UNIVERSITY OF KASSEL

DEPARTEMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Bachelor's Thesis

Formal Verification of Different Semantics for an Abstract Higher-Order Fixpoint Algebra in Isabelle/HOL

Author:	Lars-Eric Marquardt
First Examiner:	Prof. Dr. Martin Lange
Second Examiner:	Dr. Tom Hanika
Advisor:	Dr. Florian Bruse

23.12.2022

Eigenständigkeitserklärung
Ich versichere hiermit, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Declaration
I hereby confirm that this thesis is my own work and I have documented all sources and material used.

Kassel, 23.12.2022 Lars-Eric Marquardt

Acknowledgments

I want to thank Prof. Martin Lange and Florian Bruse for allowing me to work on this project. A big thank you goes out to Florian Bruse for the countless hours spent on discussing how to approach this formalization and the long times spent waiting for Isabelle to find something useful or crash, yet again.

I would also like to thank Norbert Hundeshagen, who had a big impact on my journey of becoming a computer scientist.

Lastly, I would like to express my gratitude for the Isabelle Zulip where a lot of technical questions regarding Isabelle were answered.

Table of Contents

Chapter 1

Introduction

A proof assistant (also called interactive theorem prover) is an interactive computer system used for carrying out mathematical proofs on a computer. Traditionally, mathematical proofs are carried out on paper and then are peer-reviewed to ensure they are correct. But often these proofs leave out some details, contain misstatements, missing hypotheses or unstated background assumptions [1]. In contrast, *formal proofs* contain every single application of inference rules and state exactly which axioms, assumptions or previously deduced facts the rules are applied to. However, formal proofs on paper are huge and it is unfeasible for humans to read them. Proof assistants facilitate these problems by carrying out parts of the proof automatically. They are also called *interactive* theorem provers, because they require an interplay between the user and the computer as opposed to having full automation provided by *automatic* theorem provers. Fully automated theorem proving is not expressive enough to meet the needs of formal verification [2] but is used in conjunction with proof assistants.

In the past, proof assistants have been used to formalize a variety of different domains, not only limited to the area of mathematics. In the area of pure mathematics, notable projects are the proof of the *Kepler Conjecture* [3], a verification of the *Prime Number Theorem* [4], and proving the *Four Color Theorem* [5]. Proof assistants have also been used for system and hardware verification. The *Verisoft* [6] project formalized a whole computer system from the hardware up to the microkernel, and a compiler. The *seL4* project is a formal verification of the L4 microkernel [7]. In the area of programming languages, proof assistants have been used in the verification of *CompCert* [8], a formally verified C compiler, and to formalize a subset of Java [9].

The goal of this thesis is the formal verification of different semantics of a higher-order fixpoint algebra presented in [10]. The algebra defines terms in a lattice and is used to express fixpoint evaluation problems which occur in many areas of computer science. Its semantics is denotational based on the Knaster-Tarski Fixpoint Theorem, but one can also employ an algorithmic semantics based on the Kleene Fixpoint Theorem making use of fixpoint iteration. In addition, Bruse et al. [10] defined an on-the-fly model-checking algorithm which localizes fixpoint iteration allowing to only compute the values of as few variables as possible in the best case.

This thesis formalizes the abstract higher-order fixpoint algebra and its different semantics in the proof assistant Isabelle/HOL and provides a formal proof that the denotational and algorithmic semantics are equivalent. Furthermore, the local model-checking algorithm is formalized as a recursive program.

The structure of the thesis is as follows:

- Chapter 2 introduces the needed mathematical background and the higher-order fixpoint algebra with its semantics.
- Chapter 3 introduces Isabelle and provides an introduction to theorem proving in Isabelle.
- Chapter 4 presents the formalization in Isabelle and explains and highlights some of the proofs and hurdles.
- Chapter 5 concludes this thesis.

Chapter 2

Preliminaries

This chapter provides a short introduction to the needed mathematical background for this thesis. After that, we present the abstract higher-order fixpoint algebra and its semantics, which were formalized in this thesis.

2.1 Orders, Lattices and Fixpoints

Partial Order A *partial order* is a binary relation which is reflexive, antisymmetric, and transitive.

Ordered Sets A *partially ordered set*, also called poset, $(M, \leq)$ is a set M equipped with a partial order $\leq$. The *least element* of the poset, if it exists, is denoted by $\bot$ (called bottom), such that $\forall x \in M : \bot \leq x$. Similarly, the *greatest element* is denoted by $\top$ (called top), such that $\forall x \in M : x \leq \top$. Let $S \subseteq M$. An *upper bound* of S is an element $d \in M$ such that $\forall x \in S : x \leq d$. The *least upper bound* of S is an upper bound $d \in M$ such that for every upper bound $d' \in M : d \leq d'$. When it exists, it is unique and denoted by $\bigsqcup S$. Dually, a *lower bound* of S is an element $d \in M$ such that, $\forall x \in S : d \leq x$. The *greatest lower bound* of S is a lower bound $d \in M$ such that for every lower bound $d' \in M : d' \leq d$. When it exists, it is unique and denoted by $\bigsqcap S$.

Lattices A *lattice* $(M, \leq)$ is a poset where each pair of elements $x, y \in M$ has a least upper bound and greatest lower bound. Any totally ordered set is a lattice. A *complete lattice* is a lattice where any subset $S \subseteq M$ has a least upper bound $\bigsqcup S$ and a greatest lower bound $\bigsqcap S$. A complete lattice has both a least element and a greatest element.

Monotonic Functions Let $(M_i, \leq_i), i \in \{1, 2\}$ be posets. A function $f : M_1 \to M_2$ is *monotonic* if $\forall x, y \in M_1, x \leq_1 y \longrightarrow f(x) \leq_2 f(y)$.

Fixpoints Given a poset $(M, \leq)$ and a function $f : M \to M$, a *fixpoint* of f is an element $x \in M$ such that $f(x) = x$. The *least fixpoint* of f (if it exists), written $\mathrm{lfp}\, f$, is a fixpoint of f such that for every fixpoint $x \in M$ of f, $\mathrm{lfp}\, f \leq x$. Dually, we define the *greatest fixpoint* of f, denoted by $\mathrm{gfp}\, f$.

We now recall a fundamental theorem due to Knaster [11] and Tarski [12]:

Theorem 2.1.1 (Knaster-Tarski Fixpoint Theorem) *The set of fixpoints of a monotonic function $f : M \to M$ over a complete lattice is also a complete lattice.*

The theorem guarantees that f has a least fixpoint $\text{lfp}\, f = \bigsqcap\{x \in M \mid f(x) \leq x\}$ and greatest fixpoint $\text{gfp}\, f = \bigsqcup\{x \in M \mid x \leq f(x)\}$.

These fixpoint characterizations are not constructive, hence we introduce a constructive characterization due to Kleene [13]:

Theorem 2.1.2 (Kleene Fixpoint Theorem) *Let $(M, \leq)$ be a finite lattice and let $f : M \to M$ be a monotonic function. Then f has a least fixpoint which is the least upper bound of the increasing chain*

$$\bot \leq f(\bot) \leq f(f(\bot)) \leq \ldots \leq f^n(\bot) \leq \ldots$$

i.e., $\text{lfp}\, f = \bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$.

2.2 Abstract Higher-Order Fixpoint Algebra

In [10], Bruse et al. define an abstract higher-order fixpoint algebra $\mu\mathsf{HO}$ which allows to define terms in a lattice to express fixpoint evaluation problems. In this thesis, we consider a fragment $\mu\mathsf{HO}_1$ of $\mu\mathsf{HO}$ with the following restrictions to reduce the complexity of the formalization:

- Functions are restricted to be of first-order only. This restriction is necessary to be able to formalize this algebra in Isabelle/HOL, because Isabelle/HOL does not have *dependent types*, but only *simple types*. Hence, to the best of this author's knowledge, there would be no conventional way of modeling the semantics of $\mu\mathsf{HO}_1$ in Isabelle/HOL otherwise. An explanation of this is provided in Section 4.2.
- There is no greatest fixpoint operator because most proofs for this case are carried out in a similar fashion to its dual operator.
- In $\mu\mathsf{HO}_1$, all functions are monotonic while $\mu\mathsf{HO}$ allows non-monotonic functions. When non-monotonic functions are involved one has to be careful when it comes to fixpoint operators mixed with non-monotonic functions because the semantics of fixpoint operators is not well-defined in this case. To ensure well-defined semantics, $\mu\mathsf{HO}$ comes with a type system, that is not considered in this formalization. The type system also ensures that functions are always applied to the right amount of arguments. In case of $\mu\mathsf{HO}_1$, we restrict ourselves to functions of arity 1 only.

It should be noted that the omission of both greatest fixpoint operator and non-monotonic functions is technically a weakening of the expressiveness of the algebra but the formalization could be extended with every restriction above, except adding terms of arbitrary order because of the aforementioned reason.

Next we define the syntax and semantics of $\mu\mathsf{HO}_1$.

Syntax. Let $\mathsf{Func}_0, \mathsf{Func}_1$ be a set of zero-order, resp. first-order monotonic functions and $\mathsf{Var}_0, \mathsf{Var}_1$ be a set of zero-order, resp. first-order variables. *Terms* of $\mu\mathsf{HO}_1$ are generated by the following grammars:

$$\phi ::= f \mid x \mid \psi\phi \mid \mu x.\, \phi$$
$$\psi ::= g \mid X \mid \lambda x.\, \phi \mid \mu X.\, \psi$$

where $x \in \mathsf{Var}_0, X \in \mathsf{Var}_1, f \in \mathsf{Func}_0$ and $g \in \mathsf{Func}_1$. A term of the form $\psi\phi$ is called *application* and feeds the right subterm to the left, which is a function. A term of the form $\lambda x.\, \phi$ is called *lambda abstraction*, and stands for an anonymous function that consumes an argument x and returns ϕ. If a variable occurs under λ or μ it is called a *bound* variable, otherwise it is called *free*.

Semantics. Let $\mathcal{M}$ be a complete lattice and suppose that all abstract functions in $\mathsf{Func}_0 = \{f, \dots\}$ and $\mathsf{Func}_1 = \{g, \dots\}$ have a monotonic interpretation $f^{\mathcal{M}}$, resp. $g^{\mathcal{M}}$. Let $\eta : \mathsf{Var}_0 \cup \mathsf{Var}_1 \to \mathcal{M} \cup \{\text{mono } f \mid f : \mathcal{M} \to \mathcal{M}\}$ be a variable interpretation that assigns values in the lattice, resp. functions from lattice element to lattice element to free variables. An *update* of η is denoted by $\eta[x \mapsto d]$, which maps x to d and everything else is mapped as originally given by η.

The semantics of $\mu\mathsf{HO}_1$ assigns a lattice element to terms derived from ϕ and a monotonic first-order function to terms derived from ψ. The semantics is defined inductively as follows:

$$\begin{aligned}
[\![x]\!]^{\mathcal{M}}_{\eta} &:= \eta(x) & [\![f]\!]^{\mathcal{M}}_{\eta} &:= f^{\mathcal{M}} \\
[\![X]\!]^{\mathcal{M}}_{\eta} &:= \eta(X) & [\![g]\!]^{\mathcal{M}}_{\eta} &:= g^{\mathcal{M}} \\
[\![\psi\phi]\!]^{\mathcal{M}}_{\eta} &:= [\![\psi]\!]^{\mathcal{M}}_{\eta}([\![\phi]\!]^{\mathcal{M}}_{\eta}) & [\![\lambda x.\, \phi]\!]^{\mathcal{M}}_{\eta} &:= d \mapsto [\![\phi]\!]^{\mathcal{M}}_{\eta[x \mapsto d]} \\
[\![\mu x.\, \phi]\!]^{\mathcal{M}}_{\eta} &:= \bigsqcap \{d \mid [\![\phi]\!]^{\mathcal{M}}_{\eta[x \mapsto d]} \sqsubseteq d\} & [\![\mu X.\, \psi]\!]^{\mathcal{M}}_{\eta} &:= \bigsqcap \{\text{mono } d \mid [\![\psi]\!]^{\mathcal{M}}_{\eta[X \mapsto d]} \sqsubseteq d\}
\end{aligned}$$

The right-hand side of the λ-abstraction denotes the function that maps d to the value $[\![\phi]\!]^{\mathcal{M}}_{\eta[x \mapsto d]}$. In case of the fixpoint operators, the values on the right-hand side are well-defined according to the Knaster-Tarksi Fixpoint Theorem (Theorem 2.1.1). Note that all first-order functions are monotonic and therefore all terms are monotonic in their free variables, hence the semantics are well-defined.

In addition to the semantics above, we can define the finite approximation of the least fixpoint $\mu x.\, \phi$ given by the Kleene Fixpoint Theorem (Theorem 2.1.2) as

$$F^0_x := \bot_{\mathcal{M}} \qquad F^{i+1}_x := [\![\phi]\!]^{\mathcal{M}}_{\eta[x \mapsto F^i_x]}$$

and for $\mu X.\, \psi$ as

$$F^0_X := \_ \mapsto \bot_{\mathcal{M}} \qquad F^{i+1}_X := [\![\psi]\!]^{\mathcal{M}}_{\eta[X \mapsto F^i_X]}$$

where $\bot_{\mathcal{M}}$ denotes the bottom element of the lattice $\mathcal{M}$ and $\_ \mapsto \bot_{\mathcal{M}}$ denotes the function that maps every argument to the bottom element.

Local Model-Checking Algorithm. The *evaluation problem* for $\mu\mathsf{HO}_1$ is: given a term ϕ with function symbols in Func_0 and Func_1 interpreted over a finite lattice $\mathcal{M}$, compute the value $[\![\phi]\!]^{\mathcal{M}}$.

A naive model-checking algorithm directly derived from the algorithmic semantics computes the value of subterms in a bottom-up manner, using Kleene fixpoint iteration, storing function values as tables. However, this algorithm is potentially inefficient,

because it computes function tables for *all* values. This is not needed and instead functions can be computed in a *demand-driven* fashion building a partial function table where only the values of needed arguments are computed. Due to fixpoint operators, the value of a function might be defined recursively, thus may depend on the value of the same function on a different argument. In these cases, the domain of the function will be updated and the argument included in further iterations.

Figure 2.1 presents a *local* model-checking algorithm derived from [10] for the first-order fragment $\mu\mathsf{HO}_1$. Here, the algorithm is presented as two mutually recursive functions. The algorithm computes the semantics of a term recursively following the semantics of $\mu\mathsf{HO}_1$ in applicative order, meaning first evaluate the operands and then apply the operator. In case of a fixpoint operator, the algorithm only computes needed values, which means it localizes the fixpoint iteration. The algorithm starts with a function that maps its argument to $\bot_{\mathcal{M}}$, as the initial value of the iteration. The global variables $\textsc{env}_0$ and $\textsc{env}_1$ are used to keep track of the environment and are updated in the repeat loops.

$\triangleright$ global lattice $\mathcal{M}$ with interpretations $f^{\mathcal{M}}$ f.a. f and $g^{\mathcal{M}}$ f.a. g
$\triangleright$ global (partial) $\textsc{env}_0 : \mathsf{Var}_0 \to \mathcal{M}$, $\textsc{env}_1 : \mathsf{Var}_1 \to \mathcal{M} \to \mathcal{M}$

procedure $\textsc{Eval}_0(\phi)$:
  switch ϕ:
    case f: **return** $f^{\mathcal{M}}$
    case x: **return** $\textsc{env}_0(x)$
    case $\psi\phi$: **return** $\textsc{Eval}_1(\psi, \textsc{Eval}_0(\phi))$
    case $\mu x.\, \phi$: $\textsc{env}_0(x) := \bot_{\mathcal{M}}$
      repeat:
        $f := \textsc{env}_0(x)$
        $\textsc{env}_0(x) := \textsc{Eval}_0(\phi)$
      until $f = \textsc{env}_0(x)$
      return $\textsc{env}_0(x)$

procedure $\textsc{Eval}_1(\psi, \mathrm{arg})$:
  switch ψ:
    case g: **return** $g^{\mathcal{M}}\,\mathrm{arg}$
    case X: **if** $\textsc{env}_1(X)(\mathrm{arg}) = \texttt{undef}$:
        $\textsc{env}_1(X) := \textsc{env}_1(X)[\mathrm{arg} \mapsto \bot_{\mathcal{M}}]$
      return $\textsc{env}_1(X)(\mathrm{arg})$
    case $\lambda x.\, \phi$: $\textsc{env}_0(x) := \mathrm{arg}$; **return** $\textsc{Eval}_0(\phi)$
    case $\mu x.\, \psi$: $\textsc{env}_1(x) := \{\mathrm{arg} \mapsto \bot_{\mathcal{M}}\}$
      repeat:
        $f := \textsc{env}_1(x)$
        for all $\mathrm{arg}' \in \mathrm{dom}(\textsc{env}_1(x))$ **do**:
          $\textsc{env}_1(x) := \textsc{env}_1(x)[\mathrm{arg}' \mapsto \textsc{Eval}_1(\psi, \mathrm{arg}')]$
      until $f = \textsc{env}_1(x)$
      return $\textsc{env}_1(x)(\mathrm{arg})$

Figure 2.1: Local Model-Checking Algorithm for $\mu\mathsf{HO}_1$.

Chapter 3

Isabelle: A Proof Assistant

Isabelle is a generic *proof assistant* but also a generic *framework* for creating deductive systems. Generic means that Isabelle is not bound to a specific logical formalism. Isabelle's *meta logic* Isabelle/Pure, an intuitionistic fragment of higher-order logic, allows the formalization of *object logics* by introducing their characteristics as axioms [14]. Several different object logics have been developed inside Isabelle/Pure, notably Isabelle/ZF (based on Zermelo-Fraenkel set theory) and Isabelle/FOL (based on first-order logic [15] and Isabelle/HOL (based on higher-order logic with simple types). Isabelle/HOL is the most important and most used instance of Isabelle.

Isabelle/HOL provides mechanisms for specifying (co)datatypes, (co)inductive definitions and recursive functions with pattern matching and a large theory library. Isabelle/Isar is used to write *structured proofs* and specifications. Isabelle's code generator facilities allow to generate SML, Ocaml, Haskell or Scala code from Isabelle files [16]. Recently, support for generating LLVM bytecode has been added [17]. Isabelle also includes extensive typesetting support [18]. New notations using mathematical symbols can be introduced, LaTeX can be directly embedded into Isabelle files, and Isabelle files can be converted into typeset documents[1].

Isabelle comes with an official interface based on the text editor jEdit[2]. The interface provides continuous proof checking of all visible proofs with instant feedback in real-time. As seen in Figure 3.1, the interface is responsible for syntax-highlighting and rendering of special symbols. At the bottom is a window showing information about the current proof (step), showcasing what remains to be proven. When a proof did not succeed, the failing part is highlighted in red.

3.1 Isabelle's Architecture

Isabelle follows the *LCF approach* [19] which consists of using a strong statically-typed programming language together with an abstract data type called `thm` for theorems where the only values of that data type are the axioms of the logical calculus, and the only functions over that data type are the inference rules of the logical calculus. The only terms of type `thm` are *derivable sequents* of the form $\Gamma \vdash \varphi$. The programming language ensures that there is no other way to construct a term of type `thm` than via inference

[1] This text was partially written in and compiled with Isabelle.
[2] http://www.jedit.org

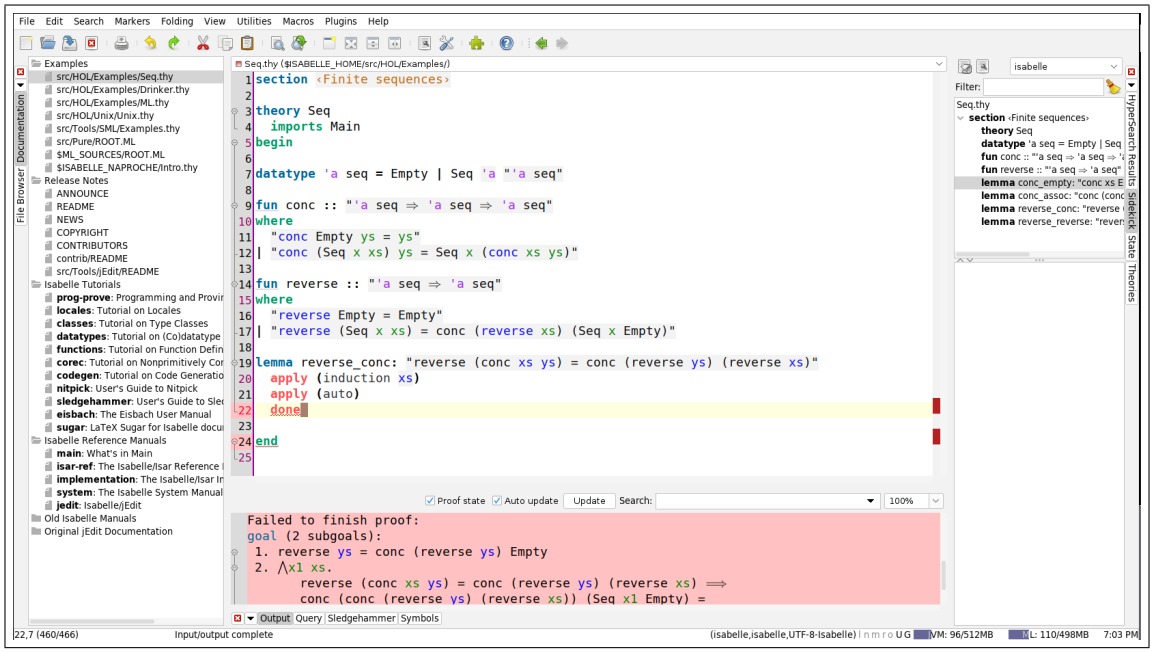

Figure 3.1: Isabelle/jEdit: Isabelle was not able to verify the correctness of a proof.

rules. For example, one of the functions over that type would be `assume : formula → thm`, with `assume` φ representing the derivable sequent $\varphi \vdash \varphi$. The LCF approach gives *soundness by construction* relative to the correctness of the proof kernel implementation, which is kept as small as possible[3]. If the goal is to prove φ and only functions on type `thm` are applied, there will never be a faulty proof. Another advantage is that the user is able to define new proof automation methods by writing more sophisticated functions on type `thm`. This enhances the power of the automation but does not lead to inconsistencies, because everything eventually goes back to the primitive inference rules of the proof kernel. This is comparable to an operating system kernel processing a system call. This approach has been very successful and has been employed in other proof assistants [20]. It was invented to remove the need of storing the proof of theorems in memory, because with the LCF approach only the result of a theorem is stored and not its proof.

3.2 Isabelle/HOL

Isabelle/HOL is an instantiation of *higher-order logic* [21] extended with polymorphism and type classes. It can be seen as a functional programming language.

Types. Base types are types like `nat` for natural numbers or `bool` for boolean values. Type constructors are polymorphic types like `'a list`, `'a × 'b`, or `'a set`. Type variables like `'a` are placeholders that can be instantiated with concrete types and always start with a prime. Function types are written with a double-dashed arrow `'a ⇒ 'b`. Functions are *curried* by default, i.e. addition on natural numbers has the type `nat ⇒ nat ⇒ nat`, as opposed to the type `nat × nat ⇒ nat`. This means that addition is a function that takes an argument of type `nat` and returns a function of type `nat ⇒ nat`, which takes an

[3]The Isabelle kernel consists of rougly 2000 lines of StandardML.

argument of type `nat` again and returns a natural number. Note that ⇒ associates to the right.

Type Classes. Types can be organized into *type classes*. Type classes are used to define properties that a type belonging to that class must fulfill. We use this in this thesis to restrict ourselves to types that implement finite lattices.

Terms. Terms are either variables, constants, function applications, or λ-abstractions:

- Variables represent an arbitrary value of a type. In addition to free and bound variables, there are *schematic variables*, e.g. `?x` which can be instantiated with arbitrary terms. When stating a theorem and proving it, variables are usually fixed. Whenever a proof of a theorem is finished, variables are treated as schematics such that the theorem can be instantiated with arbitrary terms.
- Constants represent a specific value of a type. Since functions are *first-class* objects in Isabelle, variables and constants can also represent functions.
- Function application does not use parentheses or commas for separating the arguments, i.e. `f x y` describes the application of a function `f` to its arguments `x` and `y`. Function application associates to the left.
- A λ-abstraction builds an anonymous function, e.g. λ`x. x` describes the identity function, so the term λ`x. x` has type `'a ⇒ 'a`.

Type constraints are denoted by the operator `::`, i.e. `x :: nat` is a term of type `nat`. If there are no type annotations for variables or functions, Isabelle will infer the type automatically using *type inference*.

Logical Connectives. Since Isabelle distinguishes between *meta logic* and *object logic*, there are two different versions for commonly found logical operators and quantifiers in Isabelle. Its meta logic uses non-standard syntax to leave the usual mathematical syntax for the object logics, such as HOL. In the meta logic, the universal quantifier is ⋀, implication is ⟹, and equality is ≡. These operators and quantifiers operate on the meta level truth values. In addition to that, HOL introduces the type `bool` with values `True` and `False` and a collection of operators and quantifiers usually found in mathematics. These include ¬ (negation), ∧ (conjunction), ∨ (disjunction), ⟶ (implication), and `=` (equality). Quantifiers are written as ∀`x.` (universal quantification) and ∃`x.` (existential quantification) followed by the term that is quantified over. The difference between the meta and object level operators and quantifiers is mostly technical and for this thesis they can be assumed to be equal.

Inductive Datatypes. Datatypes are defined using the command ***datatype***. It creates an *algebraic datatype* commonly used in functional programming. For example a datatype for defining binary trees could be

```
datatype 'a bintree = Leaf | Branch "'a bintree" 'a "'a bintree"
```

The name of the declared type is `bintree` and has two constructors `Leaf` for leaves of the tree and `Branch` for internal nodes of the tree that carry a value.

Function Definitions. (Recursive) Functions are defined using the command ***fun***. The following function computes the length of a list:

```
fun length :: "'a list ⇒ nat" where
```

```
"length [] = 0" |
"length (x # xs) = 1 + length xs"
```

A function definition has a name and an (optional) type signature followed by its function body defined using *pattern matching*. This function definition is defined using two cases, either an empty list or a non-empty list that will be traversed recursively.

In HOL all functions are total, therefore termination is a fundamental requirement when defining functions. When using the ***fun*** command, Isabelle tries to prove termination automatically when the definition is made. If the proof fails, the definition is rejected. This can have two causes, either the definition does indeed not terminate, or the default proof procedures were not powerful enough. For these cases, Isabelle offers the ***function*** command, where the necessary proof obligations become visible to the user and can analyzed and solved manually. In addition to this, Isabelle provides a few more proof methods for proving termination of functions automatically. More information on termination proofs can be found in [22].

Locales. Locales can be compared to *parameterized modules*. A locale fixes types, constants and assumptions within a specified context.

For the remainder of this thesis, we will use Isabelle to refer to Isabelle/HOL.

3.3 Proof Methods

Isabelle provides a rich set of automation to prove theorems, called *proof methods*. They differ in which goals they can solve and how they react if they where unsuccessful. Some proof methods will present an error message to the user that it failed and how far it got, while other proof methods will stop and only report a failure or might run forever.

The most used proof methods include:

- `simp`: Invokes the simplifier. Has a predefined pool of simplification rules that can be extended temporarily or permanently, if needed. Operates only on one subgoal. It rewrites terms with provided equations until the theorem is trivial or no further simplification rules can be applied.
- `auto`: Includes `simp` but has additional rules about logical and set-theoretic reasoning. Operates on all subgoals and will report an error message on how far it got, in case of a failed proof.
- `blast`: Includes a powerful rule set but only operates on the first goal and will only report failure, with no additional information in case of a failed proof. It also has no knowledge about simplification.
- `metis`: Mostly used by proofs delivered by ***sledgehammer*** (see Section 3.5) to replay the proofs through the proof kernel in order to verify them.
- `smt`: Is used to ask an *automatic theorem prover* to prove a proof obligation. If a proof is found, then Isabelle replays the found proof through its kernel. This means that the external tool is not *trusted*. If the proof replay fails, the proof is rejected. This can either mean that the proof is faulty or that the time limit was reached and the replay process was canceled. Since it depends on external tools such as Z3[4],

[4] https://z3prover.github.io/

compatibility with future Isabelle versions cannot be guaranteed and previously successful proofs might fail. Nevertheless, quite often it is the only proof method returned by ***sledgehammer***.

3.4 Proving in Isabelle

Statements to be proven are stated using the commands ***lemma*** and ***theorem***, which have no technical difference besides allowing the user to highlight the important statements via ***theorem***. The commands are followed by an optional name and the statement itself, e.g.

```
lemma exists_n_gt_zero: "∃n::nat. n + m > 0"
```

All free variables are implicitly universally quantified, i.e. the lemma is equivalent to

```
lemma exists_n_gt_zero_alt: "⋀m. ∃n::nat. n + m > 0"
```

Theorems with assumptions are written as

```
lemma "(n::nat) + m = m ⟹ n = 0"
```

In addition, Isabelle also offers an alternative syntax for writing theorems with assumptions. The above lemma can be rewritten equivalently as

```
lemma
  fixes n::nat and m
  assumes "n + m = m"
  shows "n = 0"
```

The command ***fixes*** is used to introduce variables. The command ***assumes*** states assumptions and ***shows*** states the conclusion.

After stating a theorem, Isabelle creates a *proof state*, which consists of a collection of statements, called *subgoals*, that must be proved to show that the theorem holds. Using *proof methods*, subgoals can be transformed into zero or more new subgoals. When the proof state consists of zero subgoals after the application of a proof method, the proof is finished and Isabelle has accepted that the theorem holds.

Isabelle uses a proof language called Isabelle/Isar [23] which allows to write proof in a *structured* and *forward* fashion starting from assumptions and ending with the proof goal. It was designed to bring formal proofs closer to ordinary mathematical proofs on paper. Traditionally, proof assistants used a *backwards* style for writing proofs. Starting with the proof goal and applying proof methods (also called tactics in the literature) to split the goal into subgoals until no goals are left. In this style, a proof only consists of a list of used proof methods without stating subsequent subgoals, which makes these proofs incomprehensible without opening the proof assistant and stepping through the proof state explicitly. Proofs written in Isabelle/Isar aim to be readable without being required to open Isabelle and looking at the proof state explicitly.

A proof in Isabelle/Isar follows the structure of the proof goal. In general, a proof goal is of the form $\bigwedge x_1 \ldots x_k.\ [\![A_1 ; \ldots ; A_n]\!] \Longrightarrow C$ where $x_1 \ldots x_k$ are variables, $A_1 ; \ldots ; A_n$ a list of assumptions, and C being the goal[5]. An Isabelle/Isar proof for this goal is given by

[5]The notation $[\![A_1 ; \ldots ; A_n]\!] \Longrightarrow C$ is a syntactic shorthand for $A_1 \Longrightarrow A_2 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C$.

proof -
  fix x_1 ... x_k
  assume "A_1"
  ⋮
  assume "A_n"
  have l_1: "P_1" ***by*** simp
  ⋮
  have l_n: "P_n" ***by*** blast
  show "C" ***by*** ...
qed

where P_1, ..., P_n are intermediate facts with optional labels l_1, ..., l_n to reference the facts in later steps. Isabelle/Isar proofs are enclosed in a ***proof***-***qed***-block. The ***proof*** command can be optionally followed by an initial proof method to transform the proof goal before entering the proof block. Otherwise, the minus symbol (-) indicates no initial proof method and the proof is started without transforming the goal. Variables and assumptions are introduced by the commands ***fix*** and ***assume***. Only assumptions present in the proof state can be assumed, otherwise it is impossible to prove the goal. Intermediate facts can be introduced using the command ***have***. To complete a goal, the command ***show*** is used. The commands ***have*** and ***show*** need to be followed by a proof method or a nested ***proof***-***qed***-block as a justification that these hold. Facts can be chained together using the ***then*** command to indicate that the proceeding fact follows from the previous fact.

3.4.1 An Example Proof

Figure 3.2 shows a proof that the length of the concatenation of two lists is equal to addition of the length of both lists. Here, the function @ is a syntactic sugar for the function append :: 'a list ⇒ 'a list ⇒ 'a list which takes two lists and concatenates them The proof is started with the initial proof method ***induction*** which indicates a proof by structural induction on the list xs resulting in the proof state:

```
1. length ([] @ ys) = length [] + length ys
2. ⋀a xs.
      length (xs @ ys) = length xs + length ys ⟹
      length ((a # xs) @ ys) = length (a # xs) + length ys
```

The numbered lines indicate the subgoals. The first subgoal is the *base case*, the second one the *induction step*. The ... are a shorthand for the term on the right-hand side of an equality in the previous fact. In the induction step the induction hypothesis is used via the ***using*** command.

3.5 Sledgehammer

Isabelle offers a unique feature called ***sledgehammer***, which calls a number of external *automatic theorem provers* that will run for up to 30 seconds, searching for a proof. If successful, a proof method will be generated which can be inserted into the proof, so the external provers do not need to be trusted. One of the advantages of using

```
lemma "length (xs @ ys) = length xs + length ys"
proof (induction xs)
  case Nil
  have "length ([] @ ys) = length ys" by simp
  also have "... = 0 + length ys" by simp
  also have "... = length [] + length ys" by simp
  finally show ?case by simp
next
  case (Cons x xs)
  assume IH: "length (xs @ ys) = length xs + length ys"
  have "length ((x # xs) @ ys) = length (x # (xs @ ys))"
    by simp
  also have "... = Suc (length (xs @ ys))" by simp
  also have "... = Suc (length xs + length ys)"
    using IH by simp
  also have "... = Suc (length xs) + length ys" by simp
  also have "... = length (x # xs) + length ys" by simp
  finally show ?case by simp
qed
```

Figure 3.2: An example proof in Isabelle.

sledgehammer is that it will take all available lemmas (selected heuristically) into account when searching for a proof. However, there is no guarantee that ***sledgehammer*** will find a proof if one exists. More information on ***sledgehammer*** and its inner workings can be found in [24].

Chapter 4

Formalization of Different Semantics

This chapter presents the formalization of the abstract higher-order fixpoint algebra and its semantics carried out in Isabelle and provides an overview of the important proofs and lemmas. It shows how the algorithm presented in Figure 2.1 can be modeled in Isabelle and which complications might arise when attempting to prove the correctness of the algorithm in Isabelle.

This thesis was formalized using Isabelle 2022[1]. The formalization can be found on GitHub[2].

4.1 Syntax

We start the formalization with the definition of a *locale* `ctx` (short for context) used to create a context in which the formalization and all proofs will be carried out[3]. This context consists of a fixed set of elements that implement `finite_lattice`, an interpretation function for zero-order abstract function symbols, an interpretation function for first-order abstract function symbols, and the requirement that the interpretation is monotonic.

```
locale ctx =
  fixes elems :: "'a :: finite_lattice"
    and abs_func0 :: "name ⇒ 'a"
    and abs_func1 :: "name ⇒ ('a ⇒ 'a)"
  assumes abs_func1_mono: "x ≤ y ⟶ abs_func1 f x ≤ abs_func1 f y"
begin
```

Terms of $\mu\mathsf{HO}_1$ are represented by two mutually recursive datatypes where `tm0` describes zero-order terms and `tm1` describes first-order terms.

```
datatype
  tm0 = Var0 name | Func0 name | App tm1 tm0 | Mu0 name tm0
and
  tm1 = Var1 name | Func1 name | Lam name tm0 | Mu1 name tm1
```

[1] https://isabelle.in.tum.de/

[2] https://github.com/waynee95/muHO-isabelle

[3] The type `name` is a synonym for `nat` and describes variables denoted by natural numbers instead of strings. Natural numbers are easier to handle and explicit variables are not needed in the formalization.

4.2 Semantics

4.2.1 Denotational Semantics

The denotational semantics of $\mu\mathsf{HO}_1$ is modeled as two mutually recursive functions, each taking variable interpretations and a term as arguments. The semantics of zero-order terms is an element in the lattice denoted by the return type `'a` of `sem_tm0` and the semantics of first-order terms is a function from lattice element to lattice element denoted by `'a ⇒ 'a` of `sem_tm1`. Because `'a` is fixed in this context and was constrained to implement `finite_lattice`, there is no explicit type constraint needed in this function definition for `'a`. The return type of the semantics of a term depends on the order of the term, therefore a separate function for each order is needed. Having only one function with a general return type for terms of arbitrary order is impossible, as this would require *dependent types*, which Isabelle does not support. More information about dependent types may be found in [25].

The syntax `η0(X := d)` is a shorthand for updating the variable interpretation. In the following, the two functions `sem_tm0` and `sem_tm1` will be abbreviated by the name `sem`.

```
fun
  sem_tm0 :: "(name ⇒ 'a) ⇒ (name ⇒ ('a ⇒ 'a)) ⇒ tm0 ⇒ 'a"
and
  sem_tm1 :: "(name ⇒ 'a) ⇒ (name ⇒ ('a ⇒ 'a)) ⇒ tm1 ⇒ ('a ⇒ 'a)"
where
"sem_tm0 η0 η1 (Var0 x) = (η0 x)" |
"sem_tm0 η0 η1 (Func0 f) = (abs_func0  f)" |
"sem_tm0 η0 η1 (App φ ψ) = (sem_tm1 η0 η1 φ) (sem_tm0 η0 η1 ψ)" |
"sem_tm0 η0 η1 (Mu0 X φ) = ⋂{d. sem_tm0 (η0(X := d)) η1 φ ≤ d}" |

"sem_tm1 η0 η1 (Var1 x) = (η1 x)" |
"sem_tm1 η0 η1 (Func1 f) = (abs_func1 f)" |
"sem_tm1 η0 η1 (Lam x φ) = (λd. sem_tm0 (η0(x := d)) η1 φ)" |
"sem_tm1 η0 η1 (Mu1 X φ) =
    ⋂{d. mono d ∧ sem_tm1 η0 (η1(X := d)) φ ≤ d}"
```

In the `Mu1` case, we use the predicate `mono` provided by Isabelle to specify that all functions in the set should be monotonic.

An important lemma needed for later proofs is the monotonicity of the semantics function `sem` under monotonic variable interpretations. This lemma can be formulated in Isabelle as follows:

```
lemma sem_mono:
"⋀η0 η1 η0' η1'.
  ⟦ ∀z. η0 z ≤ η0' z;
    ∀f z. (η1 f) z ≤ (η1' f) z;
    ∀f. mono (η1 f);
    ∀f. mono (η1' f) ⟧
  ⟹ sem_tm0 η0 η1 φ ≤ sem_tm0 η0' η1' φ"

"⋀η0 η1 η0' η1'.
  ⟦ ∀z. η0 z ≤ η0' z;
    ∀f z. (η1 f) z ≤ (η1' f) z;
    ∀f. mono (η1 f);
```

```
  ∀f. mono (η1' f) ⟧
⟹ sem_tm1 η0 η1 φ' ≤ sem_tm1 η0' η1' φ'
    ∧ mono (sem_tm1 η0 η1 φ') ∧ mono (sem_tm1 η0' η1' φ')"
```

The lemma statement consists of two statements, one for each function of `sem` because it is defined as two mututally recursive functions. As these are two separate proof goals, both need the assumptions that variable interpretations only provide monotonic functions themselves. The proof is done by simultaneous induction over φ and φ'. We will not show the full proof in detail but highlight two cases:

```
case (App ψ φ)
have 0: "sem_tm0 η0 η1 φ ≤ sem_tm0 η0' η1' φ"
  using App.IH(2) App.prems by simp

have 1: "sem_tm1 η0 η1 ψ ≤ sem_tm1 η0' η1' ψ"
  using App.IH(1) App.prems by simp

from 0 have "(sem_tm1 η0 η1 ψ) (sem_tm0 η0 η1 φ)
               ≤ (sem_tm1 η0 η1 ψ) (sem_tm0 η0' η1' φ)"
  using App.IH(1) App.prems(3) monoD by blast
with 1 have "(sem_tm1 η0 η1 ψ) (sem_tm0 η0 η1 φ)
               ≤ (sem_tm1 η0' η1' ψ) (sem_tm0 η0' η1' φ)"
  by (metis dual_order.trans le_fun_def)
then show ?case by simp
```

In order to prove the `App` case, we first introduce the facts that `sem` is monotonic in φ, resp. ψ, by using the induction hypothesis. Both facts get a label to refer to them in later proof steps. We then use the fact `0` to conclude that ψ is monotonic in its operands. From this, together with the fact that ψ is monotonic, we can conclude that the application $\psi\phi$ is monotonic. In the last step, this fact is combined with the function definition of `sem` for the `App` case to finish the proof. The lemmas `monoD`, `dual_order.trans`, and `le_fun_def` were found by using ***sledgehammer*** and describe facts about monotonicity over orders and functions.

A standard workflow when proving is to first use ***sledgehammer*** on the current subgoal. This is done for two reasons. Firstly, to see if Isabelle can finish the proof goal automatically without further user input and secondly, if ***sledgehammer*** manages to find a proof, it will provide a list of lemmas that might be used to finish the subgoal together with a proof method. Isabelle's standard library is extensive and split across many different files, so it is not always easy to locate useful lemmas for a specific proof by hand.

If ***sledgehammer*** is used on the `Lam` case, it will generate:

```
by (smt dual_order.eq_iff fun_upd_apply le_funI monoI sem_tm1.simps(3))
```

The automation had to rely on the powerful proof method `smt` together with additional lemmas in order to prove this subgoal automatically. There may be several reasons not to insert this proof method. One reason is that this proof method now hides many steps and it is not immediately clear how this subgoal was proven. Instead, this case could be split into intermediate steps, resulting in the following proof for the `Lam` case:

```
case (Lam x φ)
```

```
  then have 1: "mono (sem_tm1 η0 η1 (Lam x φ))
                  ∧ mono (sem_tm1 η0' η1' (Lam x φ))"
    by (simp add: mono_def)

  from Lam have 2: "(λd'. sem_tm0 (η0(x := d')) η1 φ) d
      ≤ (λd'. sem_tm0 (η0'(x := d')) η1' φ) d" for d
    by simp

  from 1 2 show ?case by (simp add: le_funI)
```

Instead of relying on `smt`, this proof only uses the simple proof method `simp` in each step. Two additional lemmas had to be added to the `simp` method, `mono_def` unfolds the definition of the `mono` predicate and `le_funI` describes when a function is less than or equal to another function, which is needed here because the semantics of a λ-abstraction is a function. The command ***for*** is used to introduce an arbitrary but fixed variable scoped to the preceding statement.

4.2.2 Algorithmic Semantics

Next we define a different version of the semantics of $\mu\mathsf{HO}_1$ based on approximation of fixpoints via fixpoint iteration. To improve readability, we only show the parts that are different compared to the previous definition. We introduce two new functions that approximate the respective fixpoint operator `approx_tm0` and `approx_tm1`, as per Kleene's Fixpoint Theorem (see Theorem 2.1.2). In the following, we will use the same name abbreviation as with `sem`. The whole function now consisting of four mutually recursive functions is referred to as `sem_k` and we use `approx` to refer to the two auxiliary functions `approx_tm0` and `approx_tm1`.

```
"approx_tm0 η0 η1 φ x init 0 = init" |
"approx_tm0 η0 η1 φ x init (Suc n) =
    sem_tm0_k (η0(x := approx_tm0 η0 η1 φ x init n)) η1 φ" |

"approx_tm1 η0 η1 φ x init 0 = init" |
"approx_tm1 η0 η1 φ x init (Suc n) =
    sem_tm1_k η0 (η1(x := approx_tm1 η0 η1 φ x init n)) φ" |

"sem_tm0_k η0 η1 (Mu0 X φ) =
    approx_tm0 η0 η1 φ X ⊥ (card (UNIV::'a set))" |

"sem_tm1_k η0 η1 (Mu1 X φ) =
    approx_tm1 η0 η1 φ X (λ_. ⊥) (card (UNIV::('a⇒'a) set))"

termination
  by size_change
```

The built-in constant `UNIV` is used to refer to the *universal set* of all elements of a certain type, in this case, the underlying set of the lattice in question, resp. the appropriate function set. This constant has to be constrained with a type annotation in order to refer to the correct type. The built-in function `card` describes the cardinality of a set.

The function declaration is followed by the ***termination*** command because it was defined using the ***function*** command. In this case Isabelle's automatic termination

checker was not able to conclude that this function terminates, so a proof has to be provided explicitly. This is done via the `size_change` proof method. This method is usually required in case of several mutually recursive functions with multiple arguments [22].

Similar to `sem`, we prove that `sem_k` is monotonic w.r.t. monotonic variable interpretations. Before being able to prove the monotonicity of `sem_k`, we have to prove the monotonicity of `approx`, because in cases of fixpoint operators the `approx` function is involved. We will showcase this for `approx_tm0`:

```
lemma approx_tm0_mono:
  "⟦⋀η0 η1 η0' η1'.

    ⟦ ∀z. η0 z ≤ η0' z;
      ∀f z. η1 f z ≤ η1' f z;
      ∀f. mono (η1 f);
      ∀f. mono (η1' f) ⟧
    ⟹ sem_tm0_k η0 η1 φ ≤ sem_tm0_k η0' η1' φ;

     ∀z. η0 z ≤ η0' z;
     ∀f z. (η1 f) z ≤ (η1' f) z;
     ∀f. mono (η1 f);
     ∀f. mono (η1' f) ⟧
   ⟹ approx_tm0 η0 η1 φ x ⊥ n ≤ approx_tm0 η0' η1' φ x ⊥ n"
  by (induction n) auto
```

In addition to the assumptions that the variable environments are monotonic, the assumption is required that `sem_tm0_k` is monotonic. Then the monotonicity of `approx_tm0` follows by induction on n. Since both cases in the induction can be solved with `auto`, it is not necessary to write out both cases explicitly and the shorthand ***by*** `(induction n) auto` can be used.

With the auxiliary lemmas that `approx` is monotonic, we can prove that `sem_k` is monotonic w.r.t. to monotonic variable interpretations:

```
theorem sem_k_mono:
  "⋀η0 η1 η0' η1'.
  ⟦ ∀z. η0 z ≤ η0' z;
    ∀f z. (η1 f) z ≤ (η1' f) z;
    ∀f. mono (η1 f);
    ∀f. mono (η1' f) ⟧
  ⟹ sem_tm0_k η0 η1 φ ≤ sem_tm0_k η0' η1' φ"

  "⋀η0 η1 η0' η1'.
  ⟦ ∀z. η0 z ≤ η0' z;
    ∀f z. (η1 f) z ≤ (η1' f) z;
    ∀f. mono (η1 f);
    ∀f. mono (η1' f) ⟧
  ⟹ sem_tm1_k η0 η1 φ' ≤ sem_tm1_k η0' η1' φ'
      ∧ mono (sem_tm1_k η0 η1 φ') ∧ mono (sem_tm1_k η0' η1' φ')"
```

The proof follows the same pattern as with `sem_mono`, except that the fixpoint operator cases are proved by using ***by*** `simp add: approx_mono`, which means that we can apply the auxiliary lemma here and the case is solved by simplification which instantiates the lemma and solves the proof goal.

4.2.3 Equivalence of Algorithmic and Denotational Semantics

This section covers the proof that the algorithmic semantics based on fixpoint iteration semantics is equivalent to the denotational semantics. On paper this proof involves two steps:

1. Prove that any approximant is less than the denotational fixpoint. This is done by fixpoint unfolding.
2. Prove that the approximation stabilizes and that the result is a fixpoint which is at least as big as the least fixpoint.

We start by proving that function `approx` stabilizes after at most cardinality of the base set many steps, meaning that there exists an n such that the value of `approx` applied to $n+1$ does not change anymore. We will show the proof strategy for `approx_tm0`. For this we need two auxiliary lemmas:

```
lemma approx_tm0_mono_min:
  "∀f. mono (η1 f)
  ⟹ approx_tm0 η0 η1 φ x ⊥ n ≤ approx_tm0 η0 η1 φ x ⊥ (Suc n)" for n
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case
    using sem_k_mono(1) by simp
qed
```

Stating that `approx_tm0` is monotonic in n in one step. From which we can prove, that `approx_tm0` is monotonic in n for any number of steps:

```
lemma approx_tm0_mono_min2:
  "⟦ ∀f. mono (η1 f); m ≥ n ⟧
  ⟹ approx_tm0 η0 η1 φ x ⊥ n ≤ approx_tm0 η0 η1 φ x ⊥ m"
proof (induction m)
  case 0
  then show ?case
    by (simp add: approx_tm0_mono)
next
  case (Suc m)
  then show ?case
    by (metis approx_tm0_mono_min lift_Suc_mono_le)
qed
```

Without an explicit proof for both lemmas, Isabelle is not able to figure out that `approx` is monotonic in n. A fact that is usually obvious on paper and does not get special treatment.

Now we can prove the easier case, that `approx_tm0` stays stable, once it became stable:

```
lemma approx_tm0_stays_stable:
  assumes "approx_tm0 η0 η1 φ x ⊥ n = approx_tm0 η0 η1 φ x ⊥ (Suc n)"
    and "m ≥ n"
    and "∀f. mono (η1 f)"
```

```
lemma inj_func_card:
  assumes "∀i ≤ card S + 1. ∀i' ≤ card S + 1.
           i ≠ i' ⟶ f i ≠ f i'"
  shows "n ≤ card S + 1 ⟶ n ≤ card (⋃i≤n. {f i})"
proof (rule impI, induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  have "∀i≤n. ∀i'≤n. i ≠ i' ⟶ {f i} ≠ {f i'}"
    using Suc.prems assms by auto
  then have "∀i≤n. {f i} ≠ {f (Suc n)}"
    using Suc.prems assms by auto
  then have 0: "¬ ({f (Suc n)} ⊆ (⋃i≤n. {f i}))"
    by auto

  have "(⋃i≤Suc n. {f i}) = (⋃i≤n. {f i}) ∪ {f (Suc n)}"
    by (simp add: atMost_Suc)
  then have "card (⋃i≤Suc n. {f i}) = card ((⋃i≤n. {f i}) ∪ {f (Suc
n)})"
    by auto
  with 0 have 1: "... = card (⋃i≤n. {f i}) + card ({f (Suc n)})"
    by simp

  have "Suc n ≤ card (⋃i≤n. {f i}) + 1"
    using Suc.IH Suc.prems by auto
  also have "... ≤ card ((⋃i≤n. {f i}) ∪ {f (Suc n)})"
    using 1 by auto
  also have "... ≤ card ((⋃i≤Suc n. {f i}))"
    by (simp add: atMost_Suc)
  finally show ?case by simp
qed
```

Figure 4.1: Auxiliary lemma about the fact that the image of an injective function into a set is at least as big as the set.

```
  shows "approx_tm0 η0 η1 φ x ⊥ m = approx_tm0 η0 η1 φ x ⊥ n"
```

The proof is done by splitting it into the case $\leq$ and $\geq$ and applying the previously proven lemma.

Next, we need to prove that `approx_tm0` gets eventually stable, after at most cardinality of the base set many steps:

```
lemma approx_tm0_eventually_stable:
  assumes "∀f. mono (η1 f)"
  shows "∃n ≤ card (UNIV::'a set).
      approx_tm0 η0 η1 φ x ⊥ n = approx_tm0 η0 η1 φ x ⊥ (Suc n)"
```

The proof can be found in Figure 4.2. The proof is done by contradiction denoted by the initial proof method `rule ccontr`, assuming there is no n less than the cardinality of the base set. This proof could make an argument over the length of the chain of all

```
proof (rule ccontr)
  assume 0: "¬ (∃n≤(card (UNIV::'a set)).
      approx_tm0 η0 η1 φ x ⊥ n = approx_tm0 η0 η1 φ x ⊥ (Suc n))"
  then have 1: "∀m≤(card (UNIV::'a set)).
      approx_tm0 η0 η1 φ x ⊥ m < approx_tm0 η0 η1 φ x ⊥ (Suc m)"
    using approx_tm0_mono_min assms order_less_le by blast

  let ?A = "⋃i≤(card (UNIV::'a set) + 1).
              {(approx_tm0 η0 η1 φ x ⊥ i)}"

  have 2: "?A ⊆ (UNIV::'a set)"
    by auto

  from 0 1 have "∀i≤(card (UNIV::'a set) + 1).
                 ∀i'≤(card (UNIV::'a set) + 1).
    i' ≠ i ⟶ approx_tm0 η0 η1 φ x ⊥ i ≠ approx_tm0 η0 η1 φ x ⊥ i'"
    using approx_tm0_mono_min approx_tm0_mono_min2 assms
    by (metis Suc_eq_plus1 antisym not_less_eq_eq)

  then have "∀n≤(card (UNIV::'a set) + 1).
            n ≤ card (⋃i≤n. {(approx_tm0 η0 η1 φ x ⊥ i)})"
    using inj_func_card by metis
  then have 3: "card ?A > (card (UNIV::'a set))"
    by auto

  from 2 3 show False
    by (meson card_mono finite_UNIV leD)
qed
```

Figure 4.2: Proof that `approx_tm0` becomes eventually stable.

approximants, but this would require information and lemmas about chains provided by Isabelle. Instead, the proof uses an argument about the cardinality of the set of approximants. In the proof we define a variable `?A` to be the set of all approximants. This set is clearly a subset of the set of all elements in the lattice. From the fact that, there is no n on which `approx_tm0` stabilizes, we can conclude that all approximants are pairwise disjunct, which means that the cardinality must be greater than the cardinality of the base set. However, the set of approximants `?A` is a subset of the base set, which leads to a contradiction.

This proof needed an auxiliary lemma `inj_func_card` which can be found in Figure 4.1.

With that, we can finally prove that both semantics are equivalent:

```
theorem sem_k_eq_sem:
"⋀η0 η1. ∀f. mono (η1 f) ⟹ sem_tm0_k η0 η1 φ = sem_tm0 η0 η1 φ"

"⋀η0 η1. ∀f. mono (η1 f) ⟹ sem_tm1_k η0 η1 φ' = sem_tm1 η0 η1 φ'"
```

This proof is done by simultaneous induction over both terms. A proof for the `Mu0` case can be found in Figure 4.4. As described in the initial proof strategy, this proof is split into two parts. First show that any approximant is less than the actual fixpoint,

```
lemma unfold_mu0:
  assumes "∀f. mono (η1 f)"
  shows "sem_tm0 η0 η1 (Mu0 x1a φ)
           = sem_tm0 (η0(x1a := sem_tm0 η0 η1 (Mu0 x1a φ))) η1 φ"
    (is "?l = ?r")
proof -
  have 0: "sem_tm0 η0 η1 (Mu0 x1a φ) =
             ⨅{d. sem_tm0 (η0(x1a := d)) η1 φ ≤ d}"
    by simp

  have 1: "sem_tm0 (η0(x1a := d)) η1 φ ≤ d ⟶ ?l ≤ d" for d
    by (simp add: Inf_lower)
  then have 2: "(sem_tm0 (η0(x1a := d)) η1 φ ≤ d)
        ⟶ (η0(x1a := ?l)) x ≤ (η0(x1a := d)) x" for x d
    by simp
  then have "sem_tm0 (η0(x1a := d)) η1 φ ≤ d
        ⟶ ?r ≤ sem_tm0 (η0(x1a := d)) η1 φ" for d
    by (simp add: assms sem_mono(1))
  then have "sem_tm0 (η0(x1a := d)) η1 φ ≤ d ⟶ ?r ≤ d" for d
    using dual_order.trans by blast
  then have 3: "?r ≤ ⨅{d. sem_tm0 (η0(x1a := d)) η1 φ ≤ d}"
    by (simp add: le_Inf_iff)
  with 1 2 have "sem_tm0 η0 η1 (Mu0 x1a φ)
    ≤ sem_tm0 (η0(x1a := sem_tm0 η0 η1 (Mu0 x1a φ))) η1 φ"
    using assms sem_mono(1) by auto
  with 3 show ?thesis by simp
qed
```

Figure 4.3: Proof about fixpoint operator unfolding.

which is proved by induction over n making use of the previously proven monotonicity of `sem` and using the fixpoint unfolding lemma `unfold_mu0` found in Figure 4.3. The other direction follows from the two lemmas about the stabilization of `approx_tm0` which are combined in the single lemma `approx_tm0_mu_final`.

4.3 Local Model-Checking Algorithm

4.3.1 The Algorithm

In this section, we formalize the local model-checking algorithm for $\mu\mathsf{HO}_1$ described in Figure 2.1. There are several things to consider when attempting to write this algorithm in Isabelle:

- The algorithm uses repeat-loops, which are not available in Isabelle, therefore, we need to replace the loops by recursion.
- The algorithm operates over a global variable reflecting the current environment, which gets updated in the repeat-loop. Since Isabelle is a purely functional language, it is impossible to update values in-place. Hence, the updated environments become a return value of the function.
- The algorithm builds partial functions, but this cannot be easily reflected in Isabelle since all functions are total by definition. To solve this, we add an additional return value to the function, which tracks the current domains of the partial functions.

In summary, the return type of the function `eval` is a 3-tuple: `'a` $\times$ `(name` $\Rightarrow$ `'a` $\Rightarrow$ `'a)` $\times$ `(name` $\Rightarrow$ `'a list)`, where the first value describes the value of the term that was evaluated, the second value is the updated environment, and the third value is a map from a function symbol to its current partial domain. Each domain is represented by a list of elements for which this function is currently defined. We use a list instead of a set because then it is easy to iterate over the elements of the domain by using fold. This would not be possible with a set.

We only show the parts that are different for `eval` compared to `sem_k`:

```
"lapprox_tm1 η0 η1 d φ x arg 0 = (η1 x arg, η1, d)" |
"lapprox_tm1 η0 η1 d φ x arg (Suc n) =
    (let (_,η1'',d'') = fold (λarg' ((_::'a), η1', d').
        (let (val''',η1''',d''') = eval_tm1 η0 η1' d' φ arg'
           in (⊥,
               (η1'''(x := (λz. if z = arg' then val''' else η1' x z))),
              d'''))) (d x) (⊥,η1,d)
                  in (if ((η1'' = η1) ∧ (d'' = d))
                      then (η1 x arg, η1, d)
                      else lapprox_tm1 η0 η1'' d'' φ x arg n))" |
"eval_tm1 η0 η1 d (Var1 x) arg =
    (if ListMem arg (d x)
      then (η1 x arg, η1, d)
      else (η1 x arg,
            (η1(x := (λz. if z = arg then ⊥ else (η1 x) z))),
           (d(x := arg # (d x)))))" |
```

```
proof (induction φ and φ')
  case (Mu0 x φ)

  have 1: "approx_tm0 η0 η1 φ x ⊥ n
           ≤ ⨅{d. sem_tm0 (η0(x := d)) η1 φ ≤ d}" for n
  proof (induction n)
    case 0
    then show ?case
      by simp
  next
    case (Suc n)
    have 0: "approx_tm0 η0 η1 φ x ⊥ (Suc n)
           = sem_tm0_k (η0(x := approx_tm0 η0 η1 φ x ⊥ n)) η1 φ"
      by simp
    then have 1: "approx_tm0 η0 η1 φ x ⊥ (Suc n)
           = sem_tm0 (η0(x := approx_tm0 η0 η1 φ x ⊥ n)) η1 φ"
      using Mu0.IH Mu0.prems by fastforce
    then have "approx_tm0 η0 η1 φ x ⊥ n ≤ sem_tm0 η0 η1 (Mu0 x φ)"
      by (simp add: Suc)
    then have 2: "sem_tm0 (η0(x := approx_tm0 η0 η1 φ x ⊥ n)) η1 φ
                ≤ sem_tm0 (η0(x := sem_tm0 η0 η1 (Mu0 x φ))) η1 φ"
      using Mu0.prems sem_mono(1) by auto

   have "sem_tm0 η0 η1 (Mu0 x φ) =
             sem_tm0 (η0(x := sem_tm0 η0 η1 (Mu0 x φ))) η1 φ"
      using Mu0.prems unfold_mu0 by blast
   with 0 1 2 show ?case
     by (metis sem_tm0.simps(4))
  qed

  let ?t = "approx_tm0 η0 η1 φ x ⊥ (card (UNIV::'a set))"

  have "sem_tm0 (η0(x := ?t)) η1 φ ≤ ?t"
    by (metis approx_tm0.simps(2) approx_tm0_mu_final
        dual_order.eq_iff Mu0.IH Mu0.prems)
  then have 2: "?t ≥ ⨅{d. sem_tm0 (η0(x := d)) η1 φ ≤ d}"
    by (simp add: Inf_lower)

  from 1 2 show ?case
    by (simp add: order.antisym)
```

Figure 4.4: Sub-proof for the `Mu0` case for proving the equality of `sem` and `sem_k`.

```
"eval_tm1 η0 η1 d (Mu1 x φ) arg =
    (let (val, η1', d') = lapprox_tm1 η0 (η1(x := λz. ⊥)) (d(x := Nil))
                            φ x arg (card (UNIV::('a⇒'a) set))
         in (val, (η1'(x := λz. ⊥)), (d'(x := []))))"
termination
  by size_change
```

Similar to the function definition of `sem_k`, the definition of `eval` needs an explicit termination proof. It is also necessary to introduce the last parameter of `lapprox_tm1` in order for the termination proof to work, otherwise no automatic proof method is able to conclude that the function indeed terminates. Refactoring out the iteration function used inside the `fold` to improve readability is not possible because the termination proof failed since another mutually recursive function is added.

4.3.2 Verification Strategy

The definition of algorithm `eval` complicates proofs in several ways. Since each function returns a tuple, it is necessary to introduce `let` expressions into the function definition to unpack intermediate values to pass them to the next function call. These `let` expressions have a big impact when doing proofs involving `eval` because to reason about intermediate values, every `let` has to be handled explicitly. Since `eval` includes an if-else, proofs might have to include case distinctions whether an argument belongs to the current domain of a function or not. Additionally, there are mathematical challenges. The algorithm operates on partial functions, so it is necessary to wrap partial functions with a function that maps arguments not in the domain to some value in order to totalize the functions. However, while we get total functions this way, it is not necessarily the case that those functions are monotonic. The fact that `eval` can produce non-monotonic functions has a big impact on a possible proof strategy for proving the correctness of `eval`. For the equality proof of `sem_eq_sem_k`, induction was used along the computation of the approximations. This proof strategy will not work in case of `eval` because to prove equality for the `App` case, monotonicity is required, otherwise the induction hypothesis will not be applicable. Instead of induction along the computation of the fixpoint iteration, it is necessary to start the proof when `eval` already stabilized. This requires a completely different approach to prove the correctness of `eval`. A verification proof will also involve proving several auxiliary lemmas that are necessary and need to be proven in Isabelle[4].

[4]We sketch a few lemmas and proofs that might be required to attempt a verification proof for `eval` which are not shown here because of readability reasons.

Chapter 5

Conclusion

5.1 Result

In this thesis, we formalized two different semantics for a higher-order fixpoint algebra and provided a formal proof that both semantics are equal. We showcased how such a formal proof was carried out in Isabelle. In addition, we formalized the local model-checking algorithm in Isabelle. We sketched which issues arise when attempting to follow a similar proof structure to prove that the model-checking algorithm is correct.

The formalization revealed a plethora of challenges one is faced when attempting to carry out formal proofs in Isabelle compared to classical pen-and-paper proofs. The explicit listing of all required assumptions for a lemma (or theorem) greatly impacts the readability of the lemma itself and the proof state while proving the lemma. An additional challenge was being faced with non-terminating functions because the automatic termination checker could not verify the termination, although the functions all terminated in theory. While switching to the ***function*** definition to enable more powerful termination automation resulted in successfully proving termination, these proofs took a noticeable amount of time on all machines this formalization was tested. This had a big impact on working with these functions because after each change made to a proof document, Isabelle will re-run every proof, including the termination proofs, which leads to delays when a termination proofs takes roughly 20 seconds to finish.

The statefulness of the model-checking algorithm required to carry around state explicitly in the form of return values which makes the function definition hard to work with and less readable since the insertion of `let` expressions is needed.

Isabelle offers powerful proof automation and ***sledgehammer*** is a useful tool to automate proofs and discover needed lemmas. Isabelle's standard library is extensive and it is not always easy to find needed lemmas by hand, instead ***sledgehammer*** is a great tool for finding needed lemmas, although there where a few occasions where Isabelle crashed because ***sledgehammer*** consumed too much RAM. This mainly happened on older computers and impacts the workflow inside Isabelle.

5.2 Future Work

There are a few directions for further work regarding this formalization. A formal proof of the correctness of the local model-checking algorithm could be carried out. To improve

the workflow and reduce delays, it might be useful to carry out all termination proofs that require explicit handling manually. This would reduce the waiting time since no termination proof search has to be carried out by Isabelle in the background.

It would also be interesting to see how a formalization of the abstract higher-order fixpoint algebra for arbitrary order using a proof assistant that supports dependent types would look like.

Bibliography

[1] J. Avigad and J. Harrison, "Formally verified mathematics," *Commun. ACM*, vol. 57, no. 4, pp. 66–75, 2014.

[2] L. C. Paulson, "Computational logic: Its origins and applications," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 474, 2017.

[3] T. C. Hales, M. Adams, G. J. Bauer, *et al.*, "A formal proof of the kepler conjecture," *Forum of Mathematics, Pi*, vol. 5, 2017.

[4] J. Avigad, K. Donnelly, D. Gray, and P. Raff, "A formally verified proof of the prime number theorem," *ACM Trans. Comput. Log.*, vol. 9, p. 2, 2007.

[5] G. Gonthier, "Formal proof – the four-color theorem," *Notices of the American Mathematical Society*, vol. 55, no. 11, pp. 1382–1393, 2008.

[6] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin, "The verisoft approach to systems verification," in *VSTTE*, 2008.

[7] G. Klein, K. Elphinstone, G. Heiser, *et al.*, "Sel4: Formal verification of an os kernel," in *SOSP '09*, 2009.

[8] X. Leroy, "Formal certification of a compiler back-end or: Programming a compiler with a proof assistant," POPL '06, pp. 4254, 2006.

[9] G. Klein and T. Nipkow, "A machine-checked model for a java-like language, virtual machine, and compiler," *ACM Transactions on Programming Languages and Systems*, vol. 28, pp. 619–695, 2006.

[10] F. Bruse, J. Kreiker, M. Lange, and M. Sälzer, "Local higher-order fixpoint iteration," *Information and Computation*, vol. 289, p. 104 963, 2022.

[11] B. Knaster, "Un theoreme sur les functions d'ensembles," *Ann. Soc. Polon. Math.*, vol. 6, pp. 133–134, 1928.

[12] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, vol. 5, pp. 285–309, 1955.

[13] S. C. Kleene, "On notation for ordinal numbers," *Journal of Symbolic Logic*, vol. 3, pp. 150–155, 1938.

[14] M. Wenzel, L. C. Paulson, and T. Nipkow, "The Isabelle Framework," in *TPHOLs*, 2008.

[15] L. C. Paulson. "Isabelle's logics." (2022), [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2022/doc/logics-ZF.pdf (visited on 12/20/2022).

[16] F. Haftmann and L. Bulwahn. "Code generation from Isabelle/HOL theories." (2022), [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2022/doc/codegen.pdf (visited on 12/20/2022).

[17] P. Lammich, "Generating verified LLVM from Isabelle/HOL," in *ITP*, 2019.

[18] F. Haftmann, G. Klein, T. Nipkow, and N. Schirmer. "Latex sugar for Isabelle documents." (2022), [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2022/doc/sugar.pdf (visited on 12/20/2022).

[19] L. C. Paulson, T. Nipkow, and M. Wenzel, "From LCF to Isabelle/HOL," *Formal Aspects of Computing*, vol. 31, pp. 675–698, 2019.

[20] M. Gordon, "From lcf to hol: A short history," in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000, pp. 169185.

[21] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.

[22] A. Krauss. "Defining recursive functions in Isabelle/HOL," [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2022/doc/functions.pdf (visited on 12/20/2022).

[23] M. Wenzel, "Isabelle/Isar — a versatile environment for human-readable formal proof documents," Ph.D. dissertation, Institut für Informatik, Technische Universität München, 2002.

[24] J. C. Blanchette, L. Bulwahn, and T. Nipkow, "Automatic proof and disproof in Isabelle/HOL," in *Frontiers of Combining Systems (FroCoS 2011)*, C. Tinelli and V. Sofronie-Stokkermans, Eds., vol. 6989, 2011, pp. 12–27.

[25] G. Barthe and T. Coquand, "An introduction to dependent type theory," in *APPSEM*, 2000.