Modal μ-Calculus for Free

Version of June 21, 2024

Ivan Todorov

Modal μ-Calculus for Free

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ivan Todorov
born in Varna, Bulgaria

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2024 Ivan Todorov.

Modal μ-Calculus for Free

Author: Ivan Todorov
Student ID: 5077176

Abstract

The process of using formal verification, in order to ensure that a piece of software meets it functional requirements consists of three main steps: designing a model of the given piece of software, translating the functional requirements, which the piece of software must satisfy, into properties of said model and verifying that the model satisfies those properties. Traditionally, regardless of whether the piece of software is developed based on a predesigned model or the piece of software is developed first and its model is designed after that, the piece of software and its model are two separate entities. Therefore, aside from checking that the model satisfies its properties, it must also be verified that the model accurately represents the given piece of software. While this task may initially seem simple, it gets progressively more difficult, as the piece of software becomes more complex. And, if it turns out that the model is not accurate, then the entire formal verification process is invalid, since it does not provide any guarantees about the actual piece of software. In this thesis we present a solution to this problem: a way of modelling sequential effectful programs, such that the resulting models can be directly translated into runnable programs, thereby guaranteeing the models' accuracy. We achieve this by using algebraic effects, in order to model sequential effectful programs as instances of the coinductive free monad, that could then be translated into runnable pieces of software by applying the necessary effect handlers. Furthermore, we demonstrate that it is possible to express functional requirements as properties of such models using the first-order modal μ-calculus, a fixed-point dynamic logic which has previously been used to reason about labelled transition systems (e.g. in mCRL2).

Thesis Committee:

Chair:	Dr. J. Cockx, Faculty EEMCS, TU Delft
Committee Member:	Dr. C.B. Poulsen, Faculty EEMCS, TU Delft
University Supervisor:	Dr. E. Demirović, Faculty EEMCS, TU Delft

Contents

Chapter 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 [1] 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)?

If we want to use model checking, then we can use a tool like TLA+ [2] or mCRL2 [3], in order to verify the functional requirements of our software. For example, in order to use mCRL2, we have to model our software as a labelled transition system and represent its functional requirements as properties of that labelled transition system using the *modal μ-calculus* [4], a dynamic fixed-point logic. Then, mCRL2 uses model checking to check whether the properties, that we have defined, hold for the given model. However, a downside of this approach is that the verification will be done on the model (in the case of mCRL2, the labelled transition system) rather than on the software itself. Thus, since the model is defined separately from the software, there is a risk of the model being incorrect, meaning that it does not correctly represent the behavior of the software. In such cases, verifying the model is useless, as it provides no guarantees for the actual software.

Recent work by Lago and Ghyselen [5] explores an alternative approach, namely model-checking programs involving algebraic effects and handlers. This approach makes it possible to verify functional requirements of effectful programs *directly*, without relying on models which have to be defined separately from the programs. However, as Lago and Ghyselen [5] demonstrate, this model-checking problem is, in general, undecidable.

In this thesis we explore a different strategy: we model effectful programs in the dependently-typed language Agda[1] as instances of the coinductive free monad using algebraic effects and we develop an embedding of the modal μ-calculus in Agda which makes it possible to express and prove functional requirements of those programs. With this approach we can first verify that the necessary functional requirements are satisfied by a given model and then use effect handlers to obtain a runnable program from the verified model, thereby guaranteeing that the model is correct and, by extension, that the runnable program also satisfies the functional requirements. While our work does not rely on an automated verification technique, such as model checking, instead requiring programmers to manually prove that the

[1] `https://agda.readthedocs.io`

functional requirements are met, it 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 work are transferable to other dependently-typed languages, such as Idris,[2] Coq[3] or Lean.[4] However, we do rely extensively on Agda's support for *dependent pattern matching*, *guarded coinduction*, and *copattern matching* [6], which may necessitate alternative encodings in other languages.

The main contribution of this thesis 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 Chapter 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 Chapter 5), we add support for *action formulas* – a way of representing a set of operations.
- Next (in Chapter 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 Chapter 7), we add support for regular formulas – a way of representing a (possibly infinite) sequence of action formulas.
- After that (in Chapter 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 remainder of this thesis is structured as follows: first (in Chapter 2), we discuss examples of expressing simple functional requirements using the first-order modal μ-calculus; then (in Chapter 3), we show how to define the *coinductive free monad* in Agda, which lets us represent possibly-non-terminating sequential effectful programs; next (in Chapters 4 to 8), we present our embedding of the first-order modal μ-calculus in Agda;[5] after that (in Chapter 9), we demonstrate how we can use effect handlers, in order to obtain runnable programs from the models which we have verified; finally, we discuss related work (in Chapter 10) and conclude and discuss future work (in Chapter 11).

[2] `https://idris2.readthedocs.io/en/latest/`

[3] `https://coq.inria.fr/`

[4] `https://lean-lang.org/`

[5] The full source code of our work can be found on GitHub.

Chapter 2

Using the First-Order Modal μ-Calculus

In this chapter we are going to look at a number of examples of how the first-order modal μ-calculus can be used to express the functional requirements of a sequential effectful program. However, in order to do this, we first need to have a sequential effectful program which we would like to prove the functional requirements of. For example, imagine that you are tasked with designing and implementing the software for an ATM. And, for the sake of simplicity, let us only focus on one of the functionalities of an ATM, namely, allowing its users to view the balance of their bank account. Let us assume that we would like our ATM to work as follows:

1. To start using the ATM, a user must insert their bank card into the ATM;
2. Then, the user has to provide their bank card's PIN code;
3. Next, the ATM checks the PIN code;
4. If the PIN code is correct, the user's 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.

From this description it becomes clear that the ATM's software is sequential, since the ATM can only be used by one person at a time. Furthermore, it is obvious that it contains a number of side effects (executing IO operations, throwing exceptions, etc.). Thus, we can conclude that the ATM's software is some sequential effectful program. We can think of it as a short sequence of operations which is executed a (possibly infinite) number of times (whenever a user inserts their bank card into the ATM) and only halts, if an exception is thrown due to an incorrect PIN code. Therefore, our ATM's software can be represented in pseudocode as follows:

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

Now, after we have defined our sequential effectial program, let us come up with some functional requirements for it and see how they can be expressed using the first-order modal μ-calculus. One intuitive functional requirement which the ATM's software must satisfy is that, when the ATM's first user starts interacting with it, they must be able to provide their

bank card's PIN code. The same must also be true for all subsequent users of the ATM. However, for the sake of simplicity, let us only focus on the first user for now. In that case, this requirement can also be stated as follows:

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

This 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:

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

For some program (some list of operations) the formula $\langle\ \mathsf{A}\ \rangle\ \mathsf{F}$, where A is some operation and F is some formula, can be read as "the first operation in the list is A and the remaining formula F must hold 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 where the operation returns true (the ***then*** branch), and one for false. The formula true holds for all programs. Thus, if we look at the pseudocode of our ATM's software, we can verify that it satisfies the dynamic formula $\langle\ \mathsf{getPIN}\ \rangle\ \mathsf{true}$, since the first operation in it is indeed getPIN and the formula true trivially holds for the remaining program.

Another functional requirement which the ATM's software must satisfy is that, when the ATM's first user starts interacting with it, they must not be able to directly view their bank account balance. Once again, this property must also be true for all subsequent users of the ATM, but for now we will only focus on the first user. Therefore, this requirement can also be stated as:

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

This is a modal statement as well, because it talks about what must not be possible in a particular situation, namely when the program is first executed. Therefore, it can be represented using dynamic logic as follows:

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

For some program (some list of operations) the formula $[\ \mathsf{A}\ ]\ \mathsf{F}$, where A is some operation and F is some formula, can be read as "if the first operation in the list is A, then the remaining formula F must hold for all possible continuations of the program". Furthermore, the formula false does not hold for any program. Therefore, for any operation A the only scenario in which the formula $[\ \mathsf{A}\ ]\ \mathsf{false}$ holds for some program is when the first operation in that program is not A, since, if the first operation in that program is A, then the formula false must hold for all possible continuations of the program, which is impossible. That being said, if we look at the psudocode of our ATM's software, we can verify that it satisfies the dynamic formula $[\ \mathsf{showBalance}\ ]\ \mathsf{false}$, because the first operation in it is not showBalance.

We can also add functional requirements like the previous one for other operations as well, not just for the showBalance operation. For example, another requirement could be:

> At the start of the program it must not be possible to execute the correctPIN operation (since no PIN code has been provided yet).

or:

> At the start of the program it must not be possible to execute the throwException operation (since in that case the ATM would be useless).

In fact, we can combine all of those into the following requirement:

> At the start of the program it must not be possible to execute any operation, other than the getPIN operation.

This is a modal statement, since it describes what must not be possible in a particular situation, namely when the program is first executed. However, at first, it seems that this requirement is not as straighforward to represent using dynamic logic as the previous ones. One way of expressing it would be as a conjunction of formulas like the one for the previous requirement (one formula for each operation which we want to say is impossible):

$$[\ \mathsf{correctPIN}\ ]\ \mathsf{false} \wedge [\ \mathsf{showBalance}\ ]\ \mathsf{false} \wedge [\ \mathsf{throwException}\ ]\ \mathsf{false}$$

Unfortunately, this representation has the downside that it includes all operations which are part of the ATM's software, other than the getPIN operation. This means that, if at any point the operations in the ATM's software change (e.g. some new operations are added, an operation is renamed, etc.), then this requirement will also have to be changed. Fortunately, this flaw can be remedied using *action formulas* [3]. Action formulas are a way of representing and reasoning about sets of operations. For example, the action formula true represents the set of all possible operations, while the action formula false represents the empty set of operations. Furthermore, any single operation can also be viewed as an action formula which represents the singleton set which only contains that operation. It is also possible to construct action formulas which represent the complement of a set of operations, the union of two sets of operations and the intersection of two sets of operations using $\overline{\mathsf{X}}$, $\mathsf{X} \cup \mathsf{Y}$ and $\mathsf{X} \cap \mathsf{Y}$, respectively, where X and Y are action formulas. Therefore, using action formulas the functional requirement that at the start of the program it must not be possible to execute any operation, other than the getPIN operation, can be expressed as follows:

$$[\ \overline{\mathsf{getPIN}}\ ]\ \mathsf{false}$$

This representation is a lot more compact than the alternative presented above and, more importantly, it does not need to explicitly mention all operations which are part of the ATM's software, except the getPIN operation, as was the case with the other alternative. This means that even if at some point the operations in the ATM's software change, this requirement will not have to be changed (unless the getPIN operation is changed). And, finally, if we look at the pseudocode of our ATM's software we can verify that it satisfies this requirement, because the first operation in it is indeed getPIN, meaning that any operation other than getPIN is not the first operation in the program.

All of the functional requirements which we have discussed so far have been relatively simple to represent using the first-order modal μ-calculus and to verify. A big reason for this is that so far we have only looked at requirements which talk about what must and must not be possible, when the ATM's first user starts interacting with it or, put differently, what must and must not be possible at the start of the program. However, the first-order modal μ-calculus can also be used to express more complex requirements. For example, a very important functional requirement for our ATM's software is that it is not possible to view your bank account balance without first providing your bank card's PIN code. This requirement can also be stated as:

> It is not possible to execute the showBalance operation for some user, before executing the getPIN operation for that user.

It is important to note, that in this case we do not talk only about the first user of the ATM, as we have done with all requirements so far, but for any user of the ATM. Thus, this requirement cannot be expressed using the features of the first-order modal μ-calculus which we have seen so far. If fact, although this requirement is relatively straightforward to verify by inspecting the pseudocode of the ATM's software, it requires one of the most advanced

features of the first-order modal μ-calculus, in order to be expressed, namely the *parameterized fixed-point operators* [3]. The first-order modal μ-calculus supports both a parameterized least-fixed-point operator (μ) and a parameterized greatest-fixed-point operator (ν) which can be used as follows:

$$\mu\ \mathsf{X}\ (\mathsf{p}_1{:}\mathsf{T}_1{:=}\mathsf{v}_1,\ldots,\mathsf{p}_n{:}\mathsf{T}_n{:=}\mathsf{v}_n)\ .\ \mathsf{f}$$
$$\nu\ \mathsf{X}\ (\mathsf{p}_1{:}\mathsf{T}_1{:=}\mathsf{v}_1,\ldots,\mathsf{p}_n{:}\mathsf{T}_n{:=}\mathsf{v}_n)\ .\ \mathsf{f}$$

where X is some name, p_i is the name of the i-th parameter, that has type T_i and initial value v_i, and f is some formula which can reference X, but only in positive positions, meaning that there must be an even number of negations in front of every reference to X in f, and with a well-typed list of values for the parameters of X. Using this feature of the first-order modal μ-calculus we can express the functional requirement that it is not possible to execute the showBalance operation for some user, before executing the getPIN operation for that user, as follows:

$$\begin{aligned}\nu\ \mathsf{X}\ (\mathsf{b}{:}\mathsf{bool}{:=}\mathsf{false})\ .\ &[\ \overline{\mathsf{getPIN}\cup\mathsf{showBalance}}\ ]\ \mathsf{X}(\mathsf{b})\ \wedge\\ &[\ \mathsf{getPIN}\ ]\ ((\mathsf{not}\ \mathsf{b})\ \wedge\ \mathsf{X}(\mathsf{true}))\ \wedge\\ &[\ \mathsf{showBalance}\ ]\ (\mathsf{b}\ \wedge\ \mathsf{X}(\mathsf{false}))\end{aligned}$$

First, it should be noted that this definition uses the greatest-fixed-point operator, meaning that it represents a possibly infinite sequence of operations. This is necessary, since the property must hold for all users of the ATM, the number of which could be infinite. Moreover, the greatest-fixed-point operator in this definition has a single parameter of type Bool. This parameter is used to indicate, whether the getPIN operation has been executed for the current user. Thus, its initial value is false. Then, this property states that executing any operation which is not getPIN or showBalance is inconsequential, meaning that it does not change the value of the boolean parameter; that executing the getPIN operation is possible only if it has not already been executed (otherwise $\neg$ b would not hold) and results in the value of the boolean parameter being changed to true; and that executing the showBalance operation is possible only if the getPIN operation has already been executed (otherwise b would not hold) and results in the value of the boolean parameter being changed to false.

While this last property is much more complex to write and understand than any of the ones which we have discussed so far, it serves as an example of the capabilities of the first-order modal μ-calculus and the possible complexity of the functional requirements which can be expressed using it. In the remainder of this thesis we will first demonstrate how we can write programs similar to our ATM's software in Agda by defining them as instances of the coinductive free monad; subsequently, we will present an Agda embedding of the first-order modal μ-calculus which makes it possible to express and prove properties of effectful programs, like the ones discussed in this chapter.

Chapter 3

Modelling Effectful Programs

In this chapter we are going to explain how we can model sequential effectful programs in Agda. First (in Section 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 Section 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 Section 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 Section 3.4), we describe how we can define smart constructors for our effects' operations which significantly reduce the notational overhead, that is typically introduced when combining effects.

3.1 The Inductive Free Monad

The free monad is a data structure which models effectful computations and it is typically defined as follows:[1]

```
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
```

where the functor F is a so-called *signature functor* [7], that represents the types of operations which can occur in the computation, and α is the type of the final result. However, this definition is not *strictly positive*[2] and is therefore not accepted by Agda. Thus, in our implementation we represent effects using *containers* [8], [9] – 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
```

where the Shape is a datatype which represents the operations, that are part of the effect, and the Position defines the type of the result of each operation. Furthermore, the *extension* of a container is a function which maps a container to a strictly-positive functor and is defined in Agda's standard library as follows:

[1] A universe polymorphic definition of this datatype is also possible.

[2] https://agda.readthedocs.io/en/v2.6.4.3-r1/language/data-types.html#strict-positivity

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

Using these definitions, a definition of the free monad which uses the Container datatype is also provided in Agda's standard library:

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

Using this inductive definition of the free monad it is possible to model any finite sequential effectful program. For example, if we imagine an even simpler version of the ATM described in Chapter 2, one which only works for a single user, we could represent it as an instance of the inductive free monad. In order to do this, we first need to define the effect which is used by our ATM's software as a container: We can use the following datatype to represent the operations which are part of the effect:

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

Then, we can define the effect itself by using the datatype EffectShape as the Shape and defining the Position for each operation.

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

Using this effect, we can define our model of the ATM's software as follows:

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

The program described by this model is very similar to the one discussed in Chapter 2. The only difference is that, after showing the balance of a given user, it halts, instead of starting from the beginning. Thus, an ATM which uses this software would only work for a single user: if they provide the correct PIN code, then their bank account balance is shown and the program halts, or, if they provide an incorrect PIN code, then an exception is thrown and the program halts. However, if we want to model the version of the ATM's software described in Chapter 2, we cannot use this inductive implementation of the free monad, because that software is not a finite program, as it could run forever, if every user provides their correct PIN code. Thus, in order to represent this potentially infinite version of the ATM's software, we would instead need to use a coinductive implementation of the free monad.

3.2 Representing Recursion: The Coinductive Free Monad

Intuitively, in order to define the coinductive free monad, we would need to use a coinductive datatype which has exactly the same constructors as the inductive datatype _*_. However, Agda does not support coinductive datatypes, only coinductive record types. Thus, we first need to define a record type which is isomorphic to the inductuve datatype _*_. In order to achieve this, we can use 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
```

The difference between the datatype 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. Thus, we can use the datatype Free to define the following record type:

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

The IndFree record type is isomorphic to the datatype _*_, since it only has a single field which is an instance of the datatype Free and uses IndFree itself as the value of its parameter F, meaning that the call to the parameter F in the impure constructor of Free will actually be a recursive call to IndFree. Since we have shown that we can define the inductive version of the free monad as a record type, we can now also define the coinductive version of the free monad using a record type which has the same structure as IndFree, but is coinductive:

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

Using this conindctive implementation of the free monad we can model all sequential effectful programs, even infinite ones. Thus, we can now provide our definition of a program – an instance of the coinductive free monad:

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

Next, let us look at how the software of the ATM discussed in Chapter 2 can be modelled using this approach:

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

As we can see, this model is very similar to ATM^f, the model of the finite version of the ATM's software. Thus, we can conclude that, when it comes to modelling effectful programs, the use of the coinductive free monad adds a lot expressivity compared to using the inductive free monad, but it does not significantly increase the complexity of the models.

3.3 Composing Effects

So far, when we modelled the software of our ATM, we represented the operations which it can perform using a single effect. However, realistically, the ATM's software contains a few different effects, rather than just one. Thus, a more accurate model would be one which represents different effects separately and indicates that the ATM's software uses a combination of all of those effect. For example, the container effect which we used to model the ATM's software can be split into the following three separate containers, representing three separate effects.

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

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

data VerificationShape : Set where
  correctPIN : ℕ → VerificationShape

verificationEffect : Container 0ℓ 0ℓ
Shape verificationEffect = VerificationShape
Position verificationEffect (correctPIN _) = Bool

data ExceptionShape : Set where
  throwException : ExceptionShape

exceptionEffect : Container 0ℓ 0ℓ
Shape exceptionEffect = ExceptionShape
Position exceptionEffect _ = ⊥
```

However, all implementations of the free monad which we have shown so far, are parameterized by just one container, meaning that they can only use a single effect. Thus, in order to use all three containers shown above to represent the different effects which the ATM's software uses, we need some way of combining those three containers into one container which is isomorphic to the container effect. Fortunately, in Agda we can do this using the container combinator _⊎_ which is defined as follows:

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

Using this combinator we can define the effect which is used by the ATM's software as the combination of the three simpler effects as follows:

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

However, if we want to use the container effect$^+$ in our model of the ATM's software, then we have to modify our model as follows:

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

As we can see, this model is more difficult to read than the previous one, due to all of the injections which have to be added for each operation. Furthermore, we can imagine what would happen, if we try to model a more complex piece of software, or one which uses more effects, using this approach: there will be even more injections and, as a result, the model will be even more difficult to read and understand. Thus, in order to make this approach viable, we need some way to get rid of those injections.

3.4 Adding Smart Constructors

We can get rid of the injections by introducing smart constructors [7]. In order to do this, we first have to define the following type class which indicates that there is some injection from a container C_1 into a container 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
```

Then, we have to define the following three instances for this type class, corresponding to the ones described by Swierstra [7].

```
instance
  inj-id : C :<: C
  injS ⦃ inj-id ⦄ = id
  projP ⦃ inj-id ⦄ = id

  inj-left : C₁ :<: (C₁ ⊎ C₂)
  injS ⦃ inj-left ⦄ = inj₁
  projP ⦃ inj-left ⦄ = id

  inj-right : ⦃ C₁ :<: C₂ ⦄ → C₁ :<: (C₃ ⊎ C₂)
  injS ⦃ inj-right ⦃ inst ⦄ ⦄ = inj₂ ∘ injS inst
  projP ⦃ inj-right ⦃ inst ⦄ ⦄ = projP inst
```

It should be noted that there are cases in which the second and third instance overlap and, by default, Agda does not allow this behavior. However, if we explictly add the option `--overlapping-instances`, then we can use the defined instances. Now, using the type class, we can define the following smart constructors for all of the operations which are used by the ATM's software.

```
getPIN^s : (C : Container l₁ l₂) → ⦃ IOEffect :<: C ⦄ → Shape C
getPIN^s _ ⦃ inst ⦄ = injS inst getPIN

showBalance^s : (C : Container l₁ l₂) → ⦃ IOEffect :<: C ⦄ → Shape C
showBalance^s _ ⦃ inst ⦄ = injS inst showBalance
```

```
correctPIN^s : (C : Container l1 l2) → ⦃ verificationEffect :<: C ⦄ → ℕ → Shape C
correctPIN^s _ ⦃ inst ⦄ = injS inst ∘ correctPIN

throwException^s : (C : Container l1 l2) → ⦃ exceptionEffect :<: C ⦄ → Shape C
throwException^s _ ⦃ inst ⦄ = injS inst throwException
```

Finally, using the smart constructors, we can refine our model of the ATM's software as follows:

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

As we can see, the addition of smart constructors allows us to completely remove all of the explicit injections. Their only downside is that each smart constructor requires an explicit argument denoting the container which the operation should be injected into. However, that is more convenient than having to explicitly write all of the injections for each operation, especially since that container is always the same throughout the entire program, namely the container which is given in the program's type signature. Therefore, we recommend the use of such smart constuctors, when using our framework and we will be using them in all of the examples throughout the remainder of this thesis.

Chapter 4

Hennessy-Milner Logic

Now, that we have explained how we can model any sequential effectful program using the coinductive free monad, it is time to show how we can formalize and use the first-order modal μ-calculus in Agda. We are going to start our discussion by only looking at a subset of the first-order modal μ-calculus – a simple dynamic logic, known as Hennessy-Milner logic (HML) [10]. The syntax of HML formulas can be represented using a grammar in Backus-Naur form (BNF) as follows:

$$\mathsf{f} ::= \mathsf{true} \mid \mathsf{false} \mid \neg\, \mathsf{f} \mid \mathsf{f} \wedge \mathsf{f} \mid \mathsf{f} \vee \mathsf{f} \mid \mathsf{f} \rightarrow \mathsf{f} \mid \langle\, \alpha \,\rangle\, \mathsf{f} \mid [\, \alpha \,]\, \mathsf{f}$$

As we can see from this grammar, HML contains the formulas true, which holds for all programs, and false, which does not hold for any program. Moreover, it supports all of the standard connectives from predicate logic, namely negation, conjunction, disjunction and implication, as well as the diamond ($\langle\, \alpha \,\rangle\, \mathsf{f}$) and box ($[\, \alpha \,]\, \mathsf{f}$) modalities which we introduced in Chapter 2. Therefore, we can represent HML formulas in Agda using the Formula datatype shown in Figure 4.1.

```
data Formula (C : Container l₁ l₂) : Set l₁ where
  true false : Formula C
  ~_ : Formula C → Formula C
  _∧_ _∨_ _⇒_ : Formula C → Formula C → Formula C
  ⟨_⟩_ [_]_ : Shape C → Formula C → Formula C
```

Figure 4.1: Definition of HML formulas in Agda

Clearly, the constructors of the Formula datatype correspond to the terminal and non-terminal symbols in the BNF grammer shown above. It should be noted that the constructor, corresponding to the non-terminal symbol $\mathsf{f} \rightarrow \mathsf{f}$, is _⇒_, because in Agda the symbol → is reserved. Furthermore, the constructor, corresponding to the non-terminal symbol $\neg\, \mathsf{f}$, is ~_, in order for it not to be confused with the negation function ¬_ from Agda's standard library. With this in mind, we can express that a given program x satisfies a given HML formula f as $\mathsf{x} \models \mathsf{f}$, where _⊨_ is the satisfaction relation defined in Figure 4.2.

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 of the functional requirements of our ATM's software which we discussed in Chapter 2. 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 effect⁺
property₁ = ⟨ getPIN^s effect⁺ ⟩ true
```

```
_⊨_ : {C : Container l₁ l₂} {α : Set l₃} → Program C α → Formula 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 4.2: Semantics of HML formulas

Moreover, 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 using this implementation is that at the start of the program it must not be possible to execute any operation, other than the getPIN operation. Unfortunately, since our current implementation of HML does not have support for action formulas, in order to express this functional requirement, we have to explicitly mention each operation which we want to say is impossible:

```
property₂ : ℕ → Formula effect⁺
property₂ n = [ correctPINˢ effect⁺ n ] false ∧
              [ showBalanceˢ effect⁺ ] false ∧
              [ throwExceptionˢ effect⁺ ] false
```

Then, we can once again use the satisfaction relation, in order to prove that our ATM's software satisfies this functional requirement.

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

However, as we discussed in Chapter 2, this functional requirement can be expressed in a better way using action formulas. Thus, in the next chapter we are going to describe how we can add support for action formulas to our implementation of HML.

Chapter 5

Action Formulas

Action formulas are a way of representing sets of operations and their syntax can be represented using the following BNF grammar:

$$\mathsf{af} ::= \alpha \mid \mathsf{true} \mid \mathsf{false} \mid \overline{\mathsf{af}} \mid \mathsf{af} \cap \mathsf{af} \mid \mathsf{af} \cup \mathsf{af}$$

As we mentioned in Chapter 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 the action formula false represents the empty set of operations, while the non-terminal symbols $\overline{\mathsf{af}}$, af $\cap$ af and af $\cup$ af in the grammar make it possible to represent the compliment 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 (C : Container l₁ l₂) : Set l₁ where
  true false : ActionFormula C
  act_ : Shape C → ActionFormula C
  _ᶜ : ActionFormula C → ActionFormula C
  _∩_ _∪_ : ActionFormula C → ActionFormula C → ActionFormula C
```

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:

```
_∈_ : {C : Container l₁ l₂} → Shape C → ActionFormula C → Set l₁
_ ∈ 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 definion of HML formulas shown in Figure 4.1. We can do this by changing the constructors for the diamond and box modalities to use action formulas, instead of single operations:

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

Furthermore, we need to modify the semantics of the HML formulas shown in Figure 4.2, 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 effect⁺
property₃ = [ (act (getPIN^s effect⁺)) ^c ] false
```

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

```
proof₃ : ATM^s ⊨ 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 Chapter 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 chapter 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, namely the modal μ-calculus.

Chapter 6

The Modal μ-Calculus

In this chapter 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 Section 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 Section 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:

$$\mathsf{f} ::= \mathsf{true} \mid \mathsf{false} \mid \neg\, \mathsf{f} \mid \mathsf{f} \wedge \mathsf{f} \mid \mathsf{f} \vee \mathsf{f} \mid \mathsf{f} \rightarrow \mathsf{f} \mid \langle\, \alpha \,\rangle\, \mathsf{f} \mid [\, \alpha \,]\, \mathsf{f} \mid \mu\, \mathsf{X}\, .\, \mathsf{f} \mid \nu\, \mathsf{X}\, .\, \mathsf{f} \mid \mathsf{X}$$

As we can see from this grammar, all features of HML shown in Figure 4.1 are also present in the modal μ-calculus. However, there are three new symbols in the grammar, namely the terminal symbol X, which is used to denote formula variables, as well as the non-terminal symbols μ X . f and ν X . f, which represent the least- and greatest-fixed-point operators, respectively. Using these three new constructs it is possible to express recursion, hence the modal μ-calculus is sometimes referred to as Hennessy-Milner logic with recursion. In order to explain how this works, let us go over what the fixed-point operators actually represent. The non-terminal symbols for the fixed-point operators introduce a new formula variable X which is given the value f. However, using the terminal symbol X, we can reference the formula variable X in the formula f, since the formula variable X is in scope during the definition of f. Thus, the formula variable X can reference itself recursively. This means that the fixed-point operators define a (possibly recursive) function, denoted by the formula variable X, which takes a formula as input and returns a formula, and represent either the least fixed point or the greatest fixed point of that function, depending on which fixed-point operator is used.

Now, that we have some idea about what the fixed-point operators represent, we can 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′ C → Formula′ C) → Formula′ C
ref_ : Formula′ C → Formula′ C
```

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 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 [11]. 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 6.1.

```
data Formula^fp (C : Container l1 l2) : ℕ → Set l1 where
  true false : ∀ {n} → Formula^fp C n
  ~_ : ∀ {n} → Formula^fp C n → Formula^fp C n
  _∧_ _∨_ _⇒_ : ∀ {n} → Formula^fp C n → Formula^fp C n → Formula^fp C n
  ⟨_⟩_ [_]_ : ∀ {n} → ActionFormula C → Formula^fp C n → Formula^fp C n
  μ_ ν_ : ∀ {n} → Formula^fp C (suc n) → Formula^fp C n
  ref_ : ∀ {n} → Fin n → Formula^fp C n
```

Figure 6.1: 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 expessed 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 the latter type 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 Chapter 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 Chapter 3, we solved it by using containers [8], [9]. 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* [12]. In our implementation we use a data structure which we have called Container^i, because it is heavily inspired by the indexed containers presented by Altenkirch, Ghani, Hancock, et al. [12]. It has the typical structure of a container as well as an associated extension function[1] which transforms it into an indexed functor. However, before we can look at the exact definition of Container^i, we first need to introduce the following auxiliary datatypes.

```
data FixedPoint : Set where
  leastFP : FixedPoint
  greatestFP : FixedPoint

data Maybe' (α : Set ℓ) : Set ℓ where
  val_ : α → Maybe' α
  done : Maybe' α
  fail : Maybe' α

data Result (C : Container ℓ₁ ℓ₂) (α : Set ℓ₃) : Set (ℓ₁ ⊔ ℓ₂ ⊔ ℓ₃) where
  res_ : Maybe' (Program C α ×
                 FixedPoint ×
                 ∃[ n ] Formula^dnf-dis C (suc n) × Previous C (suc n)) → Result C α
  _×∃_ : ∀ {s} → ActionFormula C → (Position C s → Result C α) → Result C α
  _×∀_ : ∀ {s} → ActionFormula C → (Position C s → Result C α) → Result C α
```

The datatype FixedPoint is very straightforward and it is used to denote the types of fixed-point operator (either a least- or a greatest-fixed-point operator). The datatype Maybe' is

[1] For the exact implementation of the extension function, please refer to our source code on GitHub.

similar to the datatype Maybe from Agda's standard library, but it has one additional constructor, and it plays an important role in the definition of the Result datatype. The datatype Result represents the result of the satisfaction relation for a given formula and program. The constructors _×∃_ and _×∀_ represent the cases of the satisfaction relation, when the formula is a diamond modality (⟨_⟩_) or a box modality ([_]_), respectively, and the program is not pure. Meanwhile, the constructor res_ represents the cases of the satisfaction relation, when either the result is ⊤ (expressed by the done constructor of Maybe') or it is ⊥ (expressed by the fail constructor of Maybe') or we have reached a different fixed-point operator – either a new fixed-point operator or a reference to a previous one. From the definition we can see that the res_ constructor also uses the datatypes Formula^{dnf}-dis and Previous. The datatype Formula^{dnf}-dis represents a formula in disjunctive normal form (DNF), while the datatype Previous is used to represent the fixed point operators which can be referenced in a given formula. Essentially, we can think of the datatype Previous as representing a list of pairs of FixedPoint and Formula^{dnf}-dis. Now, that we have gone over the necessary auxiliary data structures, we can show our implementation of Container^i:

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

As part of our work, we have also developed 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 DNF;
5. Define the Shape of the container as the number of conjunctions in the formula;
6. Define the Position 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.

Let us illustrate the functionality of our algorithm via an example. Consider the formula:

```
ν (true ⇒ ⟨ true ⟩ (true ∧ ref zero))
```

This formula does not have any particular meaning, but it suffices to demonstrate the functionality of our algorithm. Moreover this formula does not use any specific effect operations and is therefore valid for any effect. Thus, let us assume some arbitrary effect which this formula is using. In order to define the satisfaction relation for some program and this formula, we have to use our algorithm to translate the following formula into an instance of Container^i:

```
true ⇒ ⟨ true ⟩ (true ∧ ref zero)
```

The first step is to desugar all implications:

```
(∼ true) ∨ ⟨ true ⟩ (true ∧ ref zero)
```

Then, we have to desugar all negations:

```
false ∨ ⟨ true ⟩ (true ∧ ref zero)
```

After that, we have to convert the formula to DNF:

```
false ∨ ((⟨ true ⟩ true) ∧ (⟨ true ⟩ ref zero))
```

Figure 6.2: Result of converting the formula into DNF

At this point we should notice that our formula contains only one ref_ constructor and it is preceded by an an even number of negations (zero). Thus, we do not need to replace it with false. Moreover, we can see that our formula is a disjunction of two conjunctions. Therefore, we can deduce that the corresponding instance of $\mathsf{Container}^i$ will have $\mathsf{Shape} = 2$. Finally, since instances of $\mathsf{Container}^i$ are also parameterized by the return type of the programs which they can be used with, let us assume that we want to use this instance of $\mathsf{Container}^i$ with a program whose return type is $\top$. Then, we can define the instance of $\mathsf{Container}^i$ corresponding to our formula as follows:

```
container : Container^i effect ⊤
Shape container = 2
Position container zero _ = [ res fail ]
Position container (suc zero) x with free x
... | pure _ = res done ::+ [ res done ]
... | impure (s , c) = (true ×∃ (λ (_ : Position effect s) → res done)) ::+
                       [ true ×∃ (λ p → res (val (c p ,
                                                  leastFP ,
                                                  zero ,
                                                  formula^dnf ,
                                                  ( leastFP , formula^dnf ))))) ]
```

where $\mathsf{formula}^{dnf}$ refers to the formula in DNF shown in Figure 6.2.

Using this algorithm we can translate any modal μ-calculus formula into an instance of $\mathsf{Container}^i$. Thus, we can define the semantics of our fixed-point operators: they are represented as either the least fixed point or the greatest fixed 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. However, let us first consider the formulas, $\mu\ \mathsf{X}\ .\ \mathsf{X}$ and $\nu\ \mathsf{X}\ .\ \mathsf{X}$, in order to understand the difference between the least-fixed-point operator and the greatest-fixed-point operator. These formulas represent the least fixed point and the greatest fixed point, respectively, of a formula variable X, that just references itself. Since, these formulas do not use any concrete effect operations, we can express them using the datatype $\mathsf{Formula}^{fp}$ for some arbitrary effect as follows:

```
property₄ : Formula^fp effect zero
property₄ = μ ref zero

property₅ : Formula^fp effect zero
property₅ = ν ref zero
```

As we can see from the intuitive definitions of the record types Mu and Nu, that we showed in Section 6.1, the only difference between them is that the record type Mu, which is used

to define the semantics of the least-fixed-point operator, is inductive, while the record type Nu, which is used to define the semantics of the greatest-fixed-point operator, is coinductive. This means that, in order for a formula which uses a least-fixed-point operator to be satisfied, that least-fixed-point operator must be referenced a finite number of times, while formulas which use a greatest-fixed-point operator can be satisfied even if that greatest-fixed-point operator is referenced infinitely many times. Thus, property_4 is not satisfied by any program, since for all programs the least-fixed-point operator will be referenced infinitely many times. Meanwhile, property_5 is satisfied by all programs and we can prove that as follows:

```
proof₅ : {α : Set l} → {x : Program effect α} → x ⊨ property₅
nu proof₅ = zero , λ { refl → proof₅ }
```

Now, that we know the difference between the least-fixed-point operator and the greatest-fixed-point operator, let us see what functional requirements we can express using them. 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 the datatype Formula^{fp} as follows:

```
property₆ : Formula^fp effect⁺ zero
property₆ = 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:

```
proof₆ : ATM^s ⊨ ~ property₆
proof₆ = 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 chapter 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.

Chapter 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 $\cdot$ 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 $\cdot$ 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 can represent regular formulas in Agda using the following datatype:

```
data RegularFormula (C : Container l₁ l₂) : Set l₁ where
  ϵ : RegularFormula C
  actF_ : ActionFormula C → RegularFormula C
  _·_ _+_ : RegularFormula C → RegularFormula C → RegularFormula C
  _* : RegularFormula C → RegularFormula C
```

Next, in order to incorporate regular formulas into our definition of the modal μ-calculus from Figure 6.1, we need to change the constructors for the box and diamond modalities. However, instead of doing that, we will actually create an entirely new datatype to represent modal μ-calculus formulas with support for regular formulas:

```
data Formula^rf (C : Container l1 l2) : ℕ → Set l1 where
  true false : ∀ {n} → Formula^rf C n
  ~_ : ∀ {n} → Formula^rf C n → Formula^rf C n
  _∧_ _∨_ _⇒_ : ∀ {n} → Formula^rf C n → Formula^rf C n → Formula^rf C n
  ⟨_⟩_ [_]_ : ∀ {n} → RegularFormula C → Formula^rf C n → Formula^rf C n
  μ_ ν_ : ∀ {n} → Formula^rf C (suc n) → Formula^rf C n
  ref_ : ∀ {n} → Fin n → Formula^rf C n
```

The reason behind 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 would like 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 need to define a function desugar which desugars a $\mathsf{Formula}^{rf}$ into $\mathsf{Formula}^{fp}$. For the constructors true, false, $\sim\_$, $\_\wedge\_$, $\_\vee\_$, $\_\Rightarrow\_$, $\mu\_$, $\nu\_$ and ref_ the definition of this function is straightforward, since these 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_, $\_\cdot\_$ and $\_+\_$ their definitions look as follows:

```
desugar-rf^d : {C : Container l1 l2} → {n : ℕ} →
               RegularFormula C → Formula^fp C n → Formula^fp C 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

desugar-rf^b : {C : Container l1 l2} → {n : ℕ} →
               RegularFormula C → Formula^fp C n → Formula^fp C 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
```

As we can see, for those constructors the desugaring is straighforward, because we translate them accordaing to their meanings presented above. Unfortunately, we cannot use this approach for the constructor _*. For that case we need to use the following definitions:

$$\begin{aligned} \langle\ \mathsf{rf}\ ^*\ \rangle\ \mathsf{f} &= \mu\ \mathsf{X}\ .\ ((\langle\ \mathsf{rf}\ \rangle\ \mathsf{X})\ \vee\ \mathsf{f}) \\ [\ \mathsf{rf}\ ^*\ ]\ \mathsf{f} &= \nu\ \mathsf{X}\ .\ (([\ \mathsf{rf}\ ]\ \mathsf{X})\ \wedge\ \mathsf{f}) \end{aligned}$$

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 formula:

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

This formula does not have any particular meaning, but it suffices to demonstrate the desugaring process. Using the datatype $\mathsf{Formula}^{rf}$ 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 introduced.

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

```
property₇ : Formula^rf effect⁺ zero
property₇ = [ actF true * ] ⟨ actF true ⟩ true
```

As we can see, this representation is a lot more compact and readable than the one we saw at the end of Section 6.2. Moreover, since the regular formula gets desugared precisely into the fixed-point operator which we used to express this property at the end of Section 6.2, we can use the same proof to show that our ATM's software does not satisfy this property:

```
proof₇ : ATM^s ⊨ ~ property₇
proof₇ = proof₆
```

This example illustrates the benefit of using regular formulas: they make some common use cases of the fixed-point operators significantly simpler and cleaner to write. Now, we have implemented all features of the modal μ-calculus into our framework. Thus, in the next chapter we will introduce the first-order modal μ-calculus, which will add even more expressivity to our framework, and we will briefly explain how we have implemented it in Agda.

Chapter 8

The First-Order Modal μ-Calculus

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

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

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 existantial quantifiers are supported;
3. Least- and greatest-fixed-point operators are parameterized.

For the sake of brevity and simplicity, in this chapter we will only describe the intuition behind our implementation of those features without going into the implementation details.[1]

By far the simplest one of the new features is allowing 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, if the boolean expression evaluates to true. In order to incorporate this feature into our implementation, we need to add a new constructor to Formula^{rf} which takes some boolean value and returns a formula. However, 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, and we add a new constructor to Formula^{rf} which looks as follows:

$$\mathsf{val\_} : \forall\ \{n\} \rightarrow \mathsf{Set}\ l \rightarrow \mathsf{Formula}^{rf}\ C\ l\ n$$

Using this new constructor, we can construct a formula using any element of Set l. This can be used together with the next feature which we will introduce, universal and existential quantifiers, in order to express, for example, conditions. For instance, we could say that "there exists a natural number n, such that $\mathsf{n} > 42$". Such a formula can be represented using the existential quantifier and the val_ constructor to express the condition $\mathsf{n} > 42$. Furthermore, the formula val X holds for any program, if and only if 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 $\mathsf{n} > 42$.

[1] It should be noted that the first two features mentioned above (denoted with a *) 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 chapter we will only describe how we have implemented them for the latter.

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}, that represent the universal and existential quantifiers, and then transforming the 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, in order to transform a formula into PNF, we might need to rename some of the variables which are introduced by the quantifiers, which cannot be done automatically. Instead, we settled for a different, but equally expresive approach, namely representing formulas in PNF from the very beginning, thereby eliminating the need to transform 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 use the containerization technique which we described in Section 6.2 and we would have needed to come up with an entirely new containerization mechanism. Thus, in our implementation we use a new datatype called Quantified, which 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 the fact that the least- and greatest-fixed-point operators are parameterized. In order to represent this in our implementation, we added two additional parameters 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 modified the constructor ref_ by giving it an additional argument: a list of values which will be passed to the parameters of the fixed-point operator, that is being referenced. Moreover, in order to keep track of what types of parameters each fixed-point operator expects, we switched from using standard de Bruijn indexing, represented as natural numbers, to indexing formulas by a `Vec (List (Set l)) n`, where the length of the vector `n` denotes the number of fixed-point operators which we can reference, while the list at each index of the vector represents the types of the parameters expected by the corresponding fixed-point operator. Finally, we introduced a new datatype called Parameterized, that is used to represent 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:

```
data Formula^i {n : ℕ} (C : Container l₁ l₂) (l : Level) : Vec (List (Set l)) n → Set ((suc l) ⊔ l₁)

data Quantified {n : ℕ}
                (C : Container l₁ l₂)
                (l : Level)
                (xs : Vec (List (Set l)) n) : List (Set l ⊎ Set l) → Set ((suc l) ⊔ l₁) where
  formula_ : Formula^i C l xs → Quantified C l xs []
  ∀(_)_ : ∀ {αs} → (α : Set l) → (α → Quantified C l xs αs) → Quantified C l xs (inj₁ α :: αs)
  ∃(_)_ : ∀ {αs} → (α : Set l) → (α → Quantified C l xs αs) → Quantified C l xs (inj₂ α :: αs)

data Parameterized {n : ℕ}
                   (C : Container l₁ l₂)
                   (l : Level)
                   (xs : Vec (List (Set l)) n) : List (Set l) → Set ((suc l) ⊔ l₁) where
  quantified_ : ∀ {αs} → Quantified C l xs αs → Parameterized C l xs []
  _↦_ : ∀ {αs} → (α : Set l) → (α → Parameterized C l xs αs) → Parameterized C l xs (α :: αs)
```

```
data Formula^i C l where
  true false : ∀ {xs} → Formula^i C l xs
  val_ : ∀ {xs} → Set l → Formula^i C l xs
  ~_ : ∀ {xs} → Formula^i C l xs → Formula^i C l xs
  _∧_ _∨_ _⇒_ : ∀ {xs} → Formula^i C l xs → Formula^i C l xs → Formula^i C l xs
  ⟨_⟩_ [_]_ : ∀ {xs} → RegularFormula C l → Formula^i C l xs → Formula^i C l xs
  μ_._ ν_._ : ∀ {αs xs} → Parameterized C l (αs :: xs) αs → Arguments l αs → Formula^i C l xs
  ref_._ : ∀ {xs} → (i : Fin (length xs)) → Arguments l (lookup xs i) → Formula^i C l xs

Formula : (C : Container l1 l2) → (l : Level) → (αs : List (Set l ⊎ Set l)) → Set ((suc l) ⊔ l1)
Formula C l αs = Quantified C l [] αs
```

Using this implementation we can now express the last functional requirement which we discussed in Chapter 2, namely that it should not be possible to execute the showBalance operation for a given user of the ATM, before executing the getPIN operation for that user. This property can be defined using our framework as follows:

```
property8 : Formula effect+ 0ℓ []
property8 = formula ν Bool ↦ (λ b → quantified formula
              [ actF (act getPIN^s effect+ ∪ act showBalance^s effect+) ^c ] ref zero . (b :: []) ∧
              [ actF act getPIN^s effect+ ] (val T (not b) ∧ ref zero . (true :: [])) ∧
              [ actF act showBalance^s effect+ ] (val T b ∧ ref zero . (false :: [])) . (false :: [])
```

As we can see, this definition is more complex than the ones which we have seen so far. However, if we compare this definition with the example from Chapter 2, we can see that the overall structure of the formula is maintained and the syntax is very similar. The main difference is that in this definition we provide the initial value of the parameter of the greatest-fixed-point operator at the end of the formula (false :: []), while in the example from Chapter 2 the initial value is provided directly after the parameter is introduced (b:Bool:=false).

We can prove that our model of the ATM's software satisfies this functional requirement as follows:

```
proof8 : ATM^s ⊨ property8
nu proof8 = zero ,
            (λ h → ⊥-elim (h (inj1 (lift refl)))) ,
            (λ { _ _ refl → nu^c tt }) ,
            (λ { _ _ refl → nu^c (zero ,
              (λ { _ false refl → nu^c (zero ,
                (λ _ ()) ,
                (λ ()) ,
                (λ ()) ,
                (λ ()) ,
                λ ()) ;
                  _ true refl → nu^c (zero ,
                (λ h → ⊥-elim (h (inj2 (lift refl)))) ,
                (λ ()) ,
                (λ ()) ,
                (λ { _ _ refl → nu^c tt }) ,
                (λ { _ _ refl → proof8 })) }) ,
              (λ ()) ,
              (λ ()) ,
              (λ ()) ,
              λ ()) }) ,
```

```
  (λ ()) ,
  λ ()
```

This proof, similarly to the definition which we just discussed, is also more complex than the ones which we have seen so far. However, after examining the proof, we can see that, although it looks complex, it only uses a few basic structures, namely the principle of explosion (denoted using the function ⊥-elim in Agda), the unit type (denoted as ⊤ and constructed using tt in Agda) and propositional equiality (denoted as _≡_ and constructed using refl in Agda). Thus, this proof serves as a demonstration of how, using our framework, we can prove even complex properties in a relatively straightforward way.

We have now introduced all features of our framework. In the remainder of this thesis, we will first demonstrate one important advantage of our framework over traditional methods for formal verification of effectful programs, namely that our framework allows us to obtain runnable programs directly from the (verified) models; then, we will discuss related work; finally, we will conclude and give some examples of future work.

Chapter 9

From Models to Runnable Programs

Thus far in this thesis we have introduced a framework which allows us to represent sequential effectful programs as instances of the coinductive free monad and to express and prove the functional requirements of those programs using the first-order modal μ-calculus. However, this approach to formal verification of effectful programs has one important advantage which we have not discussed so far. Traditionally, in order to formally verify that a given software satisfies its functional requirements, we would need to first define a model of that software and then verify that this model satisfies the necessary functional requirements. However, in that case the model is defined separately from the software itself. Thus, it is possible that the model is incorrect, meaning that it does not accurately represent the behavior of the software, in which case verifying it is useless, since that would not provide any guarantees for the behavior of the actual software. Fortunately, with our framework we can avoid this problem. The reason for this is that after defining a model of an effectful program as an instance of the coinductive free monad, we can obtain the actual implementation of said effectful program from that model by using effect handlers. Thus, since the actual software is obtained from the model, rather than being a completely separate entity, it is guaranteed that the model is correct and, by extension, that the actual software also satisfies all functional requirements which are satisfied by the model.

Let us demonstrate this with an example. Imagine that we need a program which sends some messages along a unidirectional synchronous communication channel between person A and person B, such that person A can only send messages and person B can only receive messages. In order to represent this program as an instance of the coinductive free monad, we first need to define the effect which it uses:

```
data CommunicationShape : Set where
  send : String → CommunicationShape
  receive : CommunicationShape

communicationEffect : Container 0ℓ 0ℓ
Shape communicationEffect = CommunicationShape
Position communicationEffect (send _) = ⊤
Position communicationEffect receive = String
```

This implementation of the communication effect consists of two operations, namely send and receive, where the send operation expresses that person A sends some message to person B (hence it requires a single argument of type String representing the message which is being sent) and the receive operation expresses that person B receives a message from person A (hence its position is of type String). Now, that we have defined communicationEffect, we can use it to define a model of our program:

```
program : List String → Program communicationEffect ⊤
free (program []) = pure tt
free (program (i :: is)) = impure (send i , λ _ → ⟪ impure (receive , λ _ → program is) ⟫)
```

As we can see from the definition, the program is parameterized by a List String, representing the list of messages which person A wants to send to person B. Furthermore, since the communication channel is synchronous, person B receives all messages (the receive operation is executed) directly after person A sends them (the send operation is executed). Now, we can use our framework to verify, for example, that whenever the receive operation is executed, there is a message to be received:

```
property₉ : Formula communicationEffect 0ℓ []
property₉ = formula ν ℕ ↦ (λ n → quantified formula
              [ actF ((∃( String ) λ x → act send x) ∪ act receive) ᶜ ] ref zero . (n :: []) ∧
              [ actF (∃( String ) λ x → act send x) ] ref zero . (suc n :: []) ∧
              [ actF act receive ] (val (n > 0) ∧ ref zero . (pred n :: []))) . (0 :: [])
```

The definition of this property is very similar to that of the property which we discussed at the end of Chapter 8. The main difference is that, instead of having a parameter of type Bool, the greatest-fixed-point operator has a parameter of type ℕ which is used to keep track of the number of messages which have been sent by person A, but have not been received by person B yet. We can prove that our model of the program satisfies this requirement as follows:

```
proof₉ : (is : List String) → program is ⊨ property₉
nu (proof₉ []) = zero ,
                 (λ { refl → nuᶜ tt }) ,
                 (λ { refl → nuᶜ tt }) ,
                 (λ { refl → nuᶜ tt }) ,
                 λ { refl → nuᶜ tt }
nu (proof₉ (x :: is)) = zero ,
                        (λ h → ⊥-elim (h (inj₁ (x , lift refl)))) ,
                        (λ { _ _ refl → nuᶜ (zero ,
                          (λ h → ⊥-elim (h (inj₂ (lift refl)))) ,
                          (λ ()) ,
                          (λ { _ _ refl → nuᶜ (lift (s≤s z≤n)) }) ,
                          (λ { _ _ refl → proof₉ is })) }) ,
                        (λ ()) ,
                        λ ()
```

As we can see, this proof is also similar in structure to the one presented at the end of Chapter 8. However, since in this case our program is parameterized by a List String, we need to pattern match on that parameter and provide a separate proof for each case.

Now, that we have verified that our model satisfies the necessary functional requirements, let us demonstrate how we can obtain a runnable program from our model. In order to achieve this, we need to define a handler for communicationEffect: a function which specifies the exact implementation of each operation of communicationEffect. However, before we can do that, we need one more auxiliary definition:

```
tau : Container 0ℓ 0ℓ
Shape tau = ⊤
Position tau _ = ⊤
```

The container tau represents an auxiliary effect which we use to represent silent steps [13], in order to satisfy Agda's guardedness requirements. Using the container tau, we can define a handler for communicationEffect as follows:

```
handler : {α : Set} → Program communicationEffect α →
          Program tau (Maybe (α × List String))
handler p = handler' p [] []
  where
  handler' : {α : Set} → Program communicationEffect α → List String → List String →
             Program tau (Maybe (α × List String))
  free (handler' p sent received) with free p
  ... | pure a = pure (just (a , received))
  ... | impure (send x , c) = impure (tt , λ _ → handler' (c tt) (sent ::ʳ x) received)
  ... | impure (receive , c) with sent
  ... | [] = pure nothing
  ... | x :: sent = impure (tt , λ _ → handler' (c x) sent (x :: received))
```

As we can see from this definition, handler is a function which takes some program, that uses communicationEffect and returns a program which uses the effect tau. Furthermore, the handler changes the return type of the program: it adds a value of type List String to the output of the program, which in this case represents the list of messages which person B has received from person A, and wraps the return type in a Maybe, indicating that the program may fail. By inspecting the implementation of handler, we can see that it works by keeping track of two lists: one which contains all messages, that person B has received, and one which contains all messages, that person A has sent, but person B has not received yet. Moreover, we can see that, if the receive operation is executed and there are no messages to be received, the program fails, which is why the addition of Maybe to the return type of the output program was necessary.

Finally, we can use handler to obtain a runnable program from our model. Thus, since the runnable program is obtained from the model, we know that all functional requirements which are satisfied by the model will also be satisfied by the runnable program. Moreover, we can now define additional properties for the runnable program, namely ones which reason about the behavior of the program for concrete inputs. For example, we can define the following property, which states that person B will always receive all messages which person A sends and they will be received in the same order as that in which they were sent:

```
∀ {xs} → handler (program xs) ≡ ⟪ pure ( tt , reverse (map just xs)) ⟫
```

Unfortunately, this property is not entirely correct, because it omits all of the silent steps which handler introduces (one for each send and receive operation in the program). However, even if we add all of the necessary silent steps, this property will be unprovable. The reason for this is that, since Program is defined as a coinductive record type, we cannot compare programs using propositional equality (_≡_), as they may be infinite. Instead, in order to prove that two programs have the same behavior, we need to define and use a notion of bisimulation. However, while this is possible, it is out of the scope of our work and we will therefore not cover it in this thesis.

Chapter 10

Related Work

Propositional dynamic logic (PDL) is a kind of logic which was originally introduced by Vaughan Pratt [14] for the purpose of reasoning about computer programs. Later, HML was designed by Matthew Hennessy and Robin Milner [10], 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 [4]. 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 the development of the tool mCRL2 [3] 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 [15]. 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 Lago and Ghyselen [5] extends techniques due to Ong [16] 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 Lago and Ghyselen [5] 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 [17] demonstrate how we can derive effectful programs directly from a given functional specification, represented by a pre- and postcondition. 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 thesis, as the pre- and postconditions for such program derivations.

Chapter 11

Conclusion and Future Work

In this thesis we have presented a noval way of reasoning about sequential effectful programs within a proof assistant. Through the use of algebraic effects and effect handlers our approach makes it possible to obtain runnable programs directly from a (verified) model. Because of this, when we use our framework to verify that a model satisfies some set of requirements, we are not only reasoning about that model; we are directly reasoning about all programs which can be obtained from that model by using different effect handlers. Thus, our approach removes the risk of having a model which does not represent the actual program correctly. Furthermore, we have shown that it is possible to reason about programs in a proof assistant using dynamic logic, in particular, the first-order modal μ-calculus. And, while we have written our framework entirely in Agda, we believe that our results should be reproducible in other proof assistants as well, as long as they have support for all of the necessary features (e.g. coinduction).

However, our framework has its limitations. For example, in its current state it cannot be used to reason about parallel programs. Additionally, when working with complex formulas or large programs, our framework can become slow, which is a big downside, when it comes to using it in real-world scenarios. Moreover, our framework is currently not very user-friendly, especially for novice users, since users need to be familiar with all of the transformations which are happening in the backend, in order to successfully use it.

Therefore, when it comes to future work, there are a lot of different avenues which can be taken. Firstly, it is important to test our framework in more real-world scenarios. Most of the examples which we presented in this thesis were very simple, since their goal was to introduce the features of our framework. However, through those examples we have only scratched the surface of what can be expressed using the first-order modal μ-calculus. Thus, it would be interesting to see how our framework would perform in a real-world scenario, when the functional requirements which have to be proven will be much more complex.

Aside from that, it is important to try to address the downsides of the framework which we mentioned earlier. For example, one of the reasons why the framework is currently slow, when working with large programs and/or complex functional requirements, is the large number of transformations which occur on the backend. Those transformations are necessary, in order to automate the containerization of formulas. However, containers are not the only way of representing strictly-positive functors. It is possible that a different representation (e.g. descriptions [18]) could be more suitable in our use case. And, if we find a representation which formulas can be converted to directly, or using a smaller number of transformations, that should significantly improve the performance of the framework as well as its user-friendliness.

Another possible improvement would be to extend our framework by adding support for parallel programs. For example, the tool mCRL2, which was a big source of inspiration for our work, has support for parallel programs through the use of multiactions. Maybe a

similar feature could be added to our framework, in order to increase its functionality.

Finally, in this thesis we have shown that it is possible to use the first-order modal μ-calculus to express and prove the functional requirements of sequential effectful programs represented as instances of the coinductive free monad in a proof assistant. This approach requires us to manually prove that the functional requirements are satisfied by the given model. However, an interesting option to explore is, whether it is possible to define a *refinement calculus* based on the first-order modal μ-calculus, which would allow us to, given a set of functional requirements expressed using the first-order modal μ-calculus, incrementally define a model which satisfies all of those requirements (similar to the work of Swierstra and Baanen [17]).

Bibliography

[1] Edsger Wybe Dijkstra. "Notes on Structured Programming". EWD249 http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF. Apr. 1970.

[2] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X. URL: http://research.microsoft.com/users/lamport/tla/book.html.

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

[4] Dexter Kozen. "Results on the propositional μ-calculus". In: *Theoretical computer science* 27.3 (1983), pp. 333–354.

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

[6] Andreas Abel, Brigitte Pientka, David Thibodeau, et al. "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*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 27–38. DOI: 10.1145/2429069.2429075. URL: https://doi.org/10.1145/2429069.2429075.

[7] Wouter Swierstra. "Data types à la carte". In: *J. Funct. Program.* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758. URL: https://doi.org/10.1017/S0956796808006758.

[8] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "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*. Ed. by Andrew D. Gordon. Vol. 2620. Lecture Notes in Computer Science. Springer, 2003, pp. 23–38. ISBN: 3-540-00897-7. DOI: 10.1007/3-540-36576-1_2. URL: https://doi.org/10.1007/3-540-36576-1%5C_2.

[9] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "Containers: Constructing strictly positive types". In: *Theor. Comput. Sci.* 342.1 (2005), pp. 3–27. DOI: 10.1016/j.tcs.2005.06.002. URL: https://doi.org/10.1016/j.tcs.2005.06.002.

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

[11] Nicolaas Govert De Bruijn. "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. 5. Elsevier. 1972, pp. 381–392.

[12] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, et al. "Indexed containers". In: *J. Funct. Program.* 25 (2015). DOI: 10.1017/S095679681500009X. URL: https://doi.org/10.1017/S095679681500009X.

[13] Li-yao Xia, Yannick Zakowski, Paul He, et al. "Interaction trees: representing recursive and impure programs in Coq". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–32.

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

[15] Ana de Almeida Borges. "Towards a Coq Formalization of a Quantified Modal Logic." In: *ARQNL@ IJCAR*. 2022, pp. 13–27.

[16] C.-H. Luke Ong. "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, 2006, pp. 81–90. DOI: 10.1109/LICS.2006.38. URL: https://doi.org/10.1109/LICS.2006.38.

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

[18] James Chapman, Pierre-Évariste Dagand, Conor McBride, et al. "The gentle art of levitation". In: *ACM Sigplan Notices* 45.9 (2010), pp. 3–14.