermore, any single operation is an action formula which represents the singleton set containing that operation. We can also take the complement of a set of operations (X^c), the union of two sets of operations ($X \cup Y$), and the intersection of two sets of operations ($X \cap Y$), where X and Y are action formulas. The parameterized fixed-point operators let us define formulas, respectively, inductively (via the least-fixed-point operator μ) and coinductively (via the greatest-fixed-point operator ν). Moreover, these operators introduce a user-defined list of parameters, which can be used throughout the remainder of the formula, and, whenever a fixed-point operator is referenced, it needs to be given a well-typed list of values for its parameters. Using these features, our functional requirement can be expressed as follows:

$$\begin{aligned}\nu\, X\ (b{:}bool{:=}false)\ .\ &[\ (\mathit{getPIN} \cup \mathit{showBalance})^{\,c}\ ]\ X(b) \wedge \\ &[\ \mathit{getPIN}\ ]\ ((\neg\, b) \wedge X(true)) \wedge \\ &[\ \mathit{showBalance}\ ]\ (b \wedge X(false))\end{aligned}$$

By using the greatest-fixed-point operator, we are defining a property of a possibly-infinite sequence of operations. This property says that, initially, we have not executed *getPIN* (b:bool:=false); that executing any operation other than *getPIN* or *showBalance* is inconsequential; that executing *getPIN* is possible only if it has not already been executed for the current user (otherwise ¬ b would not hold) and leads to a state in which *getPIN* has been executed for the current user (X(true)); and that executing *showBalance* is possible only if *getPIN* has already been executed for the current user (otherwise b would not hold) and leads to a state in which *getPIN* has not yet been executed for the current user (X(false)).

This formula serves as an example of the expressiveness of the first-order modal μ-calculus. In the remainder of this paper we are first going to present an embedding of algebraic effects in Agda which lets us write programs, like the contrived ATM program, using the coinductive free monad. Subsequently, we develop an embedding in Agda of the first-order modal μ-calculus which lets us verify properties of effectful programs, like the ones discussed in this section.

3 Modelling Effectful Programs

In this section we explain how we can model sequential effectful programs using the *free monad* [6, 19] in Agda. First (in § 3.1), we present an implementation of the inductive free monad and show how it can be used to represent finite sequential effectful programs. Then (in § 3.2), we show how our implementation can be extended to the coinductive free monad, which can be used to represent programs with infinite sequences of operations. After that (in § 3.3), we discuss an important property of algebraic effects, namely that we can combine a number of effects into a single effect which contains all of their operations. Finally (in § 3.4), we describe how we can define smart injections for our effects which reduce the notational overhead of programs with combined effects. §§ 3.1 and 3.3 describe known work on modeling and programming with the free monad in Agda. Readers familiar with this may wish to only read §§ 3.2 and 3.4 about the coinductive free monad and smart injections.

3.1 The Inductive Free Monad

The free monad is a structure which models the syntax of effectful computations and is typically defined as follows:[7]

```
data Free^f (f : Set → Set) (α : Set) : Set where
  pure : α → Free^f f α
  impure : f (Free^f f α) → Free^f f α
```

Here, the functor f is a so-called *signature functor*, representing the types of operations that can occur in the computation, and α is the type of the final result. However, this definition is not *strictly positive*[8] and is therefore not accepted by Agda. We remedy this by using *containers* [1, 2] – a means of representing strictly-positive functors. Containers are defined in Agda's standard library as follows:

```
record Container (s p : Level) : Set (suc (s ⊔ p)) where
  constructor _▷_
  field Shape : Set s
        Position : Shape → Set p
```

Here, we can think of Shape as the operations of an effect and of Position as the return type of each operation. The *extension* of a container gives us a functor on Set:

```
⟦_⟧ : ∀ {s p ℓ} → Container s p → Set ℓ → Set (s ⊔ p ⊔ ℓ)
⟦ S ▷ P ⟧ X = Σ[ s ∈ S ] (P s → X)
```

[7] A universe polymorphic definition of this datatype is also possible.
[8] https://agda.readthedocs.io/en/v2.6.4.3-r1/language/data-types.html#strict-positivity

Using these definitions, the free monad is given by the following datatype:

```
data _*_ (C : Container s p) (X : Set x)
         : Set (x ⊔ s ⊔ p) where
  pure : X → C * X
  impure : ⟦ C ⟧ (C * X) → C * X
```

This inductive definition models any finite sequence of operations of an effect described by a container. To illustrate, let us try to implement the contrived ATM program from § 2. We first define the operations of our ATM as a container:

```
data EffectShape : Set where
  getPIN : EffectShape
  correctPIN : ℕ → EffectShape
  showBalance : EffectShape
  throwException : EffectShape

effect : Container 0ℓ 0ℓ
Shape effect = EffectShape
Position effect getPIN = ℕ
Position effect (correctPIN _) = Bool
Position effect showBalance = ⊤
Position effect throwException = ⊥
```

Using this container, a simplified and terminating version of the ATM program from § 2 can be represented as follows:[9]

```
ATMᵗ : effect * ⊤
ATMᵗ = impure (getPIN , λ where
  n → impure (correctPIN n , λ where
    false → impure (throwException , ⊥-elim)
    true → impure (showBalance , pure)))
```

Unlike the ATM program from § 2, the ATM program above halts after showing the balance, instead of recursively returning to its initial state. To model the (potentially infinitely) recursive behavior of the ATM program from § 2 we use a coinductive implementation of the free monad instead.

3.2 Representing Recursion: The Coinductive Free Monad

To define the coinductive free monad, we need a coinductive datatype with the same structure as _*_. We model this structure using a coinductive record type in Agda. To define this record type, we first define the following datatype which captures the core structure of the free monad:

```
data Free (F : Container ℓ₁ ℓ₂ → Set ℓ₃ → Set ℓ₄)
          (C : Container ℓ₁ ℓ₂)
          (α : Set ℓ₃) : Set (ℓ₁ ⊔ ℓ₂ ⊔ ℓ₃ ⊔ ℓ₄) where
  pure : α → Free F C α
  impure : ⟦ C ⟧ (F C α) → Free F C α
```

[9]We use pattern matching lambda expressions (https://agda.readthedocs.io/en/v2.6.4.3-r1/language/lambda-abstraction.html#pattern-matching-lambda) to introduce the results of each operation.

The difference between Free and _*_ is that, while the impure constructor of _*_ recursively calls the datatype _*_ itself, in the impure constructor of Free that recursive call is replaced by a call to the parameter F. This lets us use Free to define the following inductive record type:

```
record IndFree (C : Container ℓ₁ ℓ₂)
               (α : Set ℓ₃) : Set (ℓ₁ ⊔ ℓ₂ ⊔ ℓ₃) where
  inductive; constructor ⟪_⟫
  field free : Free IndFree C α
```

This IndFree record type is isomorphic to _*_. The coinductive version of the free monad can be defined analogously as a coinductive record type:[10]

```
record CoFree (C : Container ℓ₁ ℓ₂)
              (α : Set ℓ₃) : Set (ℓ₁ ⊔ ℓ₂ ⊔ ℓ₃) where
  coinductive; constructor ⟪_⟫
  field free : Free CoFree C α
```

In the rest of this paper we will use the coinductive free monad as our representation of effectful program trees:

```
Program : Container ℓ₁ ℓ₂ → Set ℓ₃ → Set (ℓ₁ ⊔ ℓ₂ ⊔ ℓ₃)
Program = CoFree
```

For example, our ATM program from § 2 can be modelled as follows:

```
ATM : Program effect ⊤
free ATM = impure (getPIN , λ where
  n → ⟪ impure (correctPIN n , λ where
    false → ⟪ impure (throwException , ⊥-elim) ⟫
    true → ⟪ impure (showBalance , λ _ → ATM) ⟫) ⟫)
```

3.3 Composing Effects

The previous section modelled the operations of our ATM using a single container, representing a single effect. An attractive property of algebraic effects and the free monad is that we can define and combine separate effects. Concretely, the following function represents the sum of two containers:

```
_⊎_ : (C₁ : Container s₁ p) → (C₂ : Container s₂ p) →
      Container (s₁ ⊔ s₂) p
(C₁ ⊎ C₂) .Shape = (Shape C₁ Sum.⊎ Shape C₂)
(C₁ ⊎ C₂) .Position = [ Position C₁ , Position C₂ ]′
```

Here, [_,_]′ is the non-dependent eliminator for the sum datatype Sum._⊎_. Using the function _⊎_ we can, for example, define and combine separate effects for the different operations of our ATM. For brevity, we show just the definition of the IOEffect.

```
data IOShape : Set where
  getPIN : IOShape
```

[10]This coinductive record type corresponds to a version of ITrees [21] without an explicit constructor for "silent" transitions. Silent transitions can be represented using a (container-encoded) signature functor.

```
  showBalance : IOShape

IOEffect : Container 0ℓ 0ℓ
Shape IOEffect = IOShape
Position IOEffect getPIN = ℕ
Position IOEffect showBalance = ⊤
```

Other effects for verifying that a PIN code is correct (verificationEffect) and for throwing exceptions (exceptionEffect) can be defined analogously. Then, we can combine them into a single effect as follows:

```
effect⁺ : Container 0ℓ 0ℓ
effect⁺ = IOEffect ⊎ verificationEffect ⊎ exceptionEffect
```

Using effect⁺, our ATM is implemented as follows:

```
ATM⁺ : Program effect⁺ ⊤
free ATM⁺ = impure (inj₁ getPIN , λ where
  n → ⟪ impure (inj₂ (inj₁ (correctPIN n)) , λ where
    false →
      ⟪ impure (inj₂ (inj₂ throwException) , ⊥-elim) ⟫
    true →
      ⟪ impure (inj₁ showBalance , λ _ → ATM⁺) ⟫) ⟫)
```

This program is difficult to read because of all of the injections. The usual solution to this problem is to use *smart constructors* [19] to make programs more readable. Here, we will use *smart injections* which let us generically inject an operation of a smaller effect into a larger one.

3.4 Smart Injections

The following record type represents a witness that we can inject a container C_1 into C_2:

```
record _:<:_ (C₁ : Container ℓ₁ ℓ₂) (C₂ : Container ℓ₃ ℓ₄) :
  Set (ℓ₁ ⊔ ℓ₂ ⊔ ℓ₃ ⊔ ℓ₄) where
  field
    injS : Shape C₁ → Shape C₂
    projP : ∀ {s} → Position C₂ (injS s) → Position C₁ s
```

Using Agda's support for *instance arguments*[11] [11], we can declare a generic function that allows Agda to automatically infer injection witnesses that inject an operation of a smaller effect type into a larger one:

```
inj : {C₁ : Container ℓ₁ ℓ₂} {C₂ : Container ℓ₃ ℓ₄}
    ⦃ _ : C₁ :<: C₂ ⦄ → Shape C₁ → Shape C₂
inj ⦃ inst ⦄ x = injS inst x
```

Using smart injections, our ATM program becomes more readable:

```
ATMˢ : Program effect⁺ ⊤
free ATMˢ = impure (inj getPIN , λ where
  n → ⟪ impure (inj (correctPIN n) , λ where
    false →
      ⟪ impure (inj throwException , ⊥-elim) ⟫
    true →
      ⟪ impure (inj showBalance , λ _ → ATMˢ) ⟫) ⟫)
```

It is equally possible to write smart constructors akin to those used by Swierstra [19]; e.g., a function getPINˢ : ⦃ IOEffect :<: C ⦄ → Program C ℕ. However, to program with smart constructors we must use the monadic bind of the coinductive free monad. While it is possible to define a notion of monadic bind for the coinductive free monad, it can be problematic to use it in conjunction with guarded coinduction in Agda, because Agda's guardedness checker is unable to infer that recursive calls are guarded in continuations of a monadic bind.[12] For that reason, we do not use monadic bind or smart constructors in this paper. We will, however, use the smart injection function inj discussed above.

4 Hennessy-Milner Logic

We are now ready to formalize the first-order modal μ-calculus in Agda. To start, we will focus on a subset of the first-order modal μ-calculus: a simple dynamic logic known as Hennessy-Milner logic (HML) [14]. The syntax of HML formulas is:

$$\begin{aligned} f \;::=\;& \mathsf{true} \mid \mathsf{false} \mid \sim f \mid f \wedge f \mid f \vee f \mid f \Rightarrow f \\ \mid\;& \langle\, \alpha \,\rangle\, f \mid [\, \alpha \,]\, f \end{aligned}$$

Here, true is a formula that holds for all programs, and false does not hold for any program. The standard connectives from predicate logic maintain their traditional meanings, e.g., negation (∼ f), conjunction (f ∧ f), disjunction (f ∨ f) and implication (f ⇒ f). The diamond (⟨ α ⟩ f) and box ([α] f) modalities we introduced in § 2. We represent HML formulas in Agda using the following Formula datatype.

```
data Formula (S : Set ℓ) : Set ℓ where
  true false : Formula S
  ~_ : Formula S → Formula S
  _∧_ _∨_ _⇒_ : (_ _ : Formula S) → Formula S
  ⟨_⟩_ [_]_ : S → Formula S → Formula S
```

The datatype S represents the operations which can be used in the formula. The semantics of HML formulas is given by the satisfaction relation _⊨_ in Figure 1. We say that a given program x satisfies a HML formula f, when x ⊨ f.

Although this implementation does not support all features of the first-order modal μ-calculus, we can still use it to express and prove some functional requirements of our ATM example. For example, the functional requirement that at the start of the program it must be possible to execute the *getPIN* operation, can be expressed as follows:

```
property₁ : Formula (Shape effect⁺)
property₁ = ⟨ inj getPIN ⟩ true
```

[11] https://agda.readthedocs.io/en/v2.6.4.3-r1/language/instance-arguments.html

[12] It should be possible to use Agda's *sized types* [3] instead. We do not explore that here.

```
_⊨_ : {C : Container ℓ₁ ℓ₂} → {α : Set ℓ₃} →
      Program C α → Formula (Shape C) → Set (ℓ₁ ⊔ ℓ₂)
_ ⊨ true = ⊤
_ ⊨ false = ⊥
x ⊨ (~ f) = ¬ (x ⊨ f)
x ⊨ (f₁ ∧ f₂) = (x ⊨ f₁) × (x ⊨ f₂)
x ⊨ (f₁ ∨ f₂) = (x ⊨ f₁) ⊎ (x ⊨ f₂)
x ⊨ (f₁ ⇒ f₂) = (x ⊨ f₁) → (x ⊨ f₂)
x ⊨ (⟨ s₁ ⟩ f) with free x
... | pure _ = ⊥
... | impure (s₂ , c) = (s₁ ≡ s₂) × (∃[ p ] c p ⊨ f)
x ⊨ ([ s₁ ] f) with free x
... | pure _ = ⊤
... | impure (s₂ , c) = (s₁ ≡ s₂) → (∀ p → c p ⊨ f)
```

Figure 1. Semantics of HML formulas in Agda

Using the satisfaction relation we can prove that our ATM's software satisfies this functional requirement as follows:

```
proof₁ : ATMˢ ⊨ property₁
proof₁ = refl , zero , tt
```

Another functional requirement which we can express is that at the start of the program it must not be possible to execute any operation other than the *getPIN* operation. Unfortunately, since HML does not support *action formulas* (we introduce them in § 5), we have to explicitly mention each impossible operation to encode this requirement:

```
property₂ : ℕ → Formula (Shape effect⁺)
property₂ n = [ inj (correctPIN n) ] false ∧
              [ inj showBalance ] false ∧
              [ inj throwException ] false
```

We can again use the satisfaction relation to prove that our ATM satisfies this requirement.

```
proof₂ : (n : ℕ) → ATMˢ ⊨ property₂ n
proof₂ n = (λ ()) , (λ ()) , λ ()
```

However, this functional requirement can be expressed in a better way using action formulas, as described in the next section.

5 Action Formulas

Action formulas represent sets of operations. Their syntax is:

$$\text{af} \ ::= \ \alpha \mid \text{true} \mid \text{false} \mid \text{af}^{\,c} \mid \text{af} \cap \text{af} \mid \text{af} \cup \text{af}$$

As we mentioned in § 2, every operation α is also an action formula which represents a singleton set containing only that operation. Moreover, the action formula true represents the set of all operations and false the empty set of operations. The non-terminal symbols $\text{af}^{\,c}$, $\text{af} \cap \text{af}$ and $\text{af} \cup \text{af}$ in the grammar represent the complement of a set of operations, the intersection of two sets of operations and the union of two sets of operations, respectively. We can represent action formulas in Agda using the following datatype:

```
data ActionFormula (S : Set ℓ) : Set ℓ where
  true false : ActionFormula S
  act_ : S → ActionFormula S
  _ᶜ : ActionFormula S → ActionFormula S
  _∩_ _∪_ : (_ _ : ActionFormula S) → ActionFormula S
```

Furthermore, we can define a function which checks whether a given operation is part of the set of operations represented by a given action formula as follows:

```
_∈_ : {S : Set ℓ} → S → ActionFormula S → Set ℓ
_ ∈ true = ⊤
_ ∈ false = ⊥
s₁ ∈ (act s₂) = s₁ ≡ s₂
s ∈ (af ᶜ) = ¬ (s ∈ af)
s ∈ (af₁ ∩ af₂) = (s ∈ af₁) × (s ∈ af₂)
s ∈ (af₁ ∪ af₂) = (s ∈ af₁) ⊎ (s ∈ af₂)
```

Now that we have all of the necessary definitions to use action formulas, we need to incorporate them into our definition of HML formulas discussed in § 4. We can do this by changing the constructors for the diamond and box modalities to use action formulas, instead of single operations:

```
⟨_⟩_ [_]_ : ActionFormula S → Formula S → Formula S
```

Furthermore, we need to modify the semantics of our HML formulas shown in Figure 1, in order to account for the changes in the definition. In particular, we need to redefine the semantics for the diamond and box modalities as follows:

```
x ⊨ (⟨ af ⟩ f) with free x
... | pure _ = ⊥
... | impure (s , c) = (s ∈ af) × (∃[ p ] c p ⊨ f)
x ⊨ ([ af ] f) with free x
... | pure _ = ⊤
... | impure (s , c) = (s ∈ af) → (∀ p → c p ⊨ f)
```

Using our new definition of HML, we can now define the functional requirement that at the start of the program it must not be possible to execute any operation other than the *getPIN* operation, as follows:

```
property₃ : Formula (Shape effect⁺)
property₃ = [ (act (inj getPIN)) ᶜ ] false
```

Moreover, we can prove that our ATM's software satisfies this functional requirement as follows:

```
proof₃ : ATMˢ ⊨ property₃
proof₃ h = ⊥-elim (h refl)
```

As we can see, both the definition of the functional requirement and the proof that it is satisfied become cleaner with the use of action formulas, compared to the versions presented at the end of § 4. However, while action formulas are very

convenient, since they make some functional requirements simpler to represent, they do not increase the expressivity of our HML formulas. Thus, in the next section we will look at how we can extend our implementation of HML formulas by adding least- and greatest-fixed-point operators, thereby changing it into an implementation of a more expressive dynamic logic, known as the modal μ-calculus.

6 The Modal μ-Calculus

In this section we are going to extend our implementation of HML by adding least- and greatest-fixed-point operators, thereby transforming it into an implementation of a more expressive dynamic logic, known as the modal μ-calculus. First (in § 6.1), we will present the full syntax of modal μ-calculus formulas, explain the functionality behind the new structures which it introduces and describe an intuitive, but unsuccessful, attempt to implement it in Agda. Then (in § 6.2), we are going to discuss how we have solved the problems of the intuitive implementation and we will give an example of how our implementation can be used.

6.1 Intuitive Initial Attempt

The modal μ-calculus is a fixed-point dynamic logic and its syntax is described by the following BNF grammar:

$$\begin{aligned} f ::= \ & \mathsf{true} \mid \mathsf{false} \mid \sim f \mid f \wedge f \mid f \vee f \mid f \Rightarrow f \\ \mid \ & \langle\, \alpha \,\rangle\, f \mid [\, \alpha \,]\, f \mid \mu\, X \,.\, f \mid \nu\, X \,.\, f \mid X \end{aligned}$$

As we can see from this grammar, all features of HML are also present in the modal μ-calculus. However, there are three new symbols in the grammar: variables X which represent formula variables bound by least- and greatest-fixed-point operators, $\mu\, X \,.\, f$ and $\nu\, X \,.\, f$, respectively. Using these three new constructs it is possible to express requirements which involve recursion (which is also the reason why the modal μ-calculus is sometimes referred to as HML with recursion). To explain how this works, let us consider what the fixed-point operators represent. The fixed-point operators $\mu\, X \,.\, f$ and $\nu\, X \,.\, f$ introduce a new formula variable X which is bound in the body f of the corresponding fixed-point operator. The occurrences of X in f represent recursive references to the fixed-point operator (i.e., $\mu\, X \,.\, f \simeq \mu\, X \,.\, f[\mu\, X \,.\, f/X]$).

Next, to see how the least- and greatest-fixed-point operators work, let us consider two example formulas, $\mu\, X \,.\, X$ and $\nu\, X \,.\, X$. These formulas represent the least fixed point and the greatest fixed point, respectively, of a formula that recursively references itself. The difference between them is in the proofs of satisfiability which they require. Proofs that a *least*-fixed-point formula holds for a given program are given inductively (i.e., they correspond to finite derivation trees). In contrast, proofs that a *greatest*-fixed-point formula holds for a given program are given coinductively (i.e., they correspond to corecursively generated, possibly-infinite derivation trees). Thus, the formula $\mu\, X \,.\, X$ is not satisfied by any program, since for all programs the least-fixed-point operator will be referenced infinitely many times, while the formula $\nu\, X \,.\, X$ is (trivially) satisfied by all programs.

Now that we have some intuition about the fixed-point operators, let us try to implement them in Agda. A logical first attempt would be to use a datatype Formula′ which extends the Formula datatype with the following three constructors:

```
μ_ ν_ : (Formula′ S → Formula′ S) → Formula′ S
ref_ : Formula′ S → Formula′ S
```

These constructors are designed based on the intuition which we discussed above – the fixed-point operators require a function, that takes a formula as input (the new formula variable) and returns a formula, while the constructor ref_ can be used to reference the formula variables which are introduced by the fixed-point operators. Unfortunately, this definition of the fixed-point operators is not accepted by Agda, since the datatype Formula′ appears as an argument to a function in one of its constructors and is therefore not strictly-positive.

In order to solve this problem, we need to come up with a different way of representing the references to Formula′ in all of the constructors. Fortunately, we can also think of formulas as predicates on programs. Thus, we can define a datatype Formula″ which has the same constructors as Formula′, but expressed as follows:

```
μ_ ν_ : ((Program C α → Set) → Program C α → Set) →
  Formula″ C α
ref_ : (Program C α → Set) → Formula″ C α
```

While this approach is accepted by Agda, it also has its downsides. The main drawback of this approach is that, as can be seen from the definitions of the constructors, it requires the Formula″ datatype to be parameterized by the container C and the return type α of the programs which it can be used to express properties of. Another disadvantage of this approach, that is also valid for Formula′, is that, while we would like to use the ref_ constructor only for referencing the formula variables which have been introduced by the fixed-point operators, the ref_ constructor in Formula′ and Formula″ can take any formula as its argument, not only the intended formula variables.

In our implementation we solve all of these issues by representing the formula variables which are introduced by the fixed-point operators using de Bruijn indices [10]. In order to achieve this, we add a new parameter to our definition of formulas – a natural number which denotes the number of formula variables, that can be referenced in the current formula. With this implementation the introduction of new formula variables by the fixed-point operators is not represented by a function. Instead, it is expressed by incrementing the corresponding parameter of the formula. Moreover, with this approach we can guarantee that the ref_ constructor can only be used to reference the formula variables which have been introduced by the fixed-point

operators. Our implementation of modal μ-calculus formulas in Agda is shown in Figure 2.

```
data Formula^fp (S : Set ℓ) : ℕ → Set ℓ where
  true false : ∀ {n} → Formula^fp S n
  ~_ : ∀ {n} → Formula^fp S n → Formula^fp S n
  _∧_ _∨_ _⇒_ : ∀ {n} → Formula^fp S n →
    Formula^fp S n → Formula^fp S n
  ⟨_⟩_ [_]_ : ∀ {n} → ActionFormula S →
    Formula^fp S n → Formula^fp S n
  μ_ ν_ : ∀ {n} → Formula^fp S (suc n) → Formula^fp S n
  ref_ : ∀ {n} → Fin n → Formula^fp S n
```

Figure 2. Definition of modal μ-calculus formulas in Agda

Now that we have defined the constructors for the fixed-point operators, we have to also define the semantics of those operators. We have already seen that the fixed-point operators represent the least fixed point and the greatest fixed point, respectively, of a function of type Formula^{fp} C α → Formula^{fp} C α. Therefore, in order to give the semantics of the fixed-point operators, we need a way of defining the fixed points of such functions. As we have already discussed, functions of type Formula^{fp} C α → Formula^{fp} C α can also be expressed as functions of type (Program C α → Set) → Program C α → Set, since we can think of formulas as predicates over programs. Fortunately, functions of type (Program C α → Set) → Program C α → Set are examples of so-called *indexed functors* and the least and greatest fixed points of indexed functors can be represented as follows:

```
record Mu {C : Container 0ℓ 0ℓ} {α : Set}
          (F : (Program C α → Set) →
               Program C α → Set)
          (x : Program C α) : Set where
  inductive; constructor mu^c
  field mu : F (Mu F) x

record Nu {C : Container 0ℓ 0ℓ} {α : Set}
          (F : (Program C α → Set) →
               Program C α → Set)
          (x : Program C α) : Set where
  coinductive; constructor nu^c
  field nu : F (Nu F) x
```

Unfortunately, with these definitions we face the same problem which we mentioned in § 3, when talking about the free monad, namely that they are not strictly positive and are therefore not accepted by Agda. Thus, in order to complete our implementation of the modal μ-calculus, we need a way to represent strictly-positive indexed functors as well as their fixed points.

6.2 Introducing Containerization

When we were faced with a similar problem in § 3, we solved it by using containers [1, 2]. However, containers can only be used when working with normal functors. In order to represent strictly-positive indexed functors, we need a more complex version of containers known as *indexed containers* [5]. In our implementation we use a data structure which is heavily inspired by the indexed containers presented by Altenkirch et al. [5]. It has the typical structure of a container as well as an associated extension function[13] which transforms it into an indexed functor. We have called this data structure Container^i and its implementation looks as follows:

```
record Container^i (C : Container ℓ1 ℓ2)
                   (α : Set ℓ3) : Set (ℓ1 ⊔ ℓ2 ⊔ ℓ3) where
  constructor _▷_
  field Shape^i : ℕ
        Position^i : Fin Shape^i → Program C α →
          List⁺ (Result C α)
```

Here, the datatype Result represents the new program obtained by applying a modality sequence to the input program (the one which is provided as an argument to the position) as well as the modal μ-calculus formula which this new program should satisfy – both of which are necessary, in order to encode the satisfaction relation. Furthermore, we provide an algorithm for translating a modal μ-calculus formula into an instance of Container^i. Our algorithm consists of the following steps:

1. Desugar all implications according to the formula $f_1 \Rightarrow f_2 = (\sim f_1) \vee f_2$;
2. Desugar all negations by replacing every negated formula with its dual, while counting the number of negations in front of each occurrence of the ref_ constructor;
3. Replace all ref_ constructors which have an odd number of negations in front of them with false;
4. Translate the resulting formula into disjunctive normal form (DNF);
5. Define the Shape^i of the container as the number of conjunctions in the formula;
6. Define the Position^i of the container for each shape (each conjunction in the formula) as a function which relates a given input program to a list of values of type Result, one for each element in the given conjunction.

Using this algorithm, we can translate any modal μ-calculus formula into an instance of Container^i. Thus, we can define the semantics of our fixed-point operators: they are represented as either the least fixed point or the greatest fixed

[13] For the exact implementation of the extension function please refer to our source code at https://github.com/ivanstodorov/modal-mu-calculus-for-free.

point, depending on which operator is used, of the instance of $\mathsf{Container}^{i}$ corresponding to the given formula.

Using this implementation of the modal μ-calculus, we can now express and prove more complex properties of programs. For example, the liveness property specifies that a program can never terminate or, put differently, that whenever a program executes some operation, there is always at least one more operation which has to be performed after it. Going back to our ATM example, we can express the liveness property as a functional requirement of our ATM's software using our implementation of the modal μ-calculus as follows:

```
property₄ : Formula^fp (Shape effect⁺) zero
property₄ = ν (([ true ] ref zero) ∧ (⟨ true ⟩ true))
```

However, this property does not hold for our ATM's software, since it halts, if an incorrect PIN code is provided. We can prove this as follows:

```
proof₄ : ATM^s ⊨ ~ property₄
proof₄ = mu^c (zero , tt , zero , λ { refl →
  mu^c (zero , tt , false , λ { refl →
  mu^c (suc zero , λ _ → ⊥-elim) }) })
```

This example serves as a demonstration of the capabilities of the modal μ-calculus. As we can see, it allows us to represent much more complex functional requirements compared to HML. However, this additional expressivity comes at the cost of complexity, since the modal μ-calculus and, in particular, the fixed-point operators can sometimes be complicated to use. Thus, in the next section we will explain how we can extend our current implementation of the modal μ-calculus with regular formulas – a feature which can be used to express some fixed points, such as the one used in the liveness property, in a simpler and cleaner way.

7 Regular Formulas

Regular formulas are a way of representing (possibly infinite) sequences of action formulas and their syntax can be represented using the following BNF grammar:

$$\mathsf{rf} \;::=\; \epsilon \mid \mathsf{af} \mid \mathsf{rf} \cdot \mathsf{rf} \mid \mathsf{rf} + \mathsf{rf} \mid \mathsf{rf}^{*} \mid \mathsf{rf}^{+}$$

where:

- the terminal symbol ϵ represents the empty sequence of action formulas;
- the terminal symbol af represents a sequence consisting of just a single occurrence of the given action formula;
- the non-terminal symbol rf · rf represents the concatenation of two sequences of action formulas;
- the non-terminal symbol rf + rf represents a choice between two sequences of action formulas;
- the non-terminal symbol rf * represents a sequence of zero or more occurrences of the given sequence of action formulas;
- the non-terminal symbol rf + represents a sequence of one or more occurrences of the given sequence of action formulas.

It should be noted that the non-terminal symbol rf + can be expressed using the non-terminal symbols rf · rf and rf *:

$$\mathsf{rf}^{+} = \mathsf{rf} \cdot (\mathsf{rf}^{*})$$

Thus, in our representation of regular formulas we do not need the non-terminal symbol rf +, since we can always add it later as syntactic sugar. With this in mind, we represent regular formulas in Agda using the following datatype:

```
data RegularFormula (S : Set ℓ) : Set ℓ where
  ϵ : RegularFormula S
  actF_ : ActionFormula S → RegularFormula S
  _·_ _+_ : (_ _ : RegularFormula S) → RegularFormula S
  _* : RegularFormula S → RegularFormula S
```

Next, in order to incorporate regular formulas into our definition of the modal μ-calculus from Figure 2, we need to change the constructors for the box and diamond modalities. However, instead of doing that, we will create a separate datatype $\mathsf{Formula}^{rf}$ to represent modal μ-calculus formulas with support for regular formulas:

```
data Formula^rf (S : Set ℓ) : ℕ → Set ℓ where
  true false : ∀ {n} → Formula^rf S n
  ~_ : ∀ {n} → Formula^rf S n → Formula^rf S n
  _∧_ _∨_ _⇒_ : ∀ {n} → Formula^rf S n →
    Formula^rf S n → Formula^rf S n
  ⟨_⟩_ [_]_ : ∀ {n} → RegularFormula S →
    Formula^rf S n → Formula^rf S n
  μ_ ν_ : ∀ {n} → Formula^rf S (suc n) → Formula^rf S n
  ref_ : ∀ {n} → Fin n → Formula^rf S n
```

The reason for this decision is that we do not want to adjust the satisfaction relation for the datatype $\mathsf{Formula}^{fp}$, in order to incorporate regular formulas into it. Instead, we want to desugar $\mathsf{Formula}^{rf}$ into $\mathsf{Formula}^{fp}$, thereby making it possible to reuse the satisfaction relation for $\mathsf{Formula}^{fp}$. In order to do this, we define a function desugar which desugars a $\mathsf{Formula}^{rf}$ into a $\mathsf{Formula}^{fp}$. For the constructors true, false, ~_, _∧_, _∨_, _⇒_, μ_, ν_ and ref_ the definition of this function is straightforward, since those simply get mapped to the corresponding constructors of $\mathsf{Formula}^{fp}$. Thus, the cases which we need to focus on are those for the box and diamond modalities: [rf] f and ⟨ rf ⟩ f, respectively. In those cases we first desugar the remaining formula f and then we use two separate functions $\mathsf{desugar\text{-}rf}^{b}$ and $\mathsf{desugar\text{-}rf}^{d}$ for the box and diamond modality, respectively, which desugar the regular formula rf, given the remaining formula (desugar f). Those functions can be defined by pattern matching on the regular formula rf and for the constructors ϵ, actF_, _·_ and _+_ their definitions look as follows:

```
desugar-rf^b : {S : Set ℓ} → {n : ℕ} → RegularFormula S →
               Formula^fp S n → Formula^fp S n
desugar-rf^b ϵ f = f
desugar-rf^b (actF af) f = [ af ] f
desugar-rf^b (rf₁ · rf₂) f =
  desugar-rf^b rf₁ (desugar-rf^b rf₂ f)
desugar-rf^b (rf₁ + rf₂) f =
  desugar-rf^b rf₁ f ∧ desugar-rf^b rf₂ f

desugar-rf^d : {S : Set ℓ} → {n : ℕ} → RegularFormula S →
               Formula^fp S n → Formula^fp S n
desugar-rf^d ϵ f = f
desugar-rf^d (actF af) f = ⟨ af ⟩ f
desugar-rf^d (rf₁ · rf₂) f =
  desugar-rf^d rf₁ (desugar-rf^d rf₂ f)
desugar-rf^d (rf₁ + rf₂) f =
  desugar-rf^d rf₁ f ∨ desugar-rf^d rf₂ f
```

As we can see, for those constructors the desugaring is straightforward, because we simply translate the constructors according to their meanings presented above. Unfortunately, we cannot use this approach for the constructor _*. For that case we need to use the following definitions [13]:

$$\langle\ \mathrm{rf}\ ^*\ \rangle\ \mathrm{f} = \mu\ \mathrm{X}\ .\ ((\langle\ \mathrm{rf}\ \rangle\ \mathrm{X}) \vee \mathrm{f})$$
$$[\ \mathrm{rf}\ ^*\ ]\ \mathrm{f} = \nu\ \mathrm{X}\ .\ (([\ \mathrm{rf}\ ]\ \mathrm{X}) \wedge \mathrm{f})$$

However, since we represent the variables which are introduced by the fixed-point operators using de Bruijn indices (instead of giving them explicit names), we need to use a slightly modified version of those definitions. In order to demonstrate the desugaring of the _* constructor, let us consider the following contrived formula:

$$\nu\ \mathrm{X}\ .\ \langle\ \mathrm{true}\ ^*\ \rangle\ \mathrm{X}$$

Using the datatype $\mathrm{Formula}^{rf}$ this formula can be expressed as follows:

```
ν ⟨ (actF true) * ⟩ ref zero
```

After desugaring the regular formula (actF true) *, which represents zero or more occurrences of the action formula true, this formula would look as follows:

```
ν μ ((⟨ actF true ⟩ ref zero) ∨ ref (suc zero))
```

As we can see, the regular formula (actF true) * is desugared by introducing a new fixed-point operator exactly like in the definition shown above. However, the ref zero, which in the original formula refers to the greatest-fixed-point operator in the beginning of the formula, becomes ref (suc zero) after the desugaring, in order to account for the new fixed-point operator which has been added.

Using this technique, we can desugar $\mathrm{Formula}^{rf}$ into $\mathrm{Formula}^{fp}$. Thus, we can define a satisfaction relation for $\mathrm{Formula}^{rf}$ by desugaring it into $\mathrm{Formula}^{fp}$ and reusing the satisfaction relation for $\mathrm{Formula}^{fp}$. Using regular formulas, we can now express the liveness property which we stated at the end of § 6.2 as follows:

```
property₅ : Formula^rf (Shape effect⁺) zero
property₅ = [ actF true * ] ⟨ actF true ⟩ true
```

As we can see, this representation is more compact and readable than the one we saw in the definition of $\mathrm{property}_4$ at the end of § 6.2. Moreover, since the regular formula gets desugared precisely into the fixed-point operator which was used to express this property at the end of § 6.2, we can use the same proof to show that our ATM's software does not satisfy this property.

```
proof₅ : ATM^s ⊨ ~ property₅
proof₅ = proof₄
```

This example illustrates the benefit of using regular formulas, namely that they make some very common use cases of the fixed-point operators simpler to write. With this we have implemented all features of the modal μ-calculus into our framework. Thus, in the next section we will introduce the first-order modal μ-calculus and we will briefly explain how we have implemented it in Agda, thereby adding even more expressivity to our framework.

8 The First-Order Modal μ-Calculus

The syntax of first-order modal μ-calculus formulas is described by the following BNF grammar:

$$\begin{aligned}\mathrm{f}\ ::=\ & \mathrm{true} \mid \mathrm{false} \mid \mathrm{B} \mid \sim \mathrm{f} \mid \mathrm{f} \wedge \mathrm{f} \mid \mathrm{f} \vee \mathrm{f} \mid \mathrm{f} \Rightarrow \mathrm{f} \mid \\ & \forall\ \mathrm{p{:}T}\ .\ \mathrm{f} \mid \exists\ \mathrm{p{:}T}\ .\ \mathrm{f} \mid \langle\ \mathrm{rf}\ \rangle\ \mathrm{f} \mid [\ \mathrm{rf}\ ]\ \mathrm{f} \mid \\ & \mu\ \mathrm{X}\ (\mathrm{p}_1{:}\mathrm{T}_1{:=}\mathrm{v}_1,\ldots,\mathrm{p}_n{:}\mathrm{T}_n{:=}\mathrm{v}_n)\ .\ \mathrm{f} \mid \\ & \nu\ \mathrm{X}\ (\mathrm{p}_1{:}\mathrm{T}_1{:=}\mathrm{v}_1,\ldots,\mathrm{p}_n{:}\mathrm{T}_n{:=}\mathrm{v}_n)\ .\ \mathrm{f} \mid \mathrm{X}(\mathrm{v}_1,\ldots,\mathrm{v}_n)\end{aligned}$$

From the grammar we can see that the first-order modal μ-calculus extends the modal μ-calculus with three additional features:

1. Any boolean expression can be a formula;
2. Universal and existential quantifiers are supported;
3. The least- and greatest-fixed-point operators are parameterized.

It should be noted that the first two of those features have also been added to action formulas. However, their implementation for action formulas is exactly the same as that for first-order modal μ-calculus formulas. Therefore, in this section we will only focus on the latter. Furthermore, for the sake of brevity, we will only describe the intuition behind our implementation of those features without going into the implementation details.

The simplest new feature is that we allow any boolean expression to be a formula. This is represented by the terminal symbol B in the grammar shown above and such a formula holds, iff the boolean expression evaluates to true. To incorporate this feature into our implementation, we can

add a new constructor to Formula^{rf} which takes a single parameter, that is of type ⊤, if the boolean value evaluates to true, or of type ⊥, otherwise. In our implementation we provide a more general version of this feature: we add another parameter to our definition of Formula^{rf}, a level ℓ_1, and we add the following new constructor to Formula^{rf}:

```
val_ : ∀ {n} → Set ℓ₁ → Formula^rf S ℓ₁ n
```

Using this new constructor, we can construct a formula using any element of Set ℓ_1. This feature increases the expressivity of our framework, because it allows us to reason about more than just effect operations. For example, we can use it together with the next feature which we will introduce – universal and existential quantifiers, to model the statement "there exists a natural number n, such that n > 42". We can represent this statement as a formula using the existential quantifier and the val_ constructor to express the condition n > 42. Furthermore, the formula val X holds for any program, iff we can provide a value of type X. Thus, for the example, that we just introduced, in order to prove that the formula holds for a given program, we would need to give a natural number n as well as a proof that n > 42.

The next new feature of the first-order modal μ-calculus is the addition of universal and existential quantifiers which are represented by the non-terminal symbols ∀ p:T . f and ∃ p:T . f, respectively, in the BNF grammar shown above. Unfortunately, when it comes to these two operators, our implementation does not match their definitions in the grammar. Initially, we attempted to implement the exact behavior from the grammar by adding constructors to Formula^{rf} to represent the universal and existential quantifiers and converting all formulas into prenex normal form (PNF), in order to separate the quantifiers from the remainder of the formula, before continuing to transform the remainder of the formula using the techniques which we have described thus far. However, this turned out to be impossible, since the conversion of formulas into PNF could not be automated. Instead, we settled for a different, but equally expressive, approach, namely requiring all formulas to be written in PNF in the first place, thereby eliminating the need to convert formulas into PNF automatically. It should be noted, that it was necessary for us to have the formulas in PNF; otherwise, we would not have been able to use the containerization technique discussed in § 6.2. Thus, in our implementation we use a new datatype called Quantified, that represents a formula in PNF – it can introduce any number of universal and existential quantifiers, before defining the actual formula.

The last new feature of the first-order modal μ-calculus is that the least- and greatest-fixed-point operators are now parameterized. To represent this in our implementation, we add two additional arguments to the constructors for the least- and greatest-fixed-point operators: a list denoting the types of the parameters and a list containing their initial values. Furthermore, we modify the ref_ constructor to take an additional argument: a list of values which will be given to the parameters of the fixed-point operator which is being referenced. Moreover, in order to keep track of what types of parameters each fixed-point operator expects, we switch from using standard de Bruijn indexing, represented using natural numbers, to indexing formulas by a Vec (List (Set ℓ_1)) n, where the length n of the vector denotes the number of fixed-point operators which we can reference and the list at each index of the vector represents the types of the parameters expected by the corresponding fixed-point operator. Finally, we introduce a new datatype called Parameterized, that represents the formulas which are passed to the fixed-point operators – parameterized formulas in PNF.

Having said all of this, our full definition of first-order modal μ-calculus formulas looks as follows, where $\text{Formula}^{rf\prime}$ is defined by extending Formula^{rf} with the new features discussed in this section:

```
data Quantified
       (S : Set ℓ) (ℓ₁ : Level) (xs : Vec (List (Set ℓ₁)) n)
       :  List (Set ℓ₁ ⊎ Set ℓ₁) → Set (ℓ ⊔ (suc ℓ₁)) where
  formula_ : Formula^rf′ S ℓ₁ xs → Quantified S ℓ₁ xs []
  ∀(_)_ : ∀ {αs} (α : Set ℓ₁) →
          (α → Quantified S ℓ₁ xs αs) →
          Quantified S ℓ₁ xs (inj₁ α :: αs)
  ∃(_)_ : ∀ {αs} (α : Set ℓ₁) →
          (α → Quantified S ℓ₁ xs αs) →
          Quantified S ℓ₁ xs (inj₂ α :: αs)

data Parameterized
       (S : Set ℓ) (ℓ₁ : Level) (xs : Vec (List (Set ℓ₁)) n)
       :  List (Set ℓ₁) → Set (ℓ ⊔ (suc ℓ₁)) where
  quantified_ : ∀ {αs} → Quantified S ℓ₁ xs αs →
                Parameterized S ℓ₁ xs []
  _↦_ : ∀ {αs : List (Set ℓ₁)} → (α : Set ℓ₁) →
        (α → Parameterized S ℓ₁ xs αs) →
        Parameterized S ℓ₁ xs (α :: αs)

Formula : (S : Set ℓ) → (ℓ₁ : Level) →
          (αs : List (Set ℓ₁ ⊎ Set ℓ₁)) → Set (ℓ ⊔ (suc ℓ₁))
Formula S ℓ₁ αs = Quantified S ℓ₁ [] αs
```

9 Related Work

Propositional dynamic logic (PDL) is a kind of logic which was originally introduced by Vaughan Pratt [18] for the purpose of reasoning about computer programs. Later, HML was designed by Matthew Hennessy and Robin Milner [14], based on the PDL of Pratt, with the goal of describing the behavior of concurrent programs. And after that, the modal μ-calculus, which is more expressive than both the PDL of Pratt and HML, was first introduced by Dexter Kozen [15]. Since then, the modal μ-calculus has been widely used in the

field of process theory, in order to reason about the behavior of labelled transition systems. This widespread adoption is what led to development of the tool mCRL2 [13] which uses model checking to verify properties, expressed using the first-order modal μ-calculus, of labelled transition systems. The adoption of the first-order modal μ-calculus in our work was inspired by mCRL2. However, in our work we attempt to use the first-order modal μ-calculus in a novel context, namely to verify properties of programs defined using algebraic effects.

In a different related line of work, it has been shown that it is possible to formalize modal logics in a proof assistant [9]. However, such works do not make any connection between the formalized modal logics and their meaning for computer programs. Thus, our work differs from those by the fact that we provide semantics for the first-order modal μ-calculus which directly link it to computer programs.

Another related, although less closely, topic of research is session types. Session types can be used to enforce certain properties of communication channels in a distributed setting, such as the order in which messages are sent and received through a given channel. Although session types can be used to enforce some order among the operations in a distributed program, which is similar to what can be accomplished using our framework, it should be noted that session types are typically applied to distributed programs. In contrast, our framework provides a logic for verifying properties of sequential programs. Therefore, at present, our work is clearly separated from session types. However, if we extend our framework, such that it can be used to reason about concurrent and distributed programs, then we could use it to express properties, similar to those enforced by session types.

Recent work by Dal Lago and Ghyselen [8] extends techniques due to Ong [17] for model-checking higher-order programs. This extension lets them automatically verify *monadic second-order logic* propositions about programs involving algebraic effects and handlers. The work of Dal Lago and Ghyselen [8] also demonstrates that the model-checking problem for programs involving algebraic effects and handlers is, in general, undecidable. Rather than using model checking, the goal of our work is to allow programmers working in a dependently-typed language, such as Agda, to assert and verify functional properties of effectful programs.

In another contemporary line of work, Swierstra and Baanen [20] demonstrate how we can derive effectful programs directly from a given functional specification, represented by a pre- and post-condition. The advantage of this approach is that programs derived in this way are guaranteed to comply with the provided functional specification and therefore there is no need for additional verification. While our framework does not currently support such features, an interesting prospect would be to explore whether it is possible to use first-order modal μ-calculus formulas, such as the ones shown in this paper, as the pre- and post-conditions for such program derivations.

10 Conclusion and Future Work

We have presented an embedding of the first-order modal μ-calculus in Agda for stating and proving properties of programs with algebraic effects. This embedding lets us verify properties of executable programs in Agda directly, rather than first extracting a separate model. Furthermore, our work demonstrates how to reason about programs in a proof assistant using a dynamic logic, namely the first-order modal μ-calculus. While our framework uses Agda, we believe that our results should be reproducible in other proof assistants which support the necessary features (such as coinduction).

In future work, we would like to explore the following:

- Extending our framework to reasoning about concurrent programs rather than "just" sequential ones.
- Reducing the effort required to manually prove that a given formula holds for a given program, which currently requires knowledge of the satisfaction relation setup.
- Improving performance. Our experience suggests that type checking can be slow for complex formulas or large programs. A main culprit is the large number of transformations which we use to define our satisfaction relation; in particular, the containerization discussed in § 6.2. We would like to explore whether a different approach to representing strictly-positive functors (e.g., descriptions [7]) might reduce complexity and improve performance.
- Applying our approach to real-world programs, such as cyber-physical systems.
- Defining a *refinement calculus* based on the first-order modal μ-calculus which could allow us to "calculate" programs that satisfy their functional requirements, similar to the work of Swierstra and Baanen [20].

Acknowledgments

We thank the anonymous reviewers for their comments. We would also like to thank Jesper Cockx for his feedback on our work and on an earlier version of this paper.

References

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of Containers. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2620)*, Andrew D. Gordon (Ed.). Springer, 23–38. https://doi.org/10.1007/3-540-36576-1_2

[2] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

[3] Andreas Abel and James Chapman. 2014. Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and

Sized Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 51–67. https://doi.org/10.4204/EPTCS.153.4

[4] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 27–38. https://doi.org/10.1145/2429069.2429075

[5] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S095679681500009X

[6] Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press, Inc., USA.

[7] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. *ACM Sigplan Notices* 45, 9 (2010), 3–14.

[8] Ugo Dal Lago and Alexis Ghyselen. 2024. On Model-Checking Higher-Order Effectful Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 2610–2638. https://doi.org/10.1145/3632929

[9] Ana de Almeida Borges. 2022. Towards a Coq Formalization of a Quantified Modal Logic. In *Proceedings of the 4th International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2022) affiliated with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022), Haifa, Israel, August 11, 2022 (CEUR Workshop Proceedings, Vol. 3326)*, Christoph Benzmüller and Jens Otten (Eds.). CEUR-WS.org, 13–27. https://ceur-ws.org/Vol-3326/ARQNL2022_paper1.pdf

[10] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)*, Vol. 75. Elsevier, 381–392.

[11] Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *ACM SIGPLAN international conference on Functional Programming (ICFP)*. 143–155. https://doi.org/10.1145/2034773.2034796

[12] Edsger Wybe Dijkstra. 1970. Notes on Structured Programming. (April 1970). EWD249 http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[13] Jan Friso Groote and M Mousavi. 2013. *Modelling and analysis of communicating systems*. Technische Universiteit Eindhoven.

[14] Matthew Hennessy and Robin Milner. 1980. On observing nondeterminism and concurrency. In *International Colloquium on Automata, Languages, and Programming*. Springer, 299–309.

[15] Dexter Kozen. 1983. Results on the propositional μ-calculus. *Theoretical computer science* 27, 3 (1983), 333–354.

[16] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. http://research.microsoft.com/users/lamport/tla/book.html

[17] C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 81–90. https://doi.org/10.1109/LICS.2006.38

[18] Vaughan R Pratt. 1976. Semantical considerations on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE, 109–121.

[19] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[20] Wouter Swierstra and Tim Baanen. 2019. A predicate transformer semantics for effects (functional pearl). *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–26.

[21] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

Received 2024-06-03; accepted 2024-07-10