DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Verified Monadification and Memoization

Florian Sextl

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Verified Monadification and Memoization

Verifizierte Monadifikation und Memoisierung

Author:	Florian Sextl
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	MSE Simon Wimmer
Submission Date:	February 17, 2020

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Garching, Florian Sextl

Abstract

This thesis describes an approach to extend the *Monadification and Memoization* framework for the theorem prover Isabelle. This framework allows for introducing memoization into a program by embedding it into a specific monad. In addition, an automatic correctness proof for the monadified function is performed. However, the monadification method introduces a rather large structural overhead and only works for its own memoization monad. The goal of our extension approach was, therefore, to reduce the overhead and to allow for more monads. We developed a new monadification procedure that meets both objectives. From this, we derived a proof-of-concept reference implementation that proved to fulfill all new requirements while keeping the framework's original functionalities. The usability of our approach was also examined for a real-world example, the Bellman-Ford algorithm. Despite this, there are limitations of our method as well. For instance, the generalization of usable monads comes with an increased implementation overhead for the user, as they have to handle monad-specific lemmata and effects. We nevertheless conclude that our approach is successful for a general setting and can be used to extend the existing framework.

Diese Arbeit beschreibt einen Ansatz, um das *Monadifikations und Memoisierungs* Rahmenwerk für den Theorembeweiser Isabelle zu erweitern. Dieses Rahmenwerk erlaubt es, Memoisierung in ein Programm einzubringen, indem dieses in eine spezielle Monade eingebettet wird. Zusätzlich wird ein automatischer Beweis über Korrektheit dieser Monadifikation geführt. Jedoch bringt diese Monadifikations-Methode eine merklichen Strukturoverhead und funktioniert nur für die eigene Memoisierungsmonade. Das Ziel unserer Erweiterung war dementsprechend, den Overhead zu reduzieren und mehr Monaden zu unterstützen. Wir entwickelten eine neue Monadifizierungsprozedur, welche diese beiden Anforderungen erfüllt. Davon ausgehend wurde eine Referenzimplementierung als Konzeptnachweis erstellt, welche zum einen die genannten Auflagen einhält und zum anderen die ursprüngliche Funktionalität des Rahmenwerk weiterhin aufrecht hält. Die Brauchbarkeit des Ansatzes wurde außerdem an einem Beispiel aus der echten Welt, dem Bellman-Ford Algorithmus, gezeigt. Trotz dieser Errungenschaften weist unsere Methode auch Schwächen auf. Beispielsweise bringt die Erweiterung der nutzbaren Monaden einen größeren Implementierungsaufwand für die Nutzer mit sich, da diese monaden-spezifische Lemmata und Effekte handhaben müssen. Wir kommen trotzdem zu dem Schluss, dass unser Ansatz im Allgemeinen erfolgreich und damit für eine Erweiterung des Rahmenwerks geeignet ist.

Contents

Part I.

Introduction and Theory

1. Introduction

As software influences our daily lives more and more, bugs become potentially more critical and serious, too. While thorough testing and thoughtful software design decreases the number of bugs and other faults, formal verification is the best way to ensure complete correctness. Formal verification is often done with verification tools, of which the proof assistants turned out to be the most usable. Most proof assistants, like Isabelle, use purely functional programming languages for formalization, as they are structurally suited for simple proving techniques and have no side effects that could affect the proofs. The only way purely functional programming can make use of side-effect-like functionalities is by monads. As a result, monads have become an important concept to the functional programming paradigm. However, they are often rather complicated to introduce into a program. The programmer has to choose which monadic effects they want to use and then introduce them in all parts of the program, where they are necessary. Afterwards, the programmer has to adapt all otherwise pure functions that make use of the monadic effects. This requires a lot of knowledge about both the program and the used monad and takes a lot of work for bigger software systems. As a result, a method for automatically monadifying a pure program would be of great help. If this method would in addition provide its own correctness proofs, it could be used trivially for optimizing verified code. A tool achieving this is the *Monadification and Memoization* [24] framework for Isabelle/HOL. It allows for automatic embedding of functions in a memoization monad and, in addition, proves this step to be correct.

This thesis describes an approach to extend this tool. Our goal is to reduce the introduced monadic overhead while also enabling the insertion of generic monadic effects in a still verified way. Therefore, we developed a new monadification procedure and implemented it as a proof-of-concept. We then demonstrated, how our method can be used instead of the old framework and how it influences the correctness proofs.

This thesis begins by explaining all important theoretical backgrounds that are necessary to understand the details of our approach. This is done in Chapter 2 and involves short introductions to the Isabelle system, the monad pattern, parametricity reasoning and dynamic programming. Following these introductions, we give a short overview over related work and the research we built upon in Chapter 3. Starting the main part of our thesis, we introduce both the old and the new monadification procedures in detail in Chapter 4. We reason not only about the monadification rules, but also about how monadic effects and other special cases are handled. Subsequently, we explain how the monadified functions are proven to be correct in Chapter 5. We show how the proof works in detail and what parts have to be changed to adapt the proof method to our new procedure. After this, we compare our proposed method to the old one in detail and conclude with a real-world example featuring the Bellman-Ford algorithm in Chapter 6. We demonstrate how the monadification procedure changes the program and how this step is then proven to be correct. In the last part of this thesis, we summarize our findings and discuss our work. In Chapter 7, we recapitulate the improvements achieved by our approach and balance them with the associated limitations and shortcomings. We then summarize our work and discuss possible future work in Chapter 8.

2. Theory

The following sections introduce the most relevant theoretical backgrounds this thesis builds upon. At first, the Isabelle system is described in Section 2.1, as the *Monadification and Memoization* framework is implemented in part in Isabelle/HOL and in part in Isabelle/ML, which are both Isabelle subsystems. Afterwards the concept of monads is introduced in Section 2.2 as foundation for the tool. Next, the theoretical background of the automated proof mechanism is explained in Section 2.3. At last, the frameworks original use case is depicted in Section 2.4.

2.1. Isabelle/HOL

The generic theorem prover Isabelle [17, 18] is a LCF-style [7] theorem prover and provides an intuitionistic fragment of higher-order logic, called *meta-logic* or Isabelle/Pure, on which (nearly) arbitrary user-defined logic systems can be built. Following the LCF-style, the *meta-logic* is implemented as a small trusted kernel in the meta language (Standard) ML[1]. On top of this kernel, a number of predefined object logics are provided, of which Isabelle/HOL [15] is the most frequently used. Isabelle/HOL implements the common higher-order logic (HOL, cf. [6] for the theorem proving environment of the same name) that can be defined as a combination of functional programming and logic reasoning.

Isabelle/HOL provides both automatic and interactive proving capabilities and comes with a number of different proof methods by default. Though, we mainly use one of these methods that comprises natural deduction and backtracking. All proof methods are composed of several proof tactics at the *meta-logic* level. These tactics are ML functions working directly on kernel data types and are therefore complex and hard to handle. As a result, the *Isar* language, short for *Intelligible Semi-Automated Reasoning* [23], was introduced as an intuitive way of structuring proofs and making them more readable for humans. Isar, as well as the whole Isabelle system, are thoroughly documented for users on the Isabelle webpage[2], whereas Isabelle/ML is described in detail for Isabelle developers in the Isabelle Cookbook [20].

[1]The whole implementation is also referred to as Isabelle/ML and is based on Poly/ML (`https://polyml.org/`).

[2]`https://isabelle.in.tum.de/`

Aside from the proof mechanisms, Isabelle/HOL also provides a purely functional programming language. This programming language is similar to the underlying meta language and is centered around well-defined, higher-order functions. The standard way of defining a function in Iabelle/HOL is via the **fun** command. This command allows for total and (mutually) recursive functions, which may also use pattern-matching on their arguments, and automatically generates not only the corresponding simplification and induction rules, but also determines and proves the termination order of these functions. If a function is not recursive or does not use pattern-matching on its input, it can also be implemented with the **definition** command. This command introduces the function as an abbreviation for its definition and generates a rule to unfold this definition (denoted by the suffix *_def*). In contrast to **fun**, **definition** does not generate an induction scheme or find the termination order. This difference is of minor importance for our work later on.

Regardless of their definition method, all functions used in this thesis are defined in Isabelle/HOL's type system, which is derived from the simply-typed lambda calculus by Church [2], as HOL is directly based on that. The notation $x :: t$ denotes thereby that value x is of type t. As Isabelle/HOL can infer most types correctly, this notation is rarely required in the source code and only used in this thesis to help the reader understand the types. Isabelle/HOL comes with a broad variety of default types like *int* for integers or *bool* for boolean values. These can be combined to form product types as tuples $'a \times\ 'b$ or lists $'a\ list$, where $'a$ and $'b$ are polymorphic type parameters that can be instantiated by any type. Function types are written as $'a \rightarrow\ 'b$[3] for a function with argument type $'a$ and result type $'b$. As Isabelle/HOL works with curried functions, any function with more than one argument has a type $'a_0 \rightarrow\ 'a_2 \rightarrow\ \cdots \rightarrow\ 'a_n \rightarrow\ 'b$ where an application to all but the last argument of type $'a_n$ returns a function again. More complex data types can be defined with the **datatype** command, which allows for polymorphic sum types with constructors and built-in destructor functions. The *list* data type can, for example, be defined as

$$\textbf{datatype}\ 'a\ list = Cons\ (head:\ 'a)\ (tail:\ 'a\ list)\ |\ Nil$$

with the type constructors *Cons* and *Nil* as well as the destructors *head* and *tail*. In addition, types can implement type classes. Type classes in Isabelle/HOL are similar to Haskell's early type classes and define both functions and properties encoded in theorems that a type instance of this class has to provide. However, unlike newer Haskell versions, Isabelle/HOL does not support polymorphic type classes. Other than that, Isabelle/HOL has the concept of locales, which denote a specific environment

[3]Normally, the function type in Isabelle/HOL is built with $\Rightarrow$, but for readability reasons $\rightarrow$ is used here instead.

with functions, constants and lemmata that may depend on polymorphic parameters. These parameters can be terms and types with associated assumptions encoded in theorems. Although locales are more powerful than type classes, they can not be used to work with polymorphic types that require a type parameter themselves (e.g. $'a\ 'b$ for a type $'b$ with one parameter $'a$). These limitations are the reason for our approach in Chapter 5.

2.2. Monads

A monad is a concept from category theory[4] first formally introduced to computer science by Eugenio Moggi [14]. Nowadays, monads are a crucial aspect of (purely) functional programming, as they can be used to structure complex computations and data flow and even to mimic side effects in a safe and pure fashion. Monads can be thought of as representations of computation, i.e. computations that are built up in a dynamic way and can be handled like data before they are executed. Hence, they can encode generic boilerplate code and lift programming to a higher level, which is the reason why monads have become such an important design pattern. Especially the programming language Haskell makes heavy use of monads [16], at least since Wadler examined the concept in [21].

As a result, a slightly adjusted version of Haskell's monad type class shall be used to define the structure of a monad in the following. In general, a monad is a polymorphic data type that expects one type parameter denoted as $'a\ M$. It needs a function to build up a monad value from a value $x :: 'a$, which is called *return* and is expressed as the surrounding operator $(\langle \ldots \rangle) :: 'a \to\ 'a\ M$ from here on, resulting in the simple monad $\langle x \rangle$. Monads can be composed by the *bind* function denoted by the infix operator $(\gg\!=) :: 'a\ M \to\ ('a \to\ 'b\ M) \to\ 'b\ M$. It unwraps a value from a monad and forwards it into the given function, which returns a monadic value itself. By this, the flow of computation can be specified. To be more specific, the *bind* operator allows for a special notation emphasizing the control flow called the *do*-notation. Within a *do* block a term $x \leftarrow m$ is equal to $m\ \gg\!=\ (\lambda x.\ f)$ where f is the next computation step. A statement without $\leftarrow$ is equal to $m\ \gg\!=\ (\lambda\_.\ f)$ and discards the embedded value. The last value of the *do* block has to be monadic again. A simple example would look like the

[4]Wadler calls category theory an "arcana" and an "abstruse theory" [21] not necessary to understand monads in computer science. Thus, we omit a detailed description of the mathematical counterpart.

following, where the *do* block on the right works the same as the term on the left:

$$\begin{array}{rl}
a \gg\!= & (\lambda x. \\
b \gg\!= & (\lambda y. \\
\mathit{log}\ \text{"a+b"} \gg\!= & (\lambda\_. \\
& \langle x+y \rangle)))
\end{array}
\qquad\qquad
\begin{array}{r}
\mathit{do}\ \{ \\
x \leftarrow a; \\
y \leftarrow b; \\
\mathit{log}\ \text{"a+b"}; \\
\langle x+y \rangle \\
\}
\end{array}$$

The *log* constant denotes a function that adds a string to a *writer monad* and, thus, to the programs log.

Both the *return* and the *bind* operators have to comply with certain rules to allow for a type to be a monad. These rules are called the *monad laws* and can be described as follows:

1. $m \gg\!= \mathit{return} = m$
2. $\langle x \rangle \gg\!= f = f\ x$
3. $m \gg\!= (\lambda x.\ f\ x \gg\!= g) = (m \gg\!= f) \gg\!= g$

To sum up, *return* has to be an identity for *bind*, which also needs to be associative. Especially the second law can be used often to simplify monadic terms and is of importance for later sections.

The *writer monad* used above is one example of a simple, yet useful monad. It works like an append-only list and is often used to build a log or to document certain events. It could also be used to implement debugging capabilities like a stack trace. The opposite of the *writer monad* is the *reader monad*, which encapsulates a set of read-only values in a similar fashion as global constants in imperative programming. Next, the *state monad* combines both ideas and holds a readable and writable state memory. It can be used to implement memoization, whereby the state holds all former results that may be used for future computations. This kind of monad is the basis for Section 4.1 and will be investigated further there. Even simpler types can form a monad. Examples for this are the *list* type or the *option*[5] type. Both monads are special in that they have two type constructors, of which one encodes an empty value. Whereas the *list monad* uses the standard *Cons* and *Nil* constructors, *option* is built around *Some* and *None*, which either wrap *some* value or represent *none*. The most important monad in Haskell is probably the *IO monad*, which is used to abstract all interactions with the operating system and the file system and, therefore, encapsulates many functionalities that have side effects.

[5]In Haskell this type is called *Maybe*.

All of the monads explained above can be used to implement one specific functionality each. To combine these functionalities, monad transformers have been invented. These special structures allow for monads to be embedded in other monads while asserting direct access to each of them. Due to this property, most real-world software systems using monads are built around monad transformers rather than simple monads [16].

2.3. Relational Parametricity Reasoning

Relational parametricity is a concept based on Reynolds' abstraction theorem [19] that describes relational properties of types in the polymorphic lambda calculus also known as System F. The theorem states that terms evaluated in related environments are bound to yield related values. The basic idea to formalizing this intuition is to see a type as a binary relation on the set of its values. Constant types relate their values only reflexively, i.e. the type relation is always the identity relation, which can be implemented as simple equality for most types. This case shows the discrepancy between mathematical relations and their standard implementations as relator functions returning boolean values. We use both notations interchangeably in this thesis. Function types relate their values with the function relator $\dashrightarrow$, which describes that related arguments should induce related results:

$$R \dashrightarrow S = \lambda f\ g.\ \forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y)$$

Here, $R :: 'a \rightarrow\ 'a \rightarrow\ bool$ and $S :: 'b \rightarrow\ 'b \rightarrow\ bool$ describe type relations for $'a$ and $'b$ with the functions $f :: 'a \rightarrow\ 'b$ and $g :: 'a \rightarrow\ 'b$ respectively[6]. Other type relations are defined in a similar fashion. On top of these relations, Wadler's parametricity theorem [22] deals with relating a term with itself under its corresponding type relation. If this theorem is applied to parametric types, "free theorems" [22] can be derived by relating two arbitrary instances of the type for different parameters:

$$(t, t) \in \forall X.\ \mathcal{T} \Rightarrow \forall A ::'\ a \rightarrow\ 'b \rightarrow\ bool.\ (t_a, t_b) \in \mathcal{T}_A$$

In this theorem the term $t :: 'x\ T$ is of the polymorphic type T with type relation $\mathcal{T}$ that can be instantiated with a type parameter X. The instantiation is then formulated for two types $'a$ and $'b$ that can be related by A that also instantiates $\mathcal{T}$. All theorems derived from this approach are then valid for all terms of the used type. This finding relates to Reynolds' theorem, as the type is the environment and the theorem relates the values. The simplest theorem derived in this way is that all functions $f :: 'x \rightarrow\ 'x$

[6]The relations can be defined on different type pairs in general. This property is important for our later use of the function relator but not for pure parametricity.

for the polymorphic type $'x$ can only (modulo side effects) be the identity function. This result can be obtained by the following chain of equations. In the first step, the polymorphism of f is lifted from the type relation level to a general relation level.

$$\begin{aligned} f :: {'x} \to {'x} &\Rightarrow (f, f) \in \forall X.\ X \dashrightarrow X \\ &\Rightarrow \forall A :: {'a} \to {'b} \to bool.\ (f_a, f_b) \in A \dashrightarrow A \\ &\Rightarrow \forall A :: {'a} \to {'b} \to bool.\ \forall x\ y.\ A\ x\ y \longrightarrow A\ (f_a\ x)\ (f_b\ y) \end{aligned}$$

To make reasoning easier and more clear at this point, the relation descriptor A is exchanged for the function $\mathcal{A} :: {'a} \to {'b}$ that maps all values x of type $'a$ to a value y of type $'b$ if and only if $A\ x\ y$. Although $\mathcal{A}$ can not map all pairs of A in general, this specialization does not change the validity of the theorem that is derived below.

$$\begin{aligned} &\Rightarrow \forall \mathcal{A} :: {'a} \to {'b}.\ \forall x\ y.\ \mathcal{A}\ x = y \longrightarrow \mathcal{A}\ (f_a\ x) = (f_b\ y) \\ &\Rightarrow \forall \mathcal{A} :: {'a} \to {'b}.\ \forall x.\ \mathcal{A}(f_a\ x) = f_b(\mathcal{A}\ x) \\ &\Rightarrow \forall \mathcal{A} :: {'a} \to {'b}.\ \mathcal{A} \circ f_a = f_b \circ \mathcal{A} \end{aligned}$$

As f_a and f_b denote the same function f instantiated with different types and as the last equation has to hold for any $\mathcal{A}$, f has to be the polymorphic identity function $I :: {'x} \to {'x}$, where $I\ x = x$.

Another application of type relations is parametricity reasoning. This kind of reasoning aims for relating two terms of different types by finding a common polymorphic type relation. The important difference between the method of finding free theorems and reasoning with parametricity is that the former reasons about every possible implementation on the basis of the type, whereas the latter argues about two specific implementations. Parametricity reasoning can for instance be used to formulate correspondence theorems and, as a result, is used primarily in Chapter 5. A simple example correspondence theorem for two functions is

$$((=) \dashrightarrow (\lambda x\ y.\ x = y + 1))\ ((+)\ 1)\ I$$

where I is the polymorphic identity function and the correspondence of the two functions is denoted in their relation. In this case both functions work on the same types and, therefore, equality is possible as the first relation. More complex examples may require more complex relations arguing about different types. Especially monadic types can prove to be rather difficult to relate to. Relations between two monads were investigated by Karbyshev [9], whereas for our goal a non-monadic value has to be related to a monadic one. The necessary approach to achieve this is described in detail in Section 5.1.

2.4. Dynamic Programming and Memoization

Dynamic programming denotes both a mathematical optimization method as well as a programming pattern. Whereas the former was invented by Bellman and described in [1], the latter was developed by applying the mathematical approach to computer programming. In the following, we will only concentrate on the method for computer science.

Dynamic programming aims for solving optimization problems, i.e. problems with more than one solution, of which one is supposed to be the best or even optimal. This problem class includes the single-source shortest path problem on graphs or the knapsack problem, among others. Of these problems only a specific subset can be solved by a dynamic programming approach; the problems need to have an optimal substructure and overlapping sub-problems as defined by Cormen et al. in [3]. This means that the problem can be, on the one hand, solved by solving sub-problems first and that, on the other hand, some sub-problems depend on each other. These requirements are for instance met by recursive problems with more than one recursive call per level such as the fibonacci sequence. The solving pattern is then based on a clever approach to computing solutions to all necessary sub-problems while exploiting the dependencies among them to decrease the amount of necessary computations.

One method for implementing dynamic programming is memoization. Memoization is a special form of caching that was first introduced by Michie in [13]. It works on function application results, which are stored for future use. For this, a memoized wrapper function $=_m$ with an associated memory is introduced for a function $f :: {}'a \rightarrow {}'b$. Every call to f is then wrapped by $=_m$ by default. If the call arguments given to f are not in the memory, f is executed and the result is stored in the memory with the arguments as key. Otherwise, there already exists an entry in the memory indexed by these arguments and the corresponding result can be returned directly from the memory. As memoization introduces at least a constant amount of runtime overhead and, in the worst case, linearly growing memory consumption, it should only be used for expensive functions that are called often with a similar set of arguments. In general, this property is given for a dynamic programming approach. Other fields of application for memoization include parsers, especially top-down parsers and parser combinators. Memoized results can speed up their computations drastically, as these parsers may have to check an exponential number of possible parse trees for a given context free grammar. The gain achieved by this measure is, for instance, essential for parsing natural languages with parser combinators, an approach researched by Frost et al. [5]. The benefits of parser combinators harmonize well with the requirements enforced by natural languages, whereas the combinator's shortcomings are significantly improved by the memoization.

3. Related Work

Although monads are a rather old concept, the term "monadification" was first defined by Erwig and Ren in [4]. In their work, they introduce a monadification procedure based on context-dependent, syntactic rewriting. The authors claim that their automatic tool has several advantages over constructing monadic programs by hand. Besides being a generic tool that is reusable and versatile by design, it is also said to be more efficient and reliable than a human doing the transformation by hand. Erwig and Ren also specify two properties to describe the correctness of any monadification procedure: soundness and completeness. Together, they ensure that any monadified term describes the same behaviour as the original term wrapped in a *return* statement. As a result of this limiting definition, the authors stated that a sound monadification could not be found for all functions. Even though this conclusion applies to the presented approach, the given explanation does not hold for more modern monadification procedures in general. Another shortcoming of the correctness definition is the missing incorporation of monadic effects that may differ heavily between the monadified and the wrapped version of a code snippet. Despite this, the authors introduced a functionality to specify the order of monadification, as this detail has great influence on the application of monadic effects. Furthermore, the framework allows for context-dependent rewriting featuring its own formal language to describe this process in detail. Thus, the paper greatly influenced the research in this direction afterwards.

One result of this research is the *Monadification and Memoization* [24] framework developed by Wimmer et al., which is described in [25]. It aims at improving dynamic programming in Isabelle/HOL by automatically introducing memoization into pure programs. This effect is thereby implemented with a *state monad*, which is inserted automatically by a monadification procedure. Consequently, this transformation process is also formally verified by the tool. This property makes it a very powerful framework. On another note, this thesis describes an approach to extend said framework and is, consequently, directly based on it. As a result, the *Monadification and Memoization* framework will be examined in detail in Section 4.1 and Section 5.1. The framework includes also an interface for Imperative/HOL [11] and allows for the implementation of memoization with the associated heap monad. Moreover, the framework allows for bottom-up computation based on iterators defining the evaluation order of the monadic terms. Both features are not further investigated in this thesis, as they are not directly related to the topics we worked on.

On another note, our main idea for the extension is based on the work by Peter Lammich, which is used internally for the LLVM code generator described in [10]. The corresponding tool implements a lightweight monadification procedure, which had to be reverse engineered from its source code due to lack of formalization or concrete documentation. As a result, the method described in Section 4.2 produces similar output as Lammich's, yet has a different structure and more functions built-in.

A more recent paper by Lochbihler, [12], describes a monomorphic approach to monads in Isabelle. The introduced tool overcomes Isabelle/HOL's limitations in polymorphic data types, which hinder the development of a proper monad type class. The author shifted his focus therefor from value polymorphism to effect polymorphism, i.e. from a fixed monad with a polymorphic type parameter to a polymorphic monad with a fixed type parameter. This modification allows not only for powerful monad transformers (a rather unexamined topic in the context of automatic monadification), but also for reasoning about these monads with instance relations. These instance relations make it, among other things, possible to prove properties of complex monadic programs by exchanging the used monad with a simpler one. Nevertheless, the monomorphic approach does not quite meet the necessary requirements to be introduced into programs automatically by our tool. Due to our goal of being an extension to the *Monadification and Memoization* framework, our work has to keep the value polymorphic *state monad* and only add to this base. An approach with effect polymorphism is not capable of this and, hence, cannot be a part of our desired extension.

Part II.

Verified Monadification and Memoization

4. Monadification

The main objective of the *Monadification and Memoization* framework is to introduce memoization by the means of monadification with a *state monad* [25]. In contrast, our new extension approach aims at improving this method by allowing for generic monads and focuses more on the monadification procedure itself. Despite these differences, both versions follow the same basic principle; they take a pure (or at most partly monadic) function and wrap all function branches in their respective monadification procedures. These are introduced in detail in Section 4.1 and Section 4.2 respectively.

4.1. Current Monadification Procedure

The current monadification procedure is based on a call-by-value monadification style originating from [8]. The underlying monad is Isabelle/HOL's standard *state monad*[1] with a memory type $'m$ and a result type $'a$:

$$\textbf{datatype}\ ('m,\ 'a)\ \mathit{state} = \mathit{State}\ (\mathit{run\_state} : \ 'm \rightarrow\ 'a\ \times\ 'm)$$

Thus, the monad encapsulates a function that takes a memory, returns a new memory and a result and can be accessed by the *run_state* destructor. The corresponding *return* and *bind* operators are defined as follows:

$$\begin{aligned} \langle a \rangle &= \mathit{State}\ (\lambda M.\ (a, M)) \\ s\ \gg\!=\ f &= \mathit{State}\ (\lambda M.\ \textbf{case}\ \mathit{run\_state}\ s\ M\ \textbf{of}\ (v, M') \Rightarrow \mathit{run\_state}\ (f\ v)\ M') \end{aligned}$$

This definition of the *bind* combinator allows for the state to be threaded through the whole program.

4.1.1. Monadification with the State Monad

Building on top of this monad, the monadification procedure opts for similarity to the unmonadified version by using a special application-like operator called the "lifted function application operator" (infix as $\bullet$) [25]. It is defined as:

$$f_m \bullet x_m = f_m\ \gg\!=\ (\lambda f.\ x_m\ \gg\!=\ f)$$

[1]Located in the *State_Monad* theory from the Isabelle/HOL standard library.

To allow for this operator to be usable, the types of the monadified functions and terms are derived recursively by the following formalization based on [25]:

$$\begin{aligned} M(\tau) &:= ('m,\ M'(\tau))\ \textit{state} & \\ M'(\tau_1 \to \tau_2) &:= M'(\tau_1) \to M(\tau_2) & \\ M'(\tau_1 \oplus \tau_2) &:= M'(\tau_1) \oplus M'(\tau_2) & \text{where } \oplus \in \{+, \times\} \\ M'(\tau) &:= \tau & \text{otherwise} \end{aligned}$$

The corresponding monadification procedure generates two terms f_m and f'_m for a function f, of which only the first is of the described type, while the second is used internally. The f_m term is built as a monadified full η-expansion of the f'_m function. The method to arrive at f'_m has then been formalized in [25] as a set of rewrite rules, describing the core monadification operator $\rightsquigarrow$ (in descending order of priority):

$$\frac{e :: \tau_0 \to \tau_1 \to \cdots \to \tau_n \quad 2^e \cap \operatorname{dom} \Gamma = \emptyset \quad \forall i.\ M'(\tau_i) = \tau_i}{\Gamma \vdash e \rightsquigarrow \langle \lambda t_0.\ \langle \lambda\ t_1.\ \cdots \langle \lambda t_{n-1}.\ e\ t_0\ t_1\ \cdots\ t_{n-1} \rangle \cdots \rangle \rangle}\ \textsc{Pure}$$

$$\frac{\Gamma[x \mapsto \langle x'_m \rangle] \vdash t \rightsquigarrow t_m}{\Gamma \vdash (\lambda x :: \tau.\ t) \rightsquigarrow \langle \lambda x'_m :: M'(\tau).\ t_m \rangle}\ \lambda \qquad \frac{\Gamma \vdash e \rightsquigarrow e_m \quad \Gamma \vdash x \rightsquigarrow x_m}{\Gamma \vdash (e\ x) \rightsquigarrow (e_m \bullet x_m)}\ \textsc{App}$$

$$\frac{g \in \operatorname{dom} \Gamma}{\Gamma \vdash g \rightsquigarrow \Gamma(g)}\ \Gamma$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow \eta(t'_1) \quad \ldots \quad \Gamma \vdash t_n \rightsquigarrow \eta(t'_n)}{\Gamma \vdash \Delta\ t_1\ \ldots\ t_n \rightsquigarrow \langle \Delta\ t'_1\ \ldots\ t'_n \rangle}\ \textsc{Comb}$$

Here, Γ is a mapping of terms to their corresponding monadified versions and holds all already processed term pairs. Initially $f \mapsto f_m$ and $x \mapsto \langle x'_m \rangle$ are the sole entries in Γ for a given function branch $f\ x = t$. This is then monadified to $f'_m\ x'_m = t_m$ while $\Gamma \vdash t \rightsquigarrow t_m$ describes that t is monadified to t_m for the given Γ. The set 2^e denotes the set of subterms of e. The Δ term stands for any branching expression with branches t_1 to t_n like **if** ... **then** ... **else** ... or **case** ... **of** ..., whereas $\eta(\ldots)$ eta-expands the given term fully.

Accordingly, the monadification procedure is composed of both type and term rewrite rules. The effect of these rules are described by applying them to the *upto* function, which builds a list of consecutive integers between the given boundaries, in the following example. The types before and after the monadification are

$$\begin{gathered} \textit{upto} :: \textit{int} \to\ \textit{int} \to\ \textit{int list} \\ \textit{upto}_m :: ('m,\ \textit{int} \to\ ('m,\ \textit{int} \to\ ('m,\ \textit{int list})\ \textit{state})\ \textit{state})\ \textit{state} \end{gathered}$$

whereas the function definitions are

$$\mathit{upto}\ l\ u = \mathbf{if}\ (l \leq u)\ \mathbf{then}\ \mathit{Cons}\ l\ (\mathit{upto}\ (l+1)\ u)\ \mathbf{else}\ ([\,])$$

and

$$\begin{aligned}
\mathit{upto}_m &= \langle \lambda l.\ \langle \lambda u.\ \mathit{upto}'_m\ l\ u \rangle \rangle \\
\mathit{upto}'_m\ l\ u &= \mathit{ite}_m \bullet \langle l \leq u \rangle \bullet (\mathit{Cons}_m \bullet \langle l \rangle \bullet (\mathit{upto}_m \bullet \langle l+1 \rangle \bullet \langle u \rangle)) \bullet \langle [\,] \rangle \\
\mathit{ite}_m &= \langle \lambda a.\ \langle \lambda b.\ \langle \lambda c.\ \langle \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c \rangle \rangle \rangle \rangle \\
\mathit{Cons}_m &= \langle \lambda x.\ \langle \lambda xs.\ \langle \mathit{Cons}\ x\ xs \rangle \rangle \rangle
\end{aligned}$$

respectively. The subterms $\langle l \leq u \rangle$ and $\langle l+1 \rangle$ are shown in a simplified version for easier readability. This simplification is done in accordance with the second monad law. Furthermore, the definitions of ite_m and Cons_m are not generated by the monadification procedure directly, as the Γ rule would replace the non-monadic function calls or the *Pure* rule would not insert the innermost *return*.

4.1.2. Memoization with the State Monad

At this point, the generated terms carry only the given state with them without writing to or reading from it. Thus, the monadic effects of reading and writing need to be added to introduce memoization to the function. For this step, the monadified terms are wrapped in another function *retrieve_or_run*, which implements this functionality (cf. $=_m$ from Section 2.4). It is assumed that the memory $'m$ comes with the functions $\mathit{lookup} :: 'k \rightarrow\ ('m,\ 'v\ \mathit{option})\ \mathit{state}$ and $\mathit{update} :: 'k \rightarrow\ 'v \rightarrow\ ('m,\ \mathit{unit})\ \mathit{state}$. The memory's standard *lookup* and *update* functions may need to be wrapped in the state monad's *get* and *set* methods to fit these types. Additionally, the functions have to obey specific invariants to ensure memory consistency; a call to *lookup* must not add a mapping to the memory, whereas a call to *update* must not add any mapping except the one it was called with. If both functions comply with these conditions, the definition of $\mathit{retrieve\_or\_run} :: 'k \rightarrow\ 'v\ \mathit{state} \rightarrow\ 'v\ \mathit{state}$ can be given as:

$$\begin{aligned}
\mathit{retrieve\_or\_run}\ x\ t = \mathit{do}\ \{\ & r \leftarrow \mathit{lookup}\ x; \\
& \mathbf{case}\ r\ \mathbf{of}\ \mathit{Some}\ v \Rightarrow \langle v \rangle \\
& \quad \mid\ \mathit{None} \Rightarrow t\ \gg\!=\ (\lambda v.\ \mathit{update}\ x\ v\ \gg\!=\ \lambda\_.\ \langle v \rangle)\ \}
\end{aligned}$$

The argument t is here meant to be the lazily evaluated function term, as the method would otherwise follow the whole recursive branch without applying memoization. Memoization with the heap monad works in the same way.

4.2. New Monadification Procedure

Our new monadification method aims at simplifying the monadified functions in terms of operations as well as lifting the framework and its benefits to generic monads. Hence, our approach does not work with a fixed monad, like the *state monad*, anymore but rather with any monad M with exactly one type parameter $'a$. The corresponding *return* and *bind* functions have to be given. In general, every monad complying with the monadic laws can be used with our new method.

4.2.1. New Monadification Principle

In contrast to the former procedure, the new one changes the type of functions only marginally. It solely encapsulates the result type into the monad, resulting in, e.g.

$$map :: ('a \to\ 'b) \to\ 'a\ list \to\ 'b\ list$$

being altered to

$$map_\mu :: ('a \to\ 'b) \to\ 'a\ list \to\ 'b\ list\ M$$

where the subscript μ marks results of our new method. Due to this difference, the lifted function application operator $(\bullet)$ is not needed anymore. The monadification procedure on terms also works quite differently, as described by the following rules for the generic monad M where $\hookrightarrow$ denotes the new monadification procedure (ordered by descending priority):

$$\frac{\Gamma_\mu \vdash b[y/x] \hookrightarrow b'}{\Gamma_\mu \vdash (\lambda x.\ b) \hookrightarrow (\lambda x.\ b'[x/y])}\ \textsc{Abs}$$

$$\frac{\Gamma_\mu \vdash f \hookrightarrow f' \quad \Gamma_\mu \vdash x \hookrightarrow x' \quad \varphi(x)}{\Gamma_\mu \vdash f\$x \hookrightarrow \text{insert\_operand}(f', x')}\ \textsc{App\_var\_op}$$

$$\frac{\Gamma_\mu \vdash f \hookrightarrow f' \quad \neg\varphi(x) \wedge \mu(x)}{\Gamma_\mu \vdash f\$x \hookrightarrow \text{insert\_operand}(f',\ x)}\ \textsc{App\_monad\_op}$$

$$\frac{\Gamma_\mu \vdash f \hookrightarrow f' \quad \Gamma_\mu \vdash x \hookrightarrow x' \quad \neg\varphi(x) \wedge \neg\mu(x)}{\Gamma_\mu \vdash f\$x \hookrightarrow \text{bind\_operand}(f', x')}\ \textsc{App\_op}$$

$$\frac{x \in \text{dom}\ \Gamma_\mu}{\Gamma_\mu \vdash x \hookrightarrow \Gamma_\mu(x)}\ \textsc{Pre\_def} \qquad \frac{\text{term\_type}(x) \in \{Var, Free, Const, Bound\}}{\Gamma_\mu \vdash x \hookrightarrow x}\ \textsc{Id}$$

The above rules are then complemented by these auxiliary definitions:

$$\begin{aligned}\varphi(x) &\equiv \text{term\_type}(x) \in \{Var, Free, Bound\}\\ \mu(x) &\equiv (x :: 'a\ M \vee x :: 'a \rightarrow\ \cdots \rightarrow\ 'b\ M)\end{aligned}$$

$$\begin{aligned}\text{insert\_operand}(y \ggg\!= (\lambda tmp.\ f), x) &= y \ggg\!= (\lambda tmp.\ \text{insert\_operand}(f, x))\\ \text{insert\_operand}(f, x) &= \text{cond\_ret}(\text{dest\_ret}(f)\ x)\end{aligned}$$

$$\begin{aligned}\text{bind\_operand}(y \ggg\!= (\lambda tmp.\ f), x) &= y \ggg\!= (\lambda tmp.\ \text{bind\_operand}(f, x))\\ \text{bind\_operand}(f, x) &= \text{cond\_ret}(x) \ggg\!= (\lambda mp.\ \text{cond\_ret}(\text{dest\_ret}(f)\ x)))\end{aligned}$$

$$\begin{aligned}\text{cond\_ret}(x) &= \begin{cases} x & \text{if } x :: 'a\ M\\ \langle x\rangle & \text{else}\end{cases}\\ \text{dest\_ret}(\langle x\rangle) &= x\\ \text{dest\_ret}(x) &= x\end{aligned}$$

Every monadified top level term is afterwards also wrapped in a call to *cond_ret*. This is necessary, because function branches consisting of only one top level term would otherwise not be monadified at all. The environment Γ_μ is here, in fact, a different one than in the old procedure. By default, the mapping initially holds only the entry $f \mapsto f_\mu$ for a function f that should be monadified by the framework. The environment also holds all predefined monadic functions and operators and can be expanded by hand, which is a feature used for the label rewriting mechanism described below. The function *term_type* returns the topmost constructor of the given term defined by Isabelle/HOL's internal *term* datatype[2].

The monadification rules are each meant for specific cases. The *Abs* rule works on abstractions and lifts the monadification process into the body. For this, the abstraction variable is fixed for the monadification and later reestablished. This is necessary to keep the abstraction variables of the different abstraction levels from being interchangeable. Abstraction variables in Isabelle are implemented with de-Bruijn indices, which means that they are integer indices pointing to a specific abstraction level surrounding their occurrences. The numbering begins at 1 on the same level as the variable and increases for every enclosing layer. As the monadification procedure works in a depth first

[2]This data type represents Isabelle/HOL terms. Its constructors are *Var*, *Free*, *Bound*, *Abs*, *Const* and $ for applications. See, inter alia, [20].

manner, all abstraction variables that are reintroduced point automatically to the correct level. Next, the *App* rules handle applications depending on different attributes of the operand. If the operand is some kind of variable, as implied by φ, it does not need to be monadified any further and can directly be inserted in the monadified function term. Otherwise, the operands are distinguished according to whether they are monadic or not as described by μ. Monadic operands do not have to be monadified any further and, thus, can also be inserted directly. In the last case, the operand needs to be monadified and afterwards bound to the function term. Other than that, the *Pre_def* rule exchanges a term for a known monadified term from Γ_μ. Last but not least, the *Id* rule works on all remaining terms and returns them unmodified.

For both the binding as well as the insertion of operands, auxiliary functions are needed to work directly on the monadified term structure. The *insert_operand* function recurses to the deepest level of bound operands and adds its parameter to the function application there. As all already bound or inserted operands are applied before, the current operand is added at the correct position. In addition, the operand is lifted inside a *return* statement, if that is necessary to embed the function application. The *bind_operand* function leaps into the term structure the same way as *insert_operand* does. Though, it does not insert the operand in the deepest level, but instead introduces a new binding and a new abstraction. As this step can potentially break the correct abstraction variable indices, the same method as with the *Abs* rule is used. The correct wrapping or unwrapping of terms in *return* statements is achieved by the *cond_ret* and *dest_ret* functions respectively.

Due to the procedure working depth first, the first argument to a function application is bound with the outermost *bind*. This strategy is meant to help with, inter alia, replaying the termination proof later on in a similar way as the lifted function application operator does.

In the following, the *upto* function from above will be used as an example to show how our new method works:

$$upto_\mu :: int \to\ int \to\ int\ list\ M$$

$$\begin{aligned}
&upto_\mu\ l\ u = \\
&\langle l \leq u \rangle \ \ggg\!= (\lambda tmp.\ ((\langle l+1 \rangle \ \ggg\!= (\lambda tmpa.\ upto_\mu\ tmpa\ u)) \ \ggg\!= (\lambda tmpa.\ \langle Cons\ l\ tmpa \rangle)) \\
&\qquad\qquad \ggg\!= (\lambda tmpa.\ \langle [\,] \rangle \ \ggg\!= (\lambda tmpb.\ \langle \textbf{if}\ tmp\ \textbf{then}\ tmpa\ \textbf{else}\ tmpb \rangle)))
\end{aligned}$$

The monadified term has been slightly simplified for readability purposes as the term $\langle l+1 \rangle$ would otherwise be more verbose. Due to further simplifications obfuscating the monadification procedure, when done in compliance with the monad laws, the resulting term has not been changed any further. Although our method would allow for rewriting according to the second monad law, this step is currently not supported in the API, as the concrete effect on the proof tactics has not been investigated in full detail yet. The only case, where it is already used without problems, is the *App_var_op* rule.

4.2.2. Label Rewriting

Despite the missing simplification routine, our approach exhibits another important rewriting feature as a core functionality. It is called the label rewriting mechanism and works only on constants like *Cons*, *map* or simply 1. It is hidden in the *Pre_def* rule and requires a mapping from constant term names to fitting term building instructions. These instructions have to be given in the form of a function that takes the term to be exchanged as an argument and returns the new term as a result. It was decided to use functions, in order to give the user full control over the new term's type by making it possible to generate this information from the original term. Thus, the mechanism implements the Γ_μ mapping described above and contains initially only the function the framework currently works on. The other two core use cases for the label rewriting feature are the introduction of monadified higher-order operators and monadic effects. Both are described below.

4.2.3. Handling Higher-Order Monadic Combinators

Since our new monadification procedure changes only the body type of a function but not the expected argument types, it is necessary to handle higher-order functions differently in specific cases. These cases occur, if at least one function argument is monadified by the procedure, which can lead to a type problem. Let the definition of a function $example::'a \rightarrow\ 'b$ contain the subterm $map\ example\ l::'b\ list$ with an arbitrary list $l::'a\ list$ and, therefore, recurse indirectly through the *map* function. Then the monadified function $example_\mu::'a \rightarrow\ 'b\ M$ would contain the subterm $map\ example_\mu\ l::'b\ M\ list$. If the value of this term is used instead of a value of the expected type $'b\ list$, a type error would occur.

Our solution to this problem is a special auxiliary monadification procedure for handling higher-order operators. It runs our normal monadification procedure with an extended Γ_μ comprised of the new monadified definition of the operator with the required type and replacement values for the monadified arguments. This method then generates the corresponding higher-order monadic operator, which is used inside the normal monadification procedure. The *map* function serves as a good example for this method, as it is a widely used and easily understood higher-order operator. As the *map* function takes only one function argument, the type of the monadic combinator has to be

$$map^{\mu} :: ('a \to\ 'b\ M) \to\ 'a\ list \to\ 'b\ list\ M$$

(note the superscript μ instead of a subscript to distinguish between both new procedures). As the base case does not depend on the given function, it can be monadified directly as $map^{\mu}\ f_{\mu}\ [\,] = \langle [\,] \rangle$. For the non-empty list case $map\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (map\ f\ xs)$ the argument f is labeled to be exchanged for a monadic version in Γ_{μ} as:

$$f :: 'a \to\ 'b \mapsto f_{\mu} :: 'a \to\ 'b\ M$$

Given this Γ_{μ} mapping, the monadification procedure can be illustrated as a step-by-step term rewriting:

$$\begin{aligned} map^{\mu}\ f_{\mu}\ (Cons\ x\ xs) &\Rightarrow Cons\ (f_{\mu}\ x)\ (map^{\mu}\ f_{\mu}\ xs) \\ &\Rightarrow (map^{\mu}\ f_{\mu}\ xs) \ggg (\lambda tmp.\ Cons\ (f_{\mu}\ x)\ tmp) \\ &\Rightarrow (f_{\mu}\ x) \ggg (\lambda tmp.\ (map^{\mu}\ f_{\mu}\ xs) \ggg (\lambda tmpa.\ Cons\ tmp\ tmpa)) \end{aligned}$$

Even though the intermediate steps are not type correct, the final term has exactly the expected type and is, as a result, the correctly monadified branch of map^{μ}. With this term generated, the definition of map^{μ} is complete and can be used in the main procedure by utilizing the label rewriting mechanism. Thus, any call to *map* in the unmonadified function can be exchanged by a call to map^{μ} if needed. To achieve this effect, the corresponding function in Γ_{μ} has to either determine whether the monadic version is required or simply use it wherever possible. To return to the example from before, $map\ example\ l :: 'b\ list$ becomes $map^{\mu}\ example_{\mu}\ l :: 'b\ list\ M$. This value can then be bound to the next computation step and, thus, be used in the same fashion as the original subterm.

4.2.4. Introducing Monadic Effects

Another important use case of the label rewriting mechanism is the introduction of monadic effects to arbitrary places in the code. In contrast to rewriting higher-order operators, this method expects the labels to have no actual effect. For instance, let our tool introduce a writer monad for logging, which has a $write :: 'a \to\ string \to\ 'a\ M$ function. In this case the label, which will be exchanged with a call to *write*, has to be of type $'a \to\ string \to\ 'a$. As Isabelle/HOL uses a pure language, any function of this generic type can not depend on the second argument or have any side effects at all (cf. the parametricity theorem about the identity function in Section 2.3). Accordingly, a simple function of the necessary type could be $write\_label\ x\ s = x$, which returns the first argument unchanged for all string inputs s. Given this definition, an unmonadified usage of the label could be similar to the following:

$$f\ x = \mathit{upto}\ (\mathit{write\_label}\ 42\ \text{"the answer"})\ x$$

The monadified and unlabeled version would then be:

$$f_\mu\ x = (\mathit{write}\ 42\ \text{"the answer"}) \gg\!= (\lambda \mathit{tmp}.\ \mathit{upto}_\mu\ \mathit{tmp}\ x)$$

Other monadic effects, like memoization via *retrieve_or_run*, can be inserted in a similar fashion. Any introduced effect is required to be defined beforehand and to fit the type of the label.

In addition to these rather handy use cases for the label rewriting method, it is also possible to find other more seldomly required modifications. The system also allows for the monadification of mutual recursive functions and even more complex changes.

5. Parametricity Reasoning

As mentioned above, the *Monadification and Memoization* framework does not only introduce monadic wrappings automatically into programs, but proves this step to be correct as well. Our approach aims for generalizing these proofs while not changing their inner structure.

5.1. Basic Idea and Proof Mechanism

The main idea of the correctness proofs is to relate the original function to its monadified counterpart by a consistency relation $\Downarrow_R$ that, thus, works as an inter-type relation for monads. This relation asserts similar characteristics as the correctness definition from [4], though it does not work on values wrapped in *return* but on values unwrapped by *run_state* from within the monad. In addition, the relation ensures memory consistency (*cmem*) for a fixed function *f*, i.e. the memory contains only mappings $a \mapsto r$ for which $f\ a = r$ holds. $\Downarrow_R$ is then defined for an underlying relation R as:

$$\begin{aligned}\Downarrow_R v\ s = \forall m.\ cmem\ m \wedge inv_m\ m \longrightarrow&\\ &(\textbf{case}\ run\_state\ s\ m\ \textbf{of}\ (v',\ m') \Rightarrow\ R\ v\ v' \wedge cmem\ m' \wedge inv_m\ m')\end{aligned}$$

The memory invariant inv_m used here asserts that m is a correct memory, a property that only depends on the underlying implementation. Both the invariant and the consistency property have to be preserved by a run of the monad.

The proof itself is then built around the function relator $\dashrightarrow$ that was introduced in Section 2.3 and is used here to lift the consistency relation from values to functions. With this relator, it is possible to reason not only about monadified functions, but also about arbitrary monad combinators like *return* or the lifted function application operator $(\bullet)$ by parametricity reasoning as defined above:

$$(R \dashrightarrow \Downarrow_R)\ (\lambda x.\ x)\ return$$

$$(\Downarrow_{(R \dashrightarrow \Downarrow_S)} \dashrightarrow \Downarrow_R \dashrightarrow \Downarrow_S)\ (\lambda g\ x.\ g\ x)\ (\bullet)$$

Thus, the correspondence between these combinators and the matching non-monadic functions can be proven. In addition, both lemmata make evident how $\Downarrow$ is a fitting relation for applying parametricity reasoning to monadic values.

While the afore defined lemmata are mostly used in single proving steps, the main correspondence theorem for a monadified function f'_m can be formulated as:

$$((=) \dashrightarrow \Downarrow_{(=)})\ f\ f'_m$$

Other properties regarding both functions can be expressed in a similar fashion by exchanging the equality operator by an analogous relation. Nevertheless, it holds for any base relation that both f and f'_m have to be uncurried by a tuple abstraction, if f expects more than one argument. Thus, the correspondence theorem for *upto* is:

$$((=) \dashrightarrow \Downarrow_{(=)})\ (\lambda(l,\ u).\ upto\ l\ u)\ (\lambda(l,\ u).\ upto'_m\ l\ u)$$

5.1.1. Correspondence Proof by Induction

The correspondence proof is then built upon complete induction on the recursion structure of the monadified function[1].

Though, this approach is said to be non-trivial to automate [25]. The reasons for this are the congruence rules that are used by the function definition command **fun** to extract recursive function calls and the corresponding context. This information is crucial to the induction step, as it is used to generate the preconditions for the induction hypothesis. The congruence rules used by default work solely on non-monadic operators and can, therefore, not be applied to the hypothesis of the correspondence proof. As a result, specific congruence rules based on parametricity reasoning have to be introduced to resemble the default rules while directly relating monadic operators and their non-monadic counterparts. These new rules are structured for a function pair f and f'_m like the following example for *map*:

$$\frac{xs = ys \qquad \forall x.\ x \in set\ ys \Longrightarrow \Downarrow_S\ (f\ x)\ (f'_m\ x)}{\Downarrow_{list\_all2\ S}\ (map\ f\ xs)\ (map_m \bullet \langle f'_m \rangle \bullet \langle ys \rangle)}\ map\_map_m$$

Here, the *list_all2* relator is a function that lifts relations from single values to lists, whereas *set* builds a set of values from the list. The preconditions from map_map_m are structurally similar to those generated by the default congruence rules and, hence, fit in the proof scheme.

5.1.2. Termination

Besides proving the correctness by correspondence, the framework also proves the monadified function to terminate in the same way as the original. This property is

[1]This is, in general, the same as for the original function, because monadification does only modify single branches, but not the whole structure.

necessary to work with full parametricity, as both functions have to either return the same values for the same input or show the same meta behaviour. This meta behaviour consists of side effects, non-termination or error handling. As Isabelle/HOL uses a pure functional language not featuring error handling or side effects besides from within monads[2], only the non-termination has to be checked. Isabelle/HOL checks the termination of a function f by building a well-founded relation *f_rel* over values from its domain. This relation is based on the recursive function calls of f, thus relating the arguments of these calls. Accordingly, the tool tries to prove the equality of *f_rel* and the relation f'_m*_rel* for f'_m, as both functions share the same domain and are structurally equal with regard to recursive function calls. With this equality theorem the original termination proof can be replayed and, as a result, applied for the new function f'_m as well. The replay may fail if the monadification reorders the control flow too much leading to a disparity between *f_rel* and f'_m*_rel*. A failure could also be caused by one of the afore mentioned congruence rules, as they are sometimes used for the termination proof as well. In both cases, the framework tries to fall back to Isabelle's normal automated termination prover.

5.2. Generalized Proof Method

As one declared goal of our expansion approach is to lift as much functionality as possible to a generic level, both the correspondence and the termination proof have to be generalized. The underlying proof structure can be kept for this, because it is based around specific lemmata that are used from within generic proof tactics. These Isabelle/ML tactics form the core of the described induction proof. The necessary lemmata for the correspondence proof can be generated from a small set of monad-specific information. This set contains basic information like the used monad type M with its corresponding *bind* and *return* combinators. In addition, a fitting consistency relation $\Downarrow'_R$, asserting all needed invariants as well as value correctness under R, has to be given. This kind of relation cannot be generated for a given monad (even if no invariants are required), because it is not generally possible to extract a value from a monad without knowing the monad's internals. Accordingly, correspondence theorems for *bind* and *return* featuring $\Downarrow'_R$ must be included in the given information set. Formally these conditions can be stated as (for a given relator $R :: 'a \rightarrow\ 'b \rightarrow\ bool$ where $'a$ and $'b$ could also be the same type):

$$\Downarrow'_R :: 'a \rightarrow\ 'b\ M \rightarrow\ bool$$

[2]These monads will be left intact, which is why their correspondence is proven by the correctness proof. In addition, all effects within the newly introduced monad are automatically discarded for any parametricity proofs.

Lemma 1 *Let inv_m be the monad's correctness invariant asserted by $\Downarrow'_R$, then the following rule can be used for an application of $\Downarrow'_R$ on a value wrapped in return:*

$$\frac{inv_m \quad R\ x\ y}{\Downarrow'_R\ x\ \langle y \rangle}\ \textsc{crel\_return}$$

Lemma 2 *Let $f :: 'a \rightarrow\ 'c$ and $mf :: 'b \rightarrow\ 'd\ M$ be functions working on $v :: 'a$ and $m :: 'b\ M$ with $S :: 'c \rightarrow\ 'd \rightarrow\ bool$, then the following rule holds:*

$$\frac{\Downarrow'_R\ v\ m \quad (R \dashrightarrow \Downarrow'_S)\ f\ mf}{\Downarrow'_S\ (f\ v)\ (m\ \gg\!=\ mf)}\ \textsc{crel\_bind}$$

Both lemmata rely strongly on the inner workings of *bind* and *return*. In general, both operators are expected to be defined by the **definition** command or else come with unfolding lemmata called *bind_def* and *return_def*. Moreover, congruence lemmata for each used higher-order combinator are required as well and need to be of the same form as the auxiliary congruence theorems used in the old framework version. The associated congruence rule for *map* can be formulated as:

$$\frac{xs = ys \quad \forall x.\ x \in set\ ys \Longrightarrow \Downarrow'_S\ (f\ x)\ (f_\mu\ x)}{\Downarrow_{list\_all2\ S}\ (map\ f\ xs)\ (map^\mu\ f_\mu\ ys)}\ map\_map^\mu$$

Our extension idea is meant to already provide a small number of higher-order monadic combinators including *map*, *fold* and *comp* with corresponding congruence rules. While these can simply be selected by the user, any other needed combinators have to be given to the tool along with matching congruence rules. In addition, rules to lift the monadic effects introduced by the label rewriting mechanism are needed. Together, all lemmata, rules and the consistency relation form the basis of the main proof locale, which, as a result, exhibits all necessary lemmata and theorems for the correspondence proof. Because normal Isabelle locales are not expressive enough to work for generic monads (cf. Section 2.1), the mentioned locale has to be newly generated for every monad.

Whereas the correspondence proof is taken care of by the generated locale, proving termination of monadified functions can still be handled the same as before. This step may even become easier in some cases, as the monadification by our new extension is more lightweight. This approach is made possible by the above-defined binding order of arguments for our new monadification procedure. The method of binding the arguments in reverse order of application, i.e. the first argument is located in the outermost *bind*, ensures the same evaluation order as in the original function. This works analogously to the lifted function application operator and is, in general, sufficient for the termination proof.

6. Comparison by Example

To conclude the reasoning about our new extension idea to the *Monadification and Memoization* framework, we compare the changes it would bring in more detail in the following. At first, both the monadification and the proof process are compared, then a real world example is examined to show the usability of our approach.

6.1. General Comparison

The biggest change introduced by our proposed extension is the ability to generically handle almost any monads. As a result, the framework's scope has shifted and expanded from only simplifying dynamic programming to the processing of almost unlimited possible use cases. However, this modification comes at the price of an increased implementation overhead for users. The effort of providing all necessary functions and theorems can increase significantly for some applications. This increase is especially due to the need for term rewriting rules and congruence lemmata for higher-order combinators and the monadic effects that are introduced by the label rewriting mechanism. Although applications using the old framework version only need to provide slightly more auxiliary information for the label mechanism, they may have to be adjusted and extended by hand. The real-world example below shows the extent of the modifications needed.

Another crucial change concerns the monadification procedure. Not only does our new approach waive the need for the lifted function application operator, but it also monadifies with less overhead and less impact on function types. For example, let *fib* denote an "unusual [sic] definition of the Fibonacci sequence" [25]:

$$\mathit{fib}\ n = 1 + \mathit{sum}\ ((\lambda f.\ \mathit{map}\ f\ [0..n-2])\ \mathit{fib})$$

$[x..y]$ is a shorthand notation for the *upto* function introduced above. For simplicity, only the subterm $(\lambda f.\ \mathit{map}\ f\ [0..n-2])\ \mathit{fib}$ is examined further. For this term, the old monadification procedure produces

$$\langle \lambda f'_m.\ \mathit{map}_m \bullet \langle \mathit{fib}'_m \rangle \bullet [0..n-2]_m \rangle \bullet \langle \mathit{fib}'_m \rangle$$

where $[0..n-2]_m = \mathit{upto}_m \bullet \langle 0 \rangle \bullet (\langle \lambda x.\ \langle \lambda y.\ \langle x - y \rangle \rangle \rangle \bullet \langle n \rangle \bullet \langle 2 \rangle)$ (based on [25]). As the new monadification procedure does not use the lifted function application operator, the terms above have to be unfolded in regard to this operator to be comparable:

$$\langle \lambda f'_m.\ (map_m \gg= (\lambda g.\ \langle f'_m \rangle \gg= g)) \gg= (\lambda h.\ [0..n-2]_m \gg= h) \rangle \\ \gg= (\lambda f.\ \langle fib'_m \rangle \gg= f)$$

$$[0..n-2]_m = (upto_m \gg= (\lambda g.\ \langle 0 \rangle \gg= g)) \gg= (\lambda f.\ (((\langle \lambda x.\ \langle \lambda y.\ \langle x-y \rangle \rangle \rangle \\ \gg= (\lambda i.\ \langle n \rangle \gg= i)) \gg= (\lambda h.\ \langle 2 \rangle \gg= h)) \gg= f)$$

There are 14 applications of *bind* in total, of which at least six could be trivially simplified due to the second monad law. On the other hand, our new monadification procedure generates the following shorter result for the same input term:

$$\langle fib_\mu \rangle \gg= (\lambda f.\ (\langle 0 \rangle \gg= (\lambda tmp.\ \langle 2 \rangle \gg= (\lambda tmpa.\ \langle n - tmpa \rangle \\ \gg= (\lambda tmpb.\ upto_\mu\ tmp\ tmpb)))) \gg= (\lambda tmp.\ map^\mu\ f\ tmp))$$

Although this new term may be harder to understand than the old version using $(\bullet)$, it contains nine fewer applications of *bind*, resulting in a total of five. This finding aids our claim that the new method is in fact more lightweight.

In contrast to these major changes, the proof method does not change much. Both framework versions provide methods to automatically prove monadification correctness. Both also make heavy use of parametricity reasoning and congruence rules to handle induction proofs. In addition, both versions rely on replaying the termination proof or falling back to Isabelle's standard tool. The main difference between the two versions is the theorems used in the proofs, which depend only on the used monad. Another difference is where these theorems come from. Whereas the current framework version only uses built-in lemmata and relations, our new extension approach is meant to generate the necessary set from custom user-input information. Although this generation process is yet to be fully implemented, there are no major problems regarding this step in theory.

6.2. Real-World Example: Bellman-Ford

The Bellman-Ford algorithm solves both the single-source as well as the single-sink shortest path problem for weighted directed graphs. As both problems can be reformulated into the other by simply changing the edge directions, we will concentrate on the single-sink variant. The graph properties are modelled as a list of node weights $W :: nat \rightarrow nat \rightarrow int$ where all n nodes are denoted by natural numbers. The sink node is fixed as $t \in \{1, \ldots, n\}$. The algorithm then calculates the weight of the shortest path from each source $j \in \{1, \ldots, n\}$ to t for a maximum number of edges k along the path. To calculate the weight for a specific j and a specific $k+1$, the algorithm first computes the shortest path weight for k steps from j and then all minimal weights for the paths from each other node $i \in \{1, \ldots, n \mid i \neq j\}$ to t for k steps plus the weight from j to i. The minimum of all these paths is then the shortest path from j to t in at most $k+1$ steps. As this procedure exhibits an optimal substructure and overlapping sub-problems by definition, it is an instance of dynamic programming and can be optimized by using memoization. The basic implementation of the algorithm is defined as:

$$\begin{aligned} bf\ 0\ j =&\ (\textbf{if}\ t = j\ \textbf{then}\ 0\ \textbf{else}\ \infty) \\ bf\ (Suc\ k)\ j =&\ min\_list\ (Cons\ (bf\ k\ j)\ (map\ (\lambda i.\ W\ j\ i + bf\ k\ i)\ [1..n])) \end{aligned}$$

Here, *min_list* finds the minimum value inside a list and returns it, whereas $[x..y]$ denotes again the *upto* function from before.

Due to this simple structure, the Bellman-Ford algorithm was also used as an example in the old framework [24, 25], where the monadified version of *bf* was given as (adjusted to our formalization):

$$\begin{aligned} bf_m =&\ \langle \lambda k.\ \langle \lambda j.\ bf'_m\ k\ j \rangle \rangle \\ bf'_m\ 0\ j =_m&\ ite_m\ \langle t = j \rangle\ \langle 0 \rangle\ \langle \infty \rangle \\ bf'_m\ (Suc\ k)\ j =_m&\ \langle \lambda xs.\ \langle min\_list\ xs \rangle \rangle \bullet (\langle \lambda x.\ \langle \lambda xs.\ \langle Cons\ x\ xs \rangle \rangle \rangle \bullet (bf_m \bullet \langle k \rangle \bullet \langle j \rangle) \bullet \\ &\ (map_m \bullet \langle \lambda i.\ \langle \lambda x.\ \langle W\ j\ i + x \rangle \rangle \bullet (bf_m \bullet \langle k \rangle \bullet \langle i \rangle) \rangle \bullet [1..n]_m)) \end{aligned}$$

Here $=_m$ denotes the application of memoization to the term that follows as defined above.

As the Bellman-Ford algorithm is ideal for memoization, our new monadification procedure is here used with the memoization *state monad*. Therefore, the monadified term generated by our procedure is similar, yet more lightweight:

$$\begin{aligned} bf_\mu\ 0\ j =_m&\ \langle \textbf{if}\ t = j\ \textbf{then}\ 0\ \textbf{else}\ \infty \rangle \\ bf_\mu\ (Suc\ k)\ j =_m&\ (bf_\mu\ k\ j\ \gg\!=\ (\lambda tmp.\ (\langle [1..n]_\mu \rangle\ \gg\!=\ map^\mu\ (\lambda i.\ bf_\mu\ k\ i\ \gg\!= \\ &\ (\lambda tmpa.\ \langle W\ j\ i + tmpa \rangle)))\ \gg\!=\ (\lambda tmpa.\ \langle Cons\ tmp\ tmpa \rangle))) \\ &\ \gg\!=\ (\lambda tmp.\ \langle min\_list\ tmp \rangle)) \end{aligned}$$

This function definition has only been slightly simplified by the frameworks internal simplification method (based solely on the second monad law) to achieve a higher readability. Yet, even the original version would use significantly less *bind*s than the old framework. In contrast to this decrease in structural overhead, the setup overhead for the user is increased. Decisive for this fact is the utilization of map^{μ}. Whereas the memoization can be lifted by lemmata already given within the old framework version, a congruence lemma for *map* has to be proven by hand. Fortunately both the label function and the lemma for *map* are rather simple and can be constructed using a simple term exchange function and lemmata about the *state monad* and its operators respectively.

Both implementations try then to prove the theorem

$$((=) \dashrightarrow \Downarrow_{=})\ (\lambda(x,y).\ bf\ x\ y)\ (\lambda(x,y).\ bf_M\ x\ y)$$

where bf_M is either bf'_m or bf_μ respectively. Thus, the proof is done by induction and makes use of the described locales. As the locales for the new generic proof method can not be generated yet, it had to be implemented by hand. However, there were not many theorems that had to be given due to the great number of helpful theorems from the old version. The proof for our new implementation directly relates every *bind* with a corresponding function application in the original function and every *return* with the non-monadic value. For instance, the outermost call to *min_list* in the non-monadic function relates to the outermost *bind* of the monadified function. As a result, both the function relation for *min_list* and $(\lambda tmp.\ \langle min\_list\ tmp\rangle)$ and the value relation for the arguments have to be proven as next steps. This approach is then carried out until all subgoals are proven and all subterms are related. Despite this simple method, which works nearly the same as in the old version, our new tool has a certain limitation. Normally the rule about *bind* is automatically matched to the outermost function application in the non-monadic term. Unfortunately, the way the *bind* rule is defined and applied either leads to always matching the identity function, instantiates some relations not correctly or does both. This shortcoming can be overcome by trivially adjusting the applied rules to match the current goal by hand or in theory by a specific proof tactic that has yet to be implemented.

Despite all differences, both implementations are proven to correspond to *bf*, which in turn is also proven to be a correct implementation of the Bellman-Ford algorithm's core functionality.

Part III.

Discussion and Conclusion

7. Discussion

The main goal of our approach to extend the *Monadification and Memoization* framework was to decrease the structural overhead of the monadification method while also allowing for a greater number of different monads. The extent, to which these objectives have been reached, is the subject matter of the following sections.

7.1. Achievements

In general, the structural monadification overhead primarily depends on the number of introduced *bind* applications. For most monads the *bind* operation is more expensive than the *return* operation, as it has to ensure the correct threading of all monadic effects. Hence, the number of *bind* statements is an appropriate heuristic to measure overhead introduced by our procedure. The simplest take on this heuristic compares the two *bind* applications within the lifted function application operator to not more than one in our new method. Especially the examples in Chapter 6 give a good overview of how much this difference matters. At the same time, our new method has maintained provability, which becomes particularly evident in the example in Section 6.2. To summarize, the structural overhead introduced for all monadified terms has been successfully reduced by our new approach. Despite this theoretical gain, the concrete improvement in runtime has not been measured. As we suppose a direct causality between the number of function applications and the associated runtime, a positive outcome is expected nevertheless.

The generalization to more monads is not quantitatively measurable, as an unbounded number of monad instances could exist or be implemented. As a result, we see the general possibility to use other monads as the *state monad* as a success. In theory, most standard monads can be used with little to no implementation overhead for the user regarding the monadification process itself. In practice, the *state monad*, a simple *writer monad*, a *list monad* and some other simple monads were proven to be usable. For these monads, both the introduction by our monadification procedure and the automated proving mechanism did not come with particular problems. As our monadification method depends not on the underlying monad type and all effects other than *bind* and *return* can be inserted with the label rewriting mechanism, the applicability of a specific monad is based primarily on provability of all necessary lemmata.

Although, for some examples like the *list* and the *option* monads, a somewhat different approach to the consistency relation requires more effort by the user. Both include a type constructor that envelopes no value at all (the empty list *Nil* and the *None* constructor). Handling these cases accordingly may not be necessary for all possible use cases. Nevertheless, there are use cases that rely on these constructors and need to be handled with care. If, for example, the monadic effect of a *None* is introduced with the label rewriting system, all following values that could have been embedded in a *Some* are also reduced to *None*. This effect is important to the proof mechanism, as there are only two generally possible ways of handling the *None* in $\Downarrow'$: the relation has to either relate all elements with *None* or none of them. Whereas the first case may be only useful if the user tries to implement non-total functions by hand in a similar fashion than Isabelle/HOL does by default, the second case is the correct relation for most use cases. As the relation that does not relate *None* would fail for every occurrence of this constructor, the correspondence of the monadified function can only be proven for a program thread with no *None* at all. In most cases, this assertion corresponds to a computation without errors. Hence, the missing validity for *None* results is expected. In a similar way, other monads that can encode erroneous behaviour by a special constructor have to be handled with care, as they cannot be proven for all possible cases.

Though, it can be noted that the possible usage of the monads mentioned above is a successful extension to the original framework.

7.2. Shortcomings

Although our extension approach resulted in a new monadification method that meets both primary goals, it also comes with a certain number of limitations. These are primarily about the increased implementation overhead for the user of the framework and are described in detail in the following.

A major shortcoming of our approach is directly linked to the Isabelle/HOL system: the missing possibility of defining a type class for monads. Such a type class would make reasoning about monads a lot easier, as it provides a lot of constraints and assertions by default. As Isabelle/HOL does not come with the capabilities to provide such a type class, we were forced to handle the generalization to more monads by automatic generation from a custom set of information. This approach requires more work by the user and leads, for this reason, to a not optimally usable framework. Thus, a better formalization idea for monads in Isabelle/HOL has yet to be found.

Another shortcoming is based on the label rewriting mechanism and how it can be used. If this method is used for several cases of higher-order operators or monadic effects, at least one lemmata is required for each of these cases. As the user has to come up with all of these lemmata by himself, the users abilities in proving them can be a limiting factor. On another note, the user can introduce arbitrary many effects into the program, which may result in arbitrary many bugs or faults that are rather difficult to debug. This was not the case for the old framework and is, thus, a limitation to the usability of our approach.

Last but not least, the biggest shortcoming of this thesis is the missing implementation of our approach in the context of a whole framework extension. Only the core functionalities have been implemented as a reference implementation. Other parts that would need a direct binding to the framework have at least been tried out by hand for a small number of examples. Therefore, more shortcomings of our approach may become evident later. Though, it is quite likely, that these regard only parts not introduced in this thesis, as all main ideas have turned out to verifiably work for most cases.

8. Conclusion

We have introduced our proof-of-concept for an extension to the *Monadification and Memoization* framework [24] for Isabelle/HOL. Our goal for this proof-of-concept was to find a new monadification procedure that, firstly, has less monadification overhead than the old method and, secondly, allows for the usage of other monads than the *state monad*. At best, this would trivially include all possible monads, but exceptions for a small number of specific cases would be ok either. As the framework is implemented in Isabelle/HOL, our new procedure was also expected to allow for an automatic correctness proof.

Our approach was then built around a simple monadification idea, i.e. binding the operands to the operators and adding *return*s where needed. Hereby, the first operand is bound by the outermost *bind*. The rest of the operands is bound in the same way until the last operand is bound with the innermost *bind*. The procedure also simplifies the bindings to a certain degree, as all variables are not bound but rather directly added to the operator. This works correctly, as variables cannot be monadified any other way than by embedding them in a *return*. Therefore, the second monad law allows for this simplification step. In addition to this basic monadification method, our procedure also exhibits a facility to exchange certain terms for predefined monadic counterparts called the label rewriting mechanism. The standard use case for this is the correct handling of recursive calls. Other than that, higher-order operators using monadified functions and monadic effects are supposed to be introduced by the label rewriting method. We implemented the procedure as a reference implementation and tested it for different term structures and monads. We then adjusted the frameworks proving mechanism to match our new monadification procedure. Whereas the top level tactics based on parametricity reasoning could be kept the same, the underlying lemmata and proof information are now generated for each monad separately. We tried this mechanism out for different monads and functions and could even repeat the monadification of the Bellman-Ford algorithm.

With all examples being done, we concluded that our approach to extend the framework was successful. Our new monadification algorithm is able to monadify the same programs the old framework could. While doing this, it uses at most two fewer applications of *bind* for a function application than the old framework did with its lifted function application operator. Our approach also allows for custom monads and introduces the corresponding monadic effects via the label rewriting mechanism. Despite the changes in the monadification method, it is still possible to prove the correspondence of a monadified function and its original counterpart by extended parametricity reasoning. The example of the Bellman-Ford algorithm, where we introduced memoization and proved the resulting function to be correct, concludes our work and fulfills our goal of keeping the functionality of the old framework.

8.1. Future Work

We have provided an approach to improve and extend the *Monadification and Memoization* framework. Despite this, a lot of work is still to be done. In addition to completing the implementation, there are several points that we would need to look into in more detail. First and foremost, the proof tactics have to be improved to work with all generated and provided lemmata and still result in correct interim results. The current problems with this step are described in Section 6.2. Next, the label rewriting mechanism should be investigated further, as it can be used to implement nearly arbitrary rewriting methods as of now. The concrete effects on the proofs introduced by the rewriting should also be a future research topic. Depending on the findings, the mechanism may have to be limited to certain use cases only. In general, the focus of future research should be on how more different monads can be introduced and how this affects the proving mechanism. Based on this, more sophisticated structures as monad transformers may be worthy of further examination, as well. For this, the monomorphic approach by Lochbihler [12] could also be included in the framework as an optional extension. Moreover, the other features of the framework, namely the imperative heap monad and the bottom-up computation, should be investigated in combination with our improvements. Last but not least, after all the above steps are done, a new entry to the Archive of Formal Proofs[1] has to be submitted. Thereafter, the framework extension we aimed for can be seen as ready to use.

[1]AFP for short, see `https://www.isa-afp.org/`.

Bibliography

[1] R. Bellman. *Dynamic programming*. Princeton, NJ: Princeton Univ. Press, 1957.

[2] A. Church. “A formulation of the simple theory of types.” In: *A Formulation of the Simple Theory of Types* 5.2 (1940), pp. 56–68. ISSN: 0022-4812. DOI: `10.2307/2266170`.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. 3. ed. Cambridge, Mass.: MIT Press, 2009. ISBN: 9780262033848.

[4] M. Erwig and D. Ren. “Monadification of functional programs.” In: *Science of Computer Programming* 52.1 (2004), pp. 101–129. ISSN: 0167-6423. DOI: `10.1016/j.scico.2004.03.004`.

[5] R. A. Frost, R. Hafiz, and P. Callaghan. “Parser Combinators for Ambiguous Left-Recursive Grammars.” In: *Practical aspects of declarative languages*. Ed. by P. Hudak and D. S. Warren. Vol. 4902. Lecture Notes in Computer Science. Berlin: Springer, 2008, pp. 167–181. ISBN: 978-3-540-77441-9. DOI: `10.1007/978-3-540-77442-6_12`.

[6] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. USA: Cambridge University Press, 1993. ISBN: 0521441897.

[7] M. J. C. Gordon. *Edinburgh LCF*. Lecture Notes in Computer Science. Berlin: Springer, 1979. ISBN: 3540097244.

[8] J. Hatcliff and O. Danvy. “A generic account of continuation-passing styles.” In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Ed. by H.-J. Boehm. New York, NY: ACM Press, 1994, pp. 458–471. DOI: `10.1145/174675.178053`.

[9] A. Karbyshev. “Monadic Parametricity of Second-Order Functionals.” Dissertation. München: Technische Universität München, 2013.

[10] P. Lammich. “Generating Verified LLVM from Isabelle/HOL.” In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by J. Harrison, J. O’Leary, and A. Tolmach. Vol. 141. Leibniz International Proceedings in Informatics. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, 22:1–22:19. DOI: `10.4230/LIPIcs.ITP.2019.22`.

[11] P. Lammich. "Refinement to Imperative HOL." In: *Journal of Automated Reasoning* 62.4 (2019), pp. 481–503. ISSN: 1573-0670. DOI: `10.1007/s10817-017-9437-1`.

[12] A. Lochbihler. "Effect Polymorphism in Higher-Order Logic (Proof Pearl)." In: *Journal of Automated Reasoning* 63.2 (2019), pp. 439–462. ISSN: 1573-0670. DOI: `10.1007/s10817-018-9476-2`.

[13] D. Michie. ""Memo" Functions and Machine Learning." In: *Nature* 218.5136 (1968), pp. 19–22. ISSN: 0028-0836. DOI: `10.1038/218019a0`.

[14] E. Moggi. "Notions of computation and monads." In: *Information and Computation* 93.1 (1991), pp. 55–92. ISSN: 08905401. DOI: `10.1016/0890-5401(91)90052-4`.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. `https://isabelle.in.tum.de/doc/tutorial.pdf`, Updated Version used as Isabelle/HOL Tutorial. Berlin: Springer, 2002.

[16] B. O'Sullivan, J. Goerzen, and D. B. Stewart. *Real world Haskell: Code you can believe in*. 1. ed. Beijing: O'Reilly, 2010. ISBN: 9780596514983.

[17] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science. Berlin: Springer, 1994. ISBN: 3540582444.

[18] L. C. Paulson. "Natural deduction as higher-order resolution." In: *The Journal of Logic Programming* 3.3 (1986), pp. 237–258. ISSN: 07431066. DOI: `10.1016/0743-1066(86)90015-4`.

[19] J. C. Reynolds. "Types, Abstraction and Parametric Polymorphism." In: *Information processing 83 : proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. Mason and International Federation for Information Processing. Amsterdam: North-Holland, 1983, pp. 513–523.

[20] C. Urban. *The Isabelle Cookbook: A Gentle Tutorial for Programming Isabelle/ML (draft)*. `http://talisker.nms.kcl.ac.uk/cgi-bin/repos.cgi/isabelle-cookbook/raw-file/tip/progtutorial.pdf`. Last Accessed: 2020-02-02. 2019.

[21] P. Wadler. "The essence of functional programming." In: *Conference record of the Nineteenth Annual ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*. Ed. by R. Sethi. ACM conference proceedings series. New York, NY: ACM Press, 1992, pp. 1–14. ISBN: 0897914538. DOI: `10.1145/143165.143169`.

[22] P. Wadler. "Theorems for free!" In: *Functional Programming Languages and Computer Architectures*. ACM Press, 1989, pp. 347–359.

[23] M. M. Wenzel. "Isabelle/Isar — a versatile environment for human-readable formal proof documents." Dissertation. München: Technische Universität München, 2002.

[24] S. Wimmer, S. Hu, and T. Nipkow. “Monadification, Memoization and Dynamic Programming.” In: *Archive of Formal Proofs* (May 2018). `http://isa-afp.org/entries/Monad_Memo_DP.html`, Formal proof development. ISSN: 2150-914x.

[25] S. Wimmer, S. Hu, and T. Nipkow. “Verified Memoization and Dynamic Programming.” In: *Interactive theorem proving*. Ed. by J. Avigad and A. Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Berlin: Springer, 2018, pp. 579–596. DOI: `10.1007/978-3-319-94821-8_34`.