To The Graduate School:

The members of the Committee approve the thesis of Tjark Weber presented on July 26, 2002.

James L. Caldwell, Chairman

John R. Cowles

Sylvia Hobart

APPROVED:

Jeffrey Van Baalen, Head, Department of Computer Science

Stephen E. Williams, Dean, The Graduate School

Weber, Tjark, Program Transformations in NUPRL, M.S., Department of Computer Science, August, 2002

This thesis presents a formalization of program transformations and their general categorical framework in NUPRL. It gives formal definitions of catamorphisms and anamorphisms and formal, constructive proofs for when an arrow is a catamorphism or anamorphism. Necessary and sufficient conditions for when a function is a catamorphism are proved constructively, and a program transformation is extracted from the proofs. An instance of Bird's fusion theorem for binary trees is verified in NUPRL, and applied to the QUICKSORT algorithm to formally prove the algorithm correct.

# Program Transformations in Nuprl

by

Tjark Weber

A thesis submitted to the Department of Computer Science
and The Graduate School of The University of Wyoming
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Laramie, Wyoming
August, 2002

© 2002 by Tjark Weber

# Acknowledgments

I would like to thank my advisor James L. Caldwell for his helpful comments and his guidance on this thesis. I appreciate his help as much as the fact that he let me work on my own whenever possible. He was also the one who asked me to join the Master's Program in the Department of Computer Science at the University of Wyoming.

I would like to acknowledge support received from a Rockwell-Collins University grant.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9985239. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Among the online resources I used are Google™, CiteSeer, and LEO. I would like to thank everyone there for making their services available freely.

Thorsten Altenkirch willingly answered my questions about [GHA01]. Natarajan Shankar provided me with some of his PVS files developed for [Sha96]. The LaTeX template this thesis is based on was written by Thomas Böhne. I also want to thank him for proof-reading my thesis. Any remaining errors are of course my fault.

Last but not least I would like to thank my parents Klaus and Frauke Weber for their great support over many, many years.

# Contents

# List of Figures

*All men by nature desire to know.*
Aristotle, 350 B.C.

# Chapter 1

# Introduction

Writing good software is difficult. Large computer programs consist of several million lines of code, and with current techniques, it appears to be almost impossible to write programs that are both correct and fast.

Hundreds of programming languages and CASE[1] tools have been developed over the past decades to deal with the increasing complexity of software systems. Formal methods have successfully been applied to verify the correctness of critical systems [BM92, BS93, Sto96, HE99, Her00].

But good software should not only be correct and easy to write, understand and maintain. Despite the increasing speed of personal computers and declining hardware costs, programs should also be efficient. Unfortunately, the easy and the efficient solutions to a problem are often not the same.

Program transformations are a way to address both the issue of correctness and the issue of efficiency. By expressing algorithms in certain patterns, we can apply many standardized proof and optimization techniques [Hut98, GJ98, Hut99]. The following example was taken from [GJS93].

**Example 1.0.1.** Suppose we want to write a function `all` that tests whether all elements in a list $L$ satisfy a predicate $p$. In the programming language HASKELL [Bir98], `all` could be written as follows:

[1] Computer Aided Software Engineering

```
all p L = and (map p L)
```

The function *map* applies $p$ to every element in $L$, creating an intermediate list of boolean values. The function *and* then computes the conjunction of those values.

Creating the intermediate list requires time and memory. The following is a more efficient version of `all` that operates directly on the structure of $L$, and thereby avoids creating a second list:

```
all' p L = f L, where
   f []    =  True,
   f h::t  =  p h && f t
```

Here [] denotes the empty list, and $h :: t$ denotes a list with head $h$ and tail $t$.

Deforestation [Dav87, Wad88, GJS93] can be used to transform the first version of `all` into the second, less concise, but more efficient version `all'`.

Applying program transformations manually is error-prone and—in the case of larger programs—practically impossible. Therefore the transformation process needs to be mechanized. Many compilers make use of program transformations to improve the performance of the generated code [KH89, BGS94, Jon96]. Thus it is important that we can express transformations as algorithms, and that we have clear criteria for when a program transformation can be applied. Furthermore, we want to have proof that the transformation does not change the semantics of a program.

NUPRL [CAB+86, Jac94] is a proof development system that supports the interactive creation of programs and formal mathematical proofs. Functional programs can be written in NUPRL's base language, a form of typed $\lambda$-calculus, and then proved to be correct by formal proofs in NUPRL's constructive type theory. On the other hand, algorithms that are 'correct by construction' can be extracted from a NUPRL proof [How93]. The NUPRL system is described in greater detail in Chapter 3.

## 1.1 Objectives

We will formalize catamorphisms, anamorphisms and the required notions of category theory in NUPRL. We will formally prove conditions for when an arrow in a category is a catamorphism or anamorphism. We will give a constructive characterization of the non-constructive results from [GHA01], and we will show that those results do not, in general, hold constructively. For a certain class of constructive functions, we will present a transformation that writes such a function as a catamorphism. We will verify an instance of Bird's fusion transformation [Bir95] for binary trees in Nuprl. We will then implement the well-known Quicksort algorithm [Hoa61], apply the fusion transformation to it, and formally prove the algorithm correct.

## 1.2 Organization

This thesis is organized as follows:

Chapter 2, *Background*, briefly discusses previous work about program transformations and approaches to their mechanization.

Chapter 3, *The Nuprl System*, explains the key elements of the Nuprl system and discusses some features of Nuprl that are frequently used in this thesis.

Chapter 4, *Category Theory in Nuprl*, defines some basic notions of category theory and presents a formalization of those notions in Nuprl.

Chapter 5, *Catamorphisms and Anamorphisms*, defines catamorphisms and anamorphisms using notions of category theory from Chapter 4. Necessary and sufficient conditions for when an arrow is a catamorphism or an anamorphism are formalized and proved in Nuprl.

Chapter 6, *When is a Function a Catamorphism?*, studies the special case of arrows in the category of sets, i.e. (total) functions. A proof is given that the results in [GHA01] are not valid constructively, conditions are identified under which a constructive function $h$ is a catamorphism, and an algorithm is presented that computes a function $g$ with $h = \mathsf{fold}\, g$ in this case.

Chapter 7, *Bird's Fusion Transformation*, presents a program transformation that replaces the composition of an anamorphism and a catamorphism with a single function, thereby eliminating the intermediate data structure that is constructed by the anamorphism. An instance of Bird's fusion transformation for binary trees is formalized and verified in Nuprl.

Chapter 8, *Example: Quicksort*, applies the fusion transformation to the well-known Quicksort algorithm. The algorithm is implemented in Nuprl, and a formal proof of its correctness is given.

Chapter 9, *Conclusions*, finally lists our contributions, summarizes the results, and points out possible future work.

# Chapter 2

# Background

In this chapter we give a brief overview of previous work on program transformations and approaches to their mechanization. *Program transformation*, also known as 'software generation', 'program synthesis' or 'program calculation', is the process of changing one program into another. We distinguish between *rephrasings*—transformations where the source and target language are the same—and *translations*, i.e. transformations with different source and target languages. This thesis will only consider the former kind of transformations.

## 2.1 Squiggol

In the 1980s, Richard Bird [Bir84] and Lambert Meertens [Mee86] developed a calculus for functional programs. This calculus is now known as the 'Bird-Meertens Formalism', or 'SQUIGGOL'. Its goal is to provide an algebra that allows the derivation of efficient programs from less efficient, but obviously correct specifications. A simple example of this approach was given in Chapter 1.

The SQUIGGOL calculus originally only considered recursion functionals on lists. *Catamorphisms* (from the Greek preposition $\kappa\alpha\tau\alpha$ meaning 'downwards') are recursive functions that destruct a list. The `all` function defined in Chapter 1 is an example of a catamorphism. *Anamorphisms* (from the Greek $\alpha\nu\alpha$ meaning 'upwards') are functions that construct a list. The `flatten` function that flattens a binary tree into a list (see Chapter 8) is an example of an anamorphism. Catamorphisms are also frequently called 'folds', and 'unfolds' is another term for anamorphisms. With the introduction of category theory however, SQUIGGOL was generalized to other recursive data types [Mal90], and the terms catamorphism and anamorphism (or fold and unfold) are now used in a more general sense. The SQUIGGOL calculus and its extensions from category theory provide the theoretical foundation for our work.

## 2.2 Special-Purpose Transformation Tools

Several special-purpose software tools are commercially available for program transformations. The ‘DMS® Software Reengineering Toolkit’ [Bax01], the ‘Kestrel Interactive Development System (KIDS)’ [Smi90], ‘Stratego’ [Vis01a], the ‘Transformation Assisted Multiple Program Realization System (TAMPR)’ [BHW97], and the ‘TXL Transformation System’ [Cor00] are among the more noteworthy ones that are not purely of academical interest. E. Visser gives an overview of the techniques used by these and other systems in [Vis01b]. Since we will formalize and verify program transformations using the general-purpose system NUPRL in this thesis, we do not describe any of those special-purpose systems in detail here.

## 2.3 General-Purpose Verification Tools

General-purpose verification and proof development tools like NUPRL were not specifically designed for reasoning about program transformations. Therefore formalizing a transformation and proving its correctness (i.e. that it does not change the semantics of a program) can be a challenge in a general-purpose system. The only other application of a general-purpose theorem prover to program transformations that we are aware of is by N. Shankar [Sha96], who used the ‘PVS Specification and Verification System’ [OSR95] to implement Bird’s fusion transformation (see Chapter 7) and Wand’s continuation-based transformation [Wan80]. PVS is similar to NUPRL in many ways. It supports recursive definitions, subtyping, dependent function and product types, parametric theories, induction, and many other features essential for the formalization of program transformations. The main difference between the two systems originates from NUPRL having a constructive type theory as its logical foundation.

## 2.4 Previous Formalizations of Category Theory

While general-purpose theorem provers apparently have not been applied to program transformations frequently, the basic notions of category theory however have been formalized in a number of systems before, also to some extend in NUPRL.

A significant amount of category theory has been formalized in the Mizar system [Miz], a formal system based on Tarski Grothendieck set theory. Rydeheard and Burstall considered computational aspects of category theory in their 1988 book [RB88]. In NUPRL, some formalization of category theory has been done previously [AP90]. However, this work was in a much earlier version of the system. In the Coq system [BBC+97], Carvalho [Car98] has implemented a segment of category theory based on Huet and Saibi's formalization [HS98]. Aczel [Acz93] has formalized categories in the LEGO system [LP92].

# Chapter 3

# The Nuprl System

The 'NUPRL Proof Development System' is "a computer system which [...] supports the interactive creation of proofs, formulas, and terms in a formal theory of mathematics" [CAB+86]. Its first version was developed by R. Constable and J. Bates around 1985. Until today, NUPRL's constructive type theory and its approach to displaying and editing mathematical text distinguish NUPRL from other theorem provers. Coq [BBC+97], LEGO [LP92], and ALF [MN94] are all formal constructive systems, but NUPRL's type theory is unique in its expressive power: NUPRL proof extracts are untyped $\lambda$-terms, while the systems mentioned above are limited to the typed $\lambda$-calculus. Thus NUPRL can extract general recursion functions, not just primitive recursive [Cal02].

A comprehensive description of NUPRL is given in [CAB+86], and more up-to-date information can be found in [Jac94]. The NUPRL project also has its own web site at http://www.nuprl.org/. For this thesis we used Version 4.2 of the NUPRL system. Some of NUPRL's more important features are discussed in this chapter. Other, more technical aspects of the system are explained throughout the thesis when necessary.

## 3.1 The Type Theory

The NUPRL type theory is the logical foundation of the NUPRL system. It is a constructive type theory based on [ML82]. The relationship between types and sets is non-trivial, e.g. see [Acz99, Wer97]. One of the main differences between classical set theory and NUPRL's type theory is that equality of sets is extensional (i.e. sets are equal if and only if they contain the same elements), whereas type equality is intensional (i.e. types are equal if and only if they have the same 'structure'). For the most part however, we will just model sets as types in this thesis. Issues related to the differences between sets and types will be specifically mentioned.

Types are constructed out of a few basic types by the use of type constructors. Among the types and constructors used in this thesis are the following:

**Basic types.** The empty type `Void`, the type of integers, $\mathbb{Z}$, and the subtype $\mathbb{N} = \{z \in \mathbb{Z} \mid z \geq 0\}$. Also the type $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$ of boolean values, and the type $\mathtt{Unit} = \{\cdot\}$, which contains a single element.

**Dependent product.** Suppose $A$ is a type, and $B_x$ is a type for every $x \in A$. Then $x{:}A \times B_x$ is a type, containing all pairs $(a, b)$ such that $a \in A$ and $b \in B_a$. If $B_x$ is the same for all $x \in A$, we simply write $A \times B$ to denote the cartesian product of $A$ and $B$. The '`spread`' operator is a destructor for the product type.

**Dependent function.** Suppose $A$ is a type, and $B_x$ is a type for every $x \in A$. Then $x : A \to B_x$ is a type, containing all functions $f$ from $A$ to $\cup_{x \in A} B_x$ such that $f(a) \in B_a$ for all $a \in A$. If $B_x$ is the same for all $x \in A$, we simply write $A \to B$ to denote the type of all total functions[1] from $A$ to $B$.

**Disjoint union.** Suppose $A$ and $B$ are types. Then $A + B$ is a type, containing all elements of the form $inl(a)$ for $a \in A$, and $inr(b)$ for $b \in B$. NUPRL provides an operator '`decide`' that can be used to destruct the disjoint union type.

**Subtype.** Suppose $A$ is a type and $P_a$ is a proposition in which $a$ of type $A$ may occur free. Then $\{a : A \mid P_a\}$ is the type of all $a \in A$ for which $P_a$ is true.

**Recursive types.** NUPRL has a built-in recursive data type *List* of (finite) lists, with constructors [] for the empty list, and :: for concatenation. It also provides a way of defining other recursive types, as well as recursive functions.

**Type universes.** Types in NUPRL are elements of so-called *type universes*. NUPRL has a cumulative hierarchy of universes $\mathbb{U}_1, \mathbb{U}_2, \mathbb{U}_3, ...,$ where each universe contains all previous type universes (i.e. $\mathbb{U}_i \in \mathbb{U}_j$ for all $i < j$). Type universes are closed under the type constructors listed above. Most of our definitions and theorems are generic in the universe level; then we simply write $\mathbb{U}_i$ or $\mathbb{U}$. We use $\mathbb{U}'$ and $\mathbb{U}_{i'}$ as short notations for $\mathbb{U}_{i+1}$.

[1] Since NUPRL's type theory is constructive, these are really just all *computable* functions.

## 3.2 Constructive Aspects: Proofs as Programs

Logical propositions in Nuprl are defined via the type constructors:

- *False* is defined as `Void`.
- $A \wedge B$ is defined as $A \times B$.
- $A \vee B$ is defined as $A + B$.
- $A \implies B$ is defined as $A \rightarrow B$.
- $\forall x\!:\!T.\ P(x)$ is defined as $x\!:\!T \rightarrow P_x$.
- $\exists x\!:\!T.\ P(x)$ is defined as $x\!:\!T \times P_x$.
- $\neg P$ is defined as $P \rightarrow \mathit{False}$.

Thus every proposition corresponds to a type, and a proposition is provable if and only if the corresponding type is inhabited. Proving a proposition is equivalent to constructing a term that inhabits the corresponding type.

As a consequence, the law of excluded middle does not hold, i.e. $p \vee \neg p$ is not true for every proposition $p$. Instead, $p \vee \neg p$ only holds when we know which of the two possible cases $p$ or $\neg p$ is true. Although many classical theorems with proofs relying on the law of excluded middle will be unprovable in Nuprl, it has the advantage that proofs in Nuprl are constructive. Proving a theorem of the form

$$\forall x_1, x_2, \ldots, x_k.\ \exists y_1, y_2, \ldots, y_n.\ R(x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_n)$$

(where $R$ is some relation) yields an algorithm that takes $k$ arguments $x_1, x_2, \ldots, x_k$ and returns $n+1$ values $y_1, y_2, \ldots, y_n, \rho$ such that $\rho$ is a proof of

$$R(x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_n).$$

For instance, proving a theorem like 'for every list of integers $L$ exists a list $M$ such that $M$ is an ordered permutation of $L$' would yield a sorting algorithm for lists of integers, together with evidence that $M$ is indeed an ordered permutation of $L$. Which algorithm we get depends on the proof we give; different algorithms correspond to different proofs.

Thus we have two essentially different ways of reasoning about algorithms in Nuprl [How93, Cal98]: On the one hand, we can implement an algorithm (as a term in Nuprl's type theory) and then formally verify the algorithm by stating and proving its relevant properties. This is done for the QUICKSORT algorithm in Chapter 8 of this thesis. On the other hand, we can get an algorithm that is 'correct by construction' out of a formal proof. This approach is used to extract a program transformation in Chapter 6.

## 3.3 Well-Formedness

We say a term is *well-formed* if and only if it is a member of some type. NUPRL requires well-formedness proofs for all terms used in the statement of a theorem; i.e. to prove a theorem, we do not only have to show that is is valid, but we also have to show that it is well-formed.

Often NUPRL can discharge well-formedness goals automatically. But in general, the problem of checking well-formedness is not decidable, so manual interaction is required in some cases. Also the structure of NUPRL's proof rules sometimes forces us to prove the well-formedness of the same expression several times. One means of avoiding this is to prove well-formedness goals separately—in this case they are usually discharged automatically. However, sometimes proving well-formedness of a theorem can be the most difficult part of proving the theorem. This is one of the major reasons why even simple informal proofs can turn out to be remarkably tedious in NUPRL.

## 3.4 Display Forms, Abstractions, Proofs

Mathematical content in NUPRL is stored in '*theory*' files. A theory is a list of (usually closely related) definitions ('*abstractions*'), display forms, theorems and comments. Abstractions, which can have any number of arguments, are used to express one term by another. Each abstraction usually has an associated well-formedness theorem and an associated display form. The well-formedness theorem often simply states that the abstraction has a type; it is used automatically when NUPRL tries to prove the well-formedness of terms containing the abstraction. Display forms control how an abstraction is displayed by the NUPRL system. They can be used quite effectively to retain the usual mathematical notation. Display forms allow the use of special symbols (e.g. '$\forall$'), they can define rules for parenthesis and whitespacing, they can hide arguments, and much more. Of course their flexibility can also be misused to implement an 'abuse of notation' that can turn understanding abstractions into a game of luck.

Display forms, abstractions, theorems and their proofs are created interactively. A NUPRL proof of a theorem has tree structure. To prove a theorem, *tactics* are applied to the proof goal. These tactics, which are based on the combination and repeated application of a few simple rewrite rules, can generate zero, one or more new subgoals, thereby mapping a partial proof to a complete or partial proof. NUPRL has predefined tactics for reasoning about integer arithmetic and systems of inequalities, for (mathematical, complete, and measure) induction, for rewriting and substituting expressions, for structural induction on recursively defined types, and many more. Further tactics can be defined by the user. The AUTO tactic combines several simple tactics and is often useful to prove less complex proof goals in a single step. NUPRL also provides *tacticals* (e.g. THEN, REPEAT) to apply different tactics in a single proof step. However, with the increasing amount of computing power that is available, the system could probably benefit from a more powerful AUTO tactic. Currently, despite the existence of fairly advanced tactics, very elementary proofs can be quite tedious sometimes.

The NUPRL type theory, the concepts of theory files, display forms, abstractions and theorems, the available tactics and tacticals, and many other aspects of the NUPRL system are described in greater detail in [CAB+86] and [Jac94]. Also we did not describe the NUPRL editor and user interface in this chapter since knowledge of it is not necessary for understanding this thesis.

# Chapter 4

# Category Theory in Nuprl

Category theory is “a general mathematical theory of structures and sy[s]tems of structures” [Mar02]. This comparatively new field of mathematics arose in the study of certain group-theoretic and topological properties by S. Eilenberg and S. McLane [EM42, EM45] around 1942. Thanks to their general nature, categories have successfully been applied to problems in topology, algebra, geometry, functional analysis, and computer science.

This chapter presents some basic notions of category theory, and a formalization in Nuprl. We will need the notions presented here in Chapter 5 again to give a general definition of catamorphisms and anamorphisms.

## 4.1 Categories

A category consists of a class of *objects*, together with a class of *arrows* between these objects, which fulfill certain properties. The following definition is based on [Mac97].

**Definition 4.1.1 (Category).** A *category* $\mathcal{C}$ is a six-tuple

$$\mathcal{C} = (\mathit{Obj}, \mathit{Arr}, \mathit{dom}, \mathit{cod}, \cdot, \mathit{id}),$$

where $\mathit{Obj}$ is a class of *objects*, $\mathit{Arr}$ is a class of *arrows*, and $\mathit{dom} : \mathit{Arr} \to \mathit{Obj}$ and $\mathit{cod} : \mathit{Arr} \to \mathit{Obj}$ are functions denoting an arrow's *domain* and *codomain* respectively. $\cdot : \mathit{Arr} \to \mathit{Arr} \xrightarrow{p} \mathit{Arr}$ is a partial function, the *composition operator*, and $\mathit{id} : \mathit{Obj} \to \mathit{Arr}$ is the *identity operator*.

The operations are subject to the following properties:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "f"'] & B \arrow[d, "g"] \arrow[dl, "id(B)"'] \\ B \arrow[r, "g"'] & C \end{tikzcd}$$

Figure 4.1: $id(B) \cdot f = f$ and $g \cdot id(B) = g$

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, "g\cdot f"'] & B \arrow[d, "g"] \arrow[dr, "h\cdot g"] & \\ & C \arrow[r, "h"'] & D \end{tikzcd}$$

Figure 4.2: $h \cdot (g \cdot f) = (h \cdot g) \cdot f$

1. For all $f, g \in Arr$, $g \cdot f$ is defined if and only if $cod(f) = dom(g)$, and in this case, $dom(g \cdot f) = dom(f)$ and $cod(g \cdot f) = cod(g)$. We say that $f$ and $g$ are *composable* in $\mathcal{C}$ if and only if $cod(f) = dom(g)$.

2. For all $A \in Obj$, $dom(id(A)) = cod(id(A)) = A$. $id(A)$ is called the *identity arrow* on $A$.

3. **Unit Law**: For all $B \in Obj$ and $f \in Arr$ with $cod(f) = B$, $id(B) \cdot f = f$. For all $B \in Obj$ and $g \in Arr$ with $dom(g) = B$, $g \cdot id(B) = g$ (see Figure 4.1).

4. **Associativity**: For all $f, g, h \in Arr$ with $cod(f) = dom(g)$ and $cod(g) = dom(h)$, $h \cdot (g \cdot f) = (h \cdot g) \cdot f$ (see Figure 4.2).

Figure 4.3 shows a natural formalization of the type of categories in NUPRL. The dependent product type is used extensively here to state the properties that the composition operator and the identity operator must satisfy. The well-formedness proof for the `category` type requires manual verification of the two properties $dom(h \cdot g) = dom(g)$ (if $g$ and $h$ are composable) and $cod(f) = dom(id(cod(f)))$. Altogether, the proof is about twelve steps long.[1]

We also define six projection functions (with associated display forms and well-formedness-theorems) `cat_obj`, `cat_arr`, `cat_dom`, `cat_cod`, `cat_op`, and `cat_id` to map a category to its first, second, third, ..., component in NUPRL. These projection functions give us easy access to the specific components of a category that we need in the statement of a theorem or a proof. Also we are free to chose more self-explanatory names and display forms for them than `C.2.2.2.1` for example, which would be the standard NUPRL notation for the fourth component (i.e. the codomain operator) of a category `C`. Furthermore, when `C` is of type `Cat{i}`, NUPRL does not

[1] In NUPRL, several proof tactics can be combined into a single proof step by so-called 'tacticals', e.g. THEN. We do not count tactics that were combined in this way as separate steps. Therefore a single proof step often involves the application of two, three or even more different tactics.

```
* ABS category
Cat{i} ==
Obj:𝕌
× A:𝕌
× dom:(A ⟶ Obj)
× cod:(A ⟶ Obj)
× o:{o:g:A ⟶ f:{f:A| cod f = dom g} ⟶
    {h:A| dom h = dom f ∧ cod h = cod g} |
    ∀f,g,h:A. cod f = dom g ∧ cod g = dom h ⇒
    (h o g) o f = h o (g o f)}
× {id:p:Obj ⟶ {f:A| dom f = p ∧ cod f = p} |
    ∀f:A. (id (cod f)) o f = f ∧ f o (id (dom f)) = f}
```

Figure 4.3: Abstraction `category`

know that `C.1` is of type $\mathbb{U}$ (unless we decompose `C`). It does, however, know that `C_Obj` is of type $\mathbb{U}$ since we proved this as a well-formedness theorem. We will define similar projection functions for every component of every product type defined in this thesis.

We use the terms *morphism* and arrow interchangeably. Given a category $\mathcal{C}$, the morphisms from $p$ to $q$ (in $\mathcal{C}$) are those arrows in $\mathcal{C}$ with domain $p$ and codomain $q$. Figure 4.4 shows a NUPRL implementation for the type of all morphisms from $p$ to $q$. Here the well-formedness goal, i.e. to show that for every category $\mathcal{C}$ and for every object $p$ and $q$ in $\mathcal{C}$, `morphism[C](p,q)` is a type, is discharged in a single step by the AUTO tactic.

```
* ABS morphism
Mor[C](p,q) == {f:C_Arr| C_dom f = p ∧ C_cod f = q}
```

Figure 4.4: Abstraction `morphism`

To simplify notation, we write $f : A \to B$ for an arrow $f$ with domain $A$ and codomain $B$.

## 4.2 The Category of Types

One obvious interpretation of a category is to think of the objects as sets and of the arrows as functions. In fact we will need the *category of sets*, $\mathcal{SET}$, later in this thesis.[2] The objects of $\mathcal{SET}$ are all sets, the arrows are all (total) functions between sets, and the domain operator and codomain operator map a function to its domain and codomain respectively. Composition in $\mathcal{SET}$ is the usual composition of functions, and the identity arrow on a set $A$ is simply the identity function on $A$.

When we try to define the *category of types* in NUPRL, there are two major differences that must be considered. Firstly, the class of all sets is not a set itself. Similarly, the notion 'type of all types' leads to contradictions [Gir72]. Therefore the type of objects in the category of types cannot contain *all* (other) types. The solution is to make it contain only all types up to a certain (but arbitrary) universe level $i$. The arrows are all functions between those types.[3]

Secondly, an arrow cannot just be a function $f : A \to B$: There is no constructive means to extract the function's domain and codomain. Using the dependent product type, we formalize an arrow as a triple $(A, B, f)$, where $A$ is the domain, $B$ is the codomain, and $f : A \to B$. Thus the category's domain operator just maps every arrow to its first component, while the codomain operator maps every arrow to its second component. Of course the definitions of the composition and identity operator must be compatible with this realization of arrows. Figure 4.5 shows the final definition of the category of types, or *large category*, in NUPRL.

```
* ABS large_category
large_category{i} ==
<𝕌
, A:𝕌 × B:𝕌 × (A ⟶ B)
, λf.f.1
, λf.f.2.1
, λg,f.<f.1, g.2.1, g.2.2 o f.2.2>
, λp.<p, p, λx.x> >
```

Figure 4.5: Abstraction `large_category`

To prove that the category of types is in fact a category, we should essentially only have to verify the unit law and the associativity of the composition operator. However, if we try a direct proof, NUPRL also generates a number of well-formedness goals and auxiliary subgoals when we decompose the `category` product type. Proving them is rather tedious, and the difficulties are propagated when we want to prove that other structures (with more complex objects and arrows than in the category of types) are

[2] While is is often convenient to think of the objects as sets and of the arrows as functions, $\mathcal{SET}$ is, however, just one example of a category.

[3] This gives us a different 'category of types' for each universe level $i$. However, all theorems that we prove in this thesis hold for any universe level.

categories. Therefore we prove a lemma `category_if` first (see Figure 4.6).[4] The lemma states that every six-tuple with components of the appropriate types is in fact a category if the unit law holds and if the composition operator is associative. This way we have to deal with the additional well-formedness goals and auxiliary subgoals only once, namely when we prove the lemma. All future proofs in NUPRL that verify that some six-tuple is a category can then be simplified by using this lemma.

```
* THM category_if
∀Obj,A:𝕌.  ∀dom,cod:A ⟶ Obj.
∀o:g:A ⟶ f:{f:A| cod f = dom g} ⟶
    {h:A| dom h = dom f ∧ cod h = cod g} .
∀id:p:Obj ⟶ {f:A| dom f = p ∧ cod f = p} .
(∀f,g,h:A. cod f = dom g ∧ cod g = dom h ⇒
    (h o g) o f = h o (g o f))
⇒ (∀f:A. (id (cod f)) o f = f ∧ f o (id (dom f)) = f)
⇒ <Obj, A, dom, cod, o, id> ∈ Cat{i}
```

Figure 4.6: Theorem `category_if`

The proof of the lemma requires about 36 steps, most of them to prove the well-formedness goals and auxiliary subgoals. The formal proof that `large_category` is a category (see Figure 4.7) then requires about 13 steps, one of which is of course the instantiation of the lemma. The easier well-formedness subgoals are discharged by NUPRL's AUTO tactic. The lemma `comp_assoc`, which is part of the `FUN_1` library, is used to prove that composition of functions is associative. The lemmata `comp_id_l` and `comp_id_r` from the same library prove that the identity function is a left–identity and right–identity, respectively, for function composition. The polymorphic universe level $i'$ used in the well-formedness theorem is short NUPRL notation for $i + 1$.

```
* THM large_category_wf
large_category{i} ∈ Cat{i'}
```

Figure 4.7: Theorem `large_category_wf`

## 4.3 Dual Categories

Given any category $\mathcal{C}$, we get the *dual category* of $\mathcal{C}$ by swapping every arrow's domain and codomain, and swapping the order of arrow composition as well (i.e. $g \cdot f$ becomes

[4]This is really a specialized well-formedness lemma for the pairing constructor and would automatically be invoked by NUPRL if we had named it `pair_wf_category`.

$f \cdot g$).

**Definition 4.3.1 (Dual Category).** Suppose $\mathcal{C} = (Obj, Arr, dom, cod, \cdot, id)$ is a category. Then

$$\mathcal{C}^{op} = (Obj, Arr, cod, dom, \cdot^{op}, id)$$

is a category, where $g \cdot^{op} f$ is defined as $f \cdot g$. $\mathcal{C}^{op}$ is the *dual category* of $\mathcal{C}$.

The NUPRL abstraction defining the dual category is shown in Figure 4.8. The well-formedness theorem shown in Figure 4.9 proves that the dual category is in fact a category. The proof is straightforward and requires about six steps, including the instantiation of the lemma `category_if`.

```
* ABS dual_category
C^op == <C_Obj, C_Arr, C_cod, C_dom, λg,f.f C_op g, C_id>
```

Figure 4.8: Abstraction `dual_category`

```
* THM dual_category_wf
∀C:Cat{i}.  C^op ∈ Cat{i}
```

Figure 4.9: Theorem `dual_category_wf`

In this thesis we will not actually construct the dual category for any given category. We will however define several terms that are dual to each other, e.g. initial and terminal objects, algebras and coalgebras, homomorphisms and cohomomorphisms, and—last but not least—catamorphisms and anamorphisms. Therefore the concept of duality will be important throughout the rest of this chapter.

## 4.4 Initial and Terminal Objects

*Initial* objects are objects that have exactly one arrow going from them to every object in the category. *Terminal* objects are objects that have exactly one arrow coming to them from every object in the category. In other words, terminal objects are initial objects in the dual category.

**Definition 4.4.1 (Initial Object).** Suppose $\mathcal{C}$ is a category. We say an object $A$ in $\mathcal{C}$ is *initial* (in $\mathcal{C}$) if and only if for every object $B$ in $\mathcal{C}$, there exists a unique arrow $f : A \to B$.

**Definition 4.4.2 (Terminal Object).** Suppose $\mathcal{C}$ is a category. We say an object $B$ in $\mathcal{C}$ is *terminal* (in $\mathcal{C}$) if and only if for every object $A$ in $\mathcal{C}$, there exists a unique arrow $f : A \to B$.

This implies that if $A$ is initial or terminal, the only arrow $A \to A$ is the identity arrow on $A$. In general, a category can have zero, one or more initial and terminal objects. In the category $\mathcal{SET}$, for example, the empty set is the only initial object, and every singleton set (i.e. every set with exactly one element) is a terminal object. The corresponding NuPRL abstractions `initial` and `terminal` are shown in Figure 4.10. The well-formedness theorems for these abstractions are proved in a single step each by the Auto tactic.

```
* ABS initial
C-initial(p) == ∀q:C_Obj.  ∃!f:C_Arr.  C_dom f = p ∧ C_cod f = q

* ABS terminal
C-terminal(p) == ∀q:C_Obj.  ∃!f:C_Arr.  C_dom f = q ∧ C_cod f = p
```

Figure 4.10: Abstractions `initial` and `terminal`

Here '∃!' is short for 'there exists a unique'. An abstraction defining this quantifier in terms of '∃' and '∀' is shown in Figure 4.11.

```
* ABS exists_unique
∃!x:T. B[x] == ∃x:T. B[x] ∧ (∀y:T. B[y] ⇒ y = x)
```

Figure 4.11: Abstraction `exists_unique`

## 4.5 Functors

*Functors* are arrows between categories. A functor from $\mathcal{C}$ to $\mathcal{D}$, where $\mathcal{C}$ and $\mathcal{D}$ are categories, maps objects in $\mathcal{C}$ to objects in $\mathcal{D}$ and arrows in $\mathcal{C}$ to arrows in $\mathcal{D}$ such that these maps are compatible with the categories' domain operators, the codomain operators, the composition of arrows, and with the identity operators.

**Definition 4.5.1 (Functor).** Suppose $\mathcal{C} = (Obj_{\mathcal{C}}, Arr_{\mathcal{C}}, dom_{\mathcal{C}}, cod_{\mathcal{C}}, \cdot_{\mathcal{C}}, id_{\mathcal{C}})$ and $\mathcal{D} = (Obj_{\mathcal{D}}, Arr_{\mathcal{D}}, dom_{\mathcal{D}}, cod_{\mathcal{D}}, \cdot_{\mathcal{D}}, id_{\mathcal{D}})$ are categories. Let $F_O : Obj_{\mathcal{C}} \to Obj_{\mathcal{D}}$, and $F_M : Arr_{\mathcal{C}} \to Arr_{\mathcal{D}}$. The pair $F = (F_O, F_M)$ is a *functor* from $\mathcal{C}$ to $\mathcal{D}$ (write $F : \mathcal{C} \to \mathcal{D}$) if and only if

1. $\forall f \in Arr_{\mathcal{C}} : dom_{\mathcal{D}}(F_M(f)) = F_O(dom_{\mathcal{C}}(f))$,
2. $\forall f \in Arr_{\mathcal{C}} : cod_{\mathcal{D}}(F_M(f)) = F_O(cod_{\mathcal{C}}(f))$,
3. $\forall f, g \in Arr_{\mathcal{C}} : dom_{\mathcal{C}}(g) = cod_{\mathcal{C}}(f) \implies F_M(g \ \cdot_{\mathcal{C}} \ f) = F_M(g) \ \cdot_{\mathcal{D}} \ F_M(f)$,
4. $\forall A \in Obj_{\mathcal{C}} : F_M(id_{\mathcal{C}}(A)) = id_{\mathcal{D}}(F_O(A))$.

To simplify notation, we write $FA$ for $F_O(A)$ if $A$ is an object in $\mathcal{C}$, and $Ff$ for $F_M(f)$ if $f$ is an arrow in $\mathcal{C}$.

To formalize the type of all functors from $\mathcal{C}$ to $\mathcal{D}$ in NUPRL, we define a functor as a four-tuple, where the first component is the functor's domain (i.e. $\mathcal{C}$), the second component is the codomain (i.e. $\mathcal{D}$), and the other two components are the object function $F_O$ and the arrow function $F_M$. Since we restrict the first and second component of the functor to be of type `Cat{i}`, where $i$ is a universe level, we also need to specify the universe level $i$ in the definition of the functor type. This level should be greater than or equal to the maximum of the universe levels of $\mathcal{C}$ and $\mathcal{D}$. Typically the universe levels of $\mathcal{C}$ and $\mathcal{D}$ will be $i$, and so we use level $i$ for the type of functors between $C$ and $D$.

```
* ABS functor
Functor{i}(C,D) ==
{C:Cat{i}}
× {D:Cat{i}}
× O:(C_Obj → D_Obj)
× {M:C_Arr → D_Arr|
(∀f:C_Arr.  D_dom (M f) = O (C_dom f) ∧ D_cod (M f) = O (C_cod f))
c∧ ((∀f:C_Arr.  ∀g:{g:C_Arr| C_dom g = C_cod f} .
    M (g C_op f) = (M g) D_op (M f))
∧ (∀p:C_Obj.  M (C_id p) = D_id (O p)))}
```

Figure 4.12: Abstraction `functor`

Figure 4.12 shows the NUPRL abstraction defining the `functor` type. The proof of the associated well-formedness theorem requires about nine steps: We have to verify that if $f$ and $g$ are in $Arr_{\mathcal{C}}$ with $dom_{\mathcal{C}}(g) = cod_{\mathcal{C}}(f)$, then $dom_{\mathcal{D}}(F_M(g)) = cod_{\mathcal{D}}(F_M(f))$ to prove that $F_M(g) \ \cdot_{\mathcal{D}} \ F_M(f)$ is well-formed in this case.

## 4.6 Algebras and Coalgebras

Given a category $\mathcal{C}$ and a functor $F : \mathcal{C} \to \mathcal{C}$, an *algebra* over $F$ is a pair $(A, f)$, where $A$ is an object and $f : FA \to A$ is an arrow in $\mathcal{C}$. A *coalgebra* is a pair $(A', f')$, where

$A'$ is an object and $f' : A' \to FA'$ is an arrow in $\mathcal{C}$. In other words, coalgebras are algebras over the dual category.

**Definition 4.6.1 (Algebra).** Suppose $\mathcal{C}$ is a category and $F : \mathcal{C} \to \mathcal{C}$ is a functor. An *algebra* (over $F$) is a pair $(A, f)$, where $A$ is an object in $\mathcal{C}$ and $f : FA \to A$ is an arrow in $\mathcal{C}$.

**Definition 4.6.2 (Coalgebra).** Suppose $\mathcal{C}$ is a category and $F : \mathcal{C} \to \mathcal{C}$ is a functor. A *coalgebra* (over $F$) is a pair $(A, f)$, where $A$ is an object in $\mathcal{C}$ and $f : F \to FA$ is an arrow in $\mathcal{C}$.

For example, if $F$ is the identity functor on $\mathcal{C}$ (mapping every object and arrow in $\mathcal{C}$ to itself), then for every object $A$ in $\mathcal{C}$, $(A, id(A))$ is both an algebra and a coalgebra over $F$. However, algebras and coalgebras for a given functor $F$ may or may not exist.

```
* ABS algebra
Algebra(F) == p:F_dom_Obj ×
    {f:F_dom_Arr| F_dom_dom f = F_O p ∧ F_dom_cod f = p}

* ABS coalgebra
Coalgebra(F) == p:F_dom_Obj ×
    {f:F_dom_Arr| F_dom_dom f = p ∧ F_dom_cod f = F_O p}
```

Figure 4.13: Abstractions `algebra` and `coalgebra`

Using the dependent product type, defining the types of all algebras and coalgebras over a functor $F : \mathcal{C} \to \mathcal{C}$ is straightforward in NUPRL. The corresponding abstractions are shown in Figure 4.13. The proofs of the two well-formedness theorems `algebra_wf` and `coalgebra_wf`, showing that $Algebra(F)$ and $Coalgebra(F)$ are well-formed types for every functor $F : \mathcal{C} \to \mathcal{C}$, require about four steps each, mainly because we must prove that the functor's domain and codomain are equal.

## 4.7 Homomorphisms and Cohomomorphisms

Given two algebras $(A, f)$ and $(B, g)$ over the same functor $F$, a *homomorphism* from $(A, f)$ to $(B, g)$ is an arrow $h : A \to B$ such that the diagram shown in Figure 4.14 commutes. Similarly, a *cohomomorphism* from $(A', f')$ to $(B', g')$, where $(A', f')$ and $(B', g')$ are coalgebras over $F$, is an arrow $h' : A' \to B'$ that makes the square shown in Figure 4.15 commute.

$$\begin{tikzcd} FA \arrow[r, "Fh"] \arrow[d, "f"'] & FB \arrow[d, "g"] \\ A \arrow[r, "h"'] & B \end{tikzcd}$$

Figure 4.14: $h \cdot f = g \cdot Fh$

$$\begin{tikzcd} A' \arrow[r, "h'"] \arrow[d, "f'"'] & B' \arrow[d, "g'"] \\ FA' \arrow[r, "Fh'"'] & FB' \end{tikzcd}$$

Figure 4.15: $Fh' \cdot f' = g' \cdot h'$

**Definition 4.7.1 (Homomorphism).** Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor, and $(A, f)$ and $(B, g)$ are algebras over $F$. A *homomorphism* from $(A, f)$ to $(B, g)$ is an arrow $h : A \to B$ such that $h \cdot f = g \cdot Fh$.

**Definition 4.7.2 (Cohomomorphism).** Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor, and $(A', f')$ and $(B', g')$ are coalgebras over $F$. A *cohomomorphism* from $(A', f')$ to $(B', g')$ is an arrow $h' : A' \to B'$ such that $Fh' \cdot f' = g' \cdot h'$.

The definition of the type of all homomorphisms (and similarly, all cohomomorphisms) over a functor $F$ in NUPRL is relatively straightforward again and shown in Figure 4.16. We use the dependent product type to include the homomorphism's domain $(A, f)$ and codomain $(B, g)$. Thus a homomorphism is a triple (instead of just an arrow $h : A \to B$, from which we could only get $A$ and $B$ by applying the domain and codomain operators, but neither $f$ or $g$). The same technique was used for arrows in the category of types and for the `functor` type before. The well-formedness theorems `homomorphisms_wf` and `cohomomorphisms_wf` are proved in about eight steps each; the main work is to verify that the arrows $f$ and $h$ and $Fh$ and $g$ are composable (analogous $Fh'$ and $f'$ and $g'$ and $h'$ for cohomomorphism).

In analogy to the `morphism` type, we also define the type of all homomorphisms over $F$ from a given algebra $(A, f)$ to another given algebra $(B, g)$ (and similarly, the type of all cohomomorphisms with a given domain and codomain), see Figure 4.17. The corresponding well-formedness theorems are proved in a single step each.

## 4.8 The Category of Algebras

We can think of homomorphisms as arrows between algebras. Once we define a composition of homomorphisms that is associative, and for every algebra an identity homomorphism that satisfies the unit law, we have a new category: The category with algebras (over a functor $F$) as objects, and with homomorphisms (between those algebras) as arrows.

```
* ABS homomorphisms
Hom(F) ==
A:Algebra(F)
× B:Algebra(F)
× {f:F_dom_Arr|
    (F_dom_dom f = A_obj) c∧ (F_dom_cod f = B_obj) c∧
    (f F_dom_op A_arr = B_arr F_dom_op (F_M f))}

* ABS cohomomorphisms
Cohom(F) ==
A:Coalgebra(F)
× B:Coalgebra(F)
× {f:F_dom_Arr|
    (F_dom_dom f = A_obj) c∧ (F_dom_cod f = B_obj) c∧
    ((F_M f) F_dom_op A_arr = B_arr F_dom_op f)}
```

Figure 4.16: Abstractions `homomorphisms` and `cohomomorphisms`

```
* ABS homomorphisms_dom_cod
Hom[F](A,B) == {f:Hom(F)| f_dom = A ∧ f_cod = B}

* ABS cohomomorphisms_dom_cod
Cohom[F](A,B) == {f:Cohom(F)| f_dom = A ∧ f_cod = B}
```

Figure 4.17: Abstractions `homomorphisms_dom_cod` and `cohomomorphisms_dom_cod`

### 4.8.1 The Composition of Homomorphisms

Homomorphisms are arrows in a category $\mathcal{C}$. The obvious approach to define their composition is to simply define it as the composition in $\mathcal{C}$. Figure 4.18 shows the NUPRL abstraction `hom_composition`. The formal definition of course has to be compatible with our realization of homomorphisms as triples.

```
* ABS hom_composition
g o_hom[F] f == <f_dom, g_cod, g_arr F_dom_op f_arr>
```

Figure 4.18: Abstraction `hom_composition`

However, we have to verify that the composition of two homomorphisms is again a homomorphism. Suppose $(A, f)$, $(B, g)$ and $(C, h)$ are algebras, and $a : (A, f) \to (B, g)$ and $b : (B, g) \to (C, h)$ are homomorphisms. Then $b \cdot a$ is a homomorphism from

$$\begin{tikzcd} FA \arrow[r, "Fa"] \arrow[d, "f"'] & FB \arrow[r, "Fb"] \arrow[d, "g"] & FC \arrow[d, "h"] \\ A \arrow[r, "a"'] & B \arrow[r, "b"'] & C \end{tikzcd}$$

Figure 4.19: The Composition of Homomorphisms

$(A, f)$ to $(C, h)$: Clearly $b \cdot a : A \to C$, and

$$\begin{array}{ll} & (b \cdot a) \cdot f \\ = & \quad \{ \text{ associativity } \} \\ & b \cdot (a \cdot f) \\ = & \quad \{ \text{ homomorphisms } \} \\ & b \cdot (g \cdot Fa) \\ = & \quad \{ \text{ associativity } \} \\ & (b \cdot g) \cdot Fa \\ = & \quad \{ \text{ homomorphisms } \} \\ & (h \cdot Fb) \cdot Fa \\ = & \quad \{ \text{ associativity } \} \\ & h \cdot (Fb \cdot Fa) \\ = & \quad \{ \text{ functors } \} \\ & h \cdot F(b \cdot a). \end{array}$$

In other words, the diagram shown in Figure 4.19 commutes.

In Nuprl, we state this result as a well-formedness theorem for `hom_composition` (see Figure 4.20). The formal proof, although it is based on the calculation shown above, requires about 95 steps: Many of the transformations used above generate one or two well-formedness subgoals which require several steps to be proved.

```
* THM hom_composition_wf
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀K,L,M:Algebra(F). ∀f:Hom[F](K,L).
∀g:Hom[F](L,M). g o_hom[F] f ∈ Hom[F](K,M)
```

Figure 4.20: Theorem `hom_composition_wf`

Now we prove two useful lemmata about the composition of homomorphisms and their domain and codomain: The domain of $g \cdot f$ is equal to the domain of $f$, and the codomain of $g \cdot f$ is equal to the codomain of $g$ (see Figure 4.21). Both lemmata are proved in two steps each.

```
* THM hom_composition_dom
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀K,L,M:Algebra(F). ∀f:Hom[F](K,L).
∀g:Hom[F](L,M). (g o_hom[F] f)_dom = f_dom

* THM hom_composition_cod
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀K,L,M:Algebra(F). ∀f:Hom[F](K,L).
∀g:Hom[F](L,M). (g o_hom[F] f)_cod = g_cod
```

Figure 4.21: Theorems `hom_composition_dom` and `hom_composition_cod`

We finally have to prove that the composition of homomorphisms is associative (see Figure 4.22). Informally this is immediate because we simply defined the composition of homomorphisms as the composition in $\mathcal{C}$, which is associative by the second axiom of category theory. In NUPRL however, we have to show equality in the `homomorphisms` type, and including all well-formedness goals and auxiliary subgoals that we end up with, the proof is about 140 steps long (which makes it one of the longest proofs in the `CATEGORY_THEORY` library).

```
* THM hom_composition_assoc
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀K,L,M,N:Algebra(F). ∀f:Hom[F](K,L).
    ∀g:Hom[F](L,M). ∀h:Hom[F](M,N).
h o_hom[F] (g o_hom[F] f) = (h o_hom[F] g) o_hom[F] f
```

Figure 4.22: Theorem `hom_composition_assoc`

### 4.8.2 The Identity Homomorphism

For every algebra $(A, f)$, the identity arrow on $A$ is a homomorphism from $(A, f)$ to $(A, f)$: Clearly $id(A) : A \to A$, and

$$
\begin{array}{ll}
 & id(A) \cdot f \\
= & \quad \{ \text{ unit law } \} \\
 & f \\
= & \quad \{ \text{ unit law } \} \\
 & f \cdot id(FA) \\
= & \quad \{ \text{ functors } \} \\
 & f \cdot F(id(A)).
\end{array}
$$

Hence the following definition makes sense.

**Definition 4.8.1 (Identity Homomorphism).** Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor, and $(A, f)$ is an algebra over $F$. Then the *identity homomorphism* on $(A, f)$ is defined as the identity arrow on $A$.

Figure 4.23 shows the NUPRL abstraction `identity_hom`. Recall, we defined homomorphisms to be triples. The formal proof that this is in fact a homomorphism (see Figure 4.24) is about 23 steps long.

```
* ABS identity_hom
id_hom[F](A) == <A, A, F_dom_id A_obj>
```

Figure 4.23: Abstraction `identity_hom`

```
* THM identity_hom_wf
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀A:Algebra(F).
    id_hom[F](A) ∈ Hom[F](A,A)
```

Figure 4.24: Theorem `identity_hom_wf`

Before we use this definition, we prove two lemmata again, namely that the domain and the codomain of the identity homomorphism on $(A, f)$ are both equal to $(A, f)$. Both lemmata (see Figure 4.25) are proved in a single step each.

```
* THM hom_dom_id
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀A:Algebra(F). id_hom[F](A)_dom = A

* THM hom_cod_id
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀A:Algebra(F). id_hom[F](A)_cod = A
```

Figure 4.25: Theorems `hom_dom_id` and `hom_cod_id`

The identity homomorphism satisfies the unit law for the composition of homomorphisms (see Figure 4.26). This follows immediately from the unit law for the identity arrow. The formal proof, however, is surprisingly tedious. We have to verify that certain arrows are composable several times. Altogether, the proof that the identity homomorphism cancels out on the left is about 60 steps long, and the proof that it cancels out on the right has approximately the same length.

```
* THM hom_comp_id_l
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀h:Hom(F).
    id_hom[F](h_cod) o_hom[F] h = h

* THM hom_comp_id_r
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀h:Hom(F).
    h o_hom[F] id_hom[F](h_dom) = h
```

Figure 4.26: Theorems `hom_comp_id_l` and `hom_comp_id_r`

### 4.8.3 Definition of the Category of Algebras

We are now ready to define the *category of algebras*. As said before, this category has as objects all algebras over a given functor $F : \mathcal{C} \rightarrow \mathcal{C}$ (where $\mathcal{C}$ is a category), and as arrows all homomorphisms between these algebras. The domain operator and codomain operator map each homomorphism to its domain algebra and codomain algebra respectively. The composition of homomorphisms is defined as the usual composition of arrows in $\mathcal{C}$, and the identity operator maps each algebra $A$ to the identity homomorphism on $A$. Figure 4.27 shows the NUPRL abstraction `algebra_category`.

```
* ABS algebra_category
algebra_category(F) ==
<Algebra(F)
, Hom(F)
, λh.h_dom
, λh.h_cod
, λh,g.h o_hom[F] g
, λA.id_hom[F](A)>
```

Figure 4.27: Abstraction `algebra_category`

Theorem `algebra_category_wf` (see Figure 4.28) proves that this is in fact a category. The formal proof is about 29 steps long and uses several lemmata: `category_if` to get rid of a number of well-formedness goals, `hom_composition_wf` to prove that the composition of two homomorphisms is again a homomorphism, `hom_composition_dom` and `hom_composition_cod` to prove that the composition operator returns arrows with the proper domain and codomain, `hom_composition_assoc` to prove that it is associative, `identity_hom_wf` to prove that the identity homomorphism is in fact a homomorphism, `hom_dom_id` and `hom_cod_id` to prove that it has the proper domain and codomain, and `hom_comp_id_l` and `hom_comp_id_r` to prove the unit law. A few proof steps use other lemmata to deal with the (somewhat technical) difference between the types `homomorphisms` and `homomorphisms_dom_cod`.

```
* THM algebra_category_wf
∀C:Cat{i}.  ∀F:Functor{i}(C,C). algebra_category(F) ∈ Cat{i}
```

Figure 4.28: Theorem `algebra_category_wf`

## 4.9 The Category of Coalgebras

The *category of coalgebras* can be defined completely analogous to the category of algebras. The *composition of cohomomorphisms* is defined as the composition of arrows in the original category $\mathcal{C}$ (just like the composition of homomorphisms before), and the *identity cohomomorphism* on a coalgebra $(A, f)$ is simply the identity arrow on $A$ again. With these definitions, we can easily verify that the composition of two cohomomorphisms always is a cohomomorphism again, and that associativity and the unit law are satisfied. The proofs are not identical to the proofs presented for homomorphisms and algebras, but dual—meaning that we have to swap domain and codomain, $A$ and $FA$, and the order of arrow composition sometimes.

This is probably best illustrated by a small example, so we verify that the identity cohomomorphism on a coalgebra $(A, f)$ is in fact a cohomomorphism from $(A, f)$ to $(A, f)$: Clearly $id(A) : A \to A$, and

$$\begin{array}{ll} & F(id(A)) \cdot f \\ = & \quad \{ \text{ functors } \} \\ & id(F(A)) \cdot f \\ = & \quad \{ \text{ unit law } \} \\ & f \\ = & \quad \{ \text{ unit law } \} \\ & f \cdot id(A). \end{array}$$

Compared to the proof given for the identity homomorphism, the steps of reasoning are pretty much the same, with a few adjustments made for duality.

Unfortunately, ByDuality however is not a valid proof tactic (and would be hard to implement, because the use of dual notions does not only require us to give dual arguments, but it can also change the order in which subgoals occur in a Nuprl proof tree). Therefore all proofs from the previous section had to be modified for cohomomorphisms and coalgebras by hand, and their size is about the same as for homomorphisms and algebras before. Figures 4.29 and 4.30 show the main results of this section: A formal definition of the category of coalgebras, together with a NUPRL theorem proving that this is in fact a category.

```
* ABS coalgebra_category
coalgebra_category(F) ==
<Coalgebra(F)
, Cohom(F)
, λh.h_dom
, λh.h_cod
, λh,g.h o_cohom[F] g
, λA.id_cohom[F](A)>
```

Figure 4.29: Abstraction `coalgebra_category`

```
* THM coalgebra_category_wf
∀C:Cat{i}.  ∀F:Functor{i}(C,C). coalgebra_category(F) ∈ Cat{i}
```

Figure 4.30: Theorem `coalgebra_category_wf`

The following chapter defines catamorphisms and anamorphisms as certain arrows in the category of algebras and in the category of coalgebras, respectively.

# Chapter 5

# Catamorphisms and Anamorphisms

Catamorphisms ('folds') and anamorphisms ('unfolds') are certain arrows in the category of algebras and in the category of coalgebras, respectively. They can be used to specify many algorithms on lists, streams, trees and other recursive data types. More importantly, various optimization and proof techniques are known for algorithms that are expressed as a catamorphism or anamorphism [Hut98, GJ98, Hut99].

This chapter defines catamorphisms and anamorphisms using notions from category theory that were introduced in the previous chapter. Necessary and sufficient conditions for when an arrow is a catamorphism or an anamorphism are formalized and proved in NUPRL.

## 5.1 Catamorphisms

Suppose $\mathcal{C}$ is a category and $F : \mathcal{C} \to \mathcal{C}$ is a functor. Recall the category of algebras over $F$ defined in Chapter 4. We say an algebra $(\mu F, in)$ is *initial* if and only if it is an initial object in this category; that is, for every algebra $(A, f)$, there exists a unique homomorphism $h : (\mu F, in) \to (A, f)$.

**Example 5.1.1.** Let $\mathbf{1} = \{\cdot\}$ denote a set with exactly one element, and let $+$ denote the disjoint union. For a set $X$, consider the functor $\mathcal{L}_X : \mathcal{SET} \to \mathcal{SET}$, defined by $\mathcal{L}_X(A) = \mathbf{1} + (X \times A)$ and $\mathcal{L}_X(f) = id(\mathbf{1}) + (id(X) \times f)$. This functor has an initial algebra $(\mu\mathcal{L}_X, in) = (List(X), nil + cons)$, where $List(X)$ is the set of all finite lists over $X$, and $nil : \mathbf{1} \to List(X)$ and $cons : X \times List(X) \to List(X)$ are constructors

$$\begin{tikzcd} F(\mu F) \arrow[r, "F(\mathsf{fold}\, f)"] \arrow[d, "in"'] & FA \arrow[d, "f"] \\ \mu F \arrow[r, dotted, "\mathsf{fold}\, f"'] & A \end{tikzcd}$$

Figure 5.1: $(\mathsf{fold}\, f) \cdot in = f \cdot F(\mathsf{fold}\, f)$

for this set.[1] We write $[]$ for $nil(\cdot)$, the empty list, and $x :: L$ for $cons(x, L)$.

Fixing an initial algebra $(\mu F, in)$, we define $\mathsf{fold}\, f$ to be this unique homomorphism from $(\mu F, in)$ to $(A, f)$. Hence $\mathsf{fold}\, f$ is the unique arrow that makes the square shown in Figure 5.1 commute.

**Definition 5.1.2 (fold).** Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor and $(\mu F, in)$ is an initial algebra. Then for every algebra $(A, f)$,

$$\mathsf{fold}\, f$$

is defined as the unique homomorphism from $(\mu F, in)$ to $(A, f)$.

We say an arrow $h$ is a *catamorphism* if and only if it can be written as $\mathsf{fold}\, f$ for some arrow $f$.

**Example 5.1.3.** Consider the functor $\mathcal{L}_X$ with its initial algebra $(List(X), nil{+}cons)$. Suppose $(A, f)$ is an algebra over $\mathcal{L}_X$. Then $f : \mathbf{1} + (X \times A) \to A$ can be written as $f = f_0 + f_1$, where $f_0 : \mathbf{1} \to A$ and $f_1 : (X \times A) \to A$. We can prove by structural induction on $List(X)$ that every catamorphism $h : List(X) \to A$ satisfies the two equations

$$\begin{aligned} h([]) &= f_0(\cdot), \\ h(x :: L) &= f_1(x, h(L)). \end{aligned}$$

On the other hand, every function $h$ that can be written in this form is a catamorphism on $List(X)$. Examples are

$$\begin{aligned} length([]) &= 0, \\ length(x :: L) &= 1 + length(L) \end{aligned}$$

[1] A formal proof of this is given in Chapter 6.

to compute the length of a list,

$$\begin{aligned}\sum([]) &= 0,\\ \sum(x :: L) &= x + \sum(L)\end{aligned}$$

to compute the sum of a list of numbers, or the functions

$$\begin{aligned}\mathit{and}([]) &= \mathit{True},\\ \mathit{and}(x :: L) &= x \wedge \mathit{and}(L)\end{aligned}$$

and

$$\begin{aligned}(\mathit{map}\ p)([]) &= [],\\ (\mathit{map}\ p)(x :: L) &= p(x) :: (\mathit{map}\ p)(L)\end{aligned}$$

mentioned in Chapter 1.

Definition 5.1.2 implies the following *universal property* of the fold operator [Mal90].

**Theorem 5.1.4 (Universal Property of fold).** *Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor and $(\mu F, in)$ is an initial algebra. Furthermore, suppose that $(A, f)$ is an algebra and that $h : \mu F \to A$. Then*

$$h = \mathsf{fold}\, f \iff h \cdot in = f \cdot Fh.$$

*Proof.* If $h : \mu F \to A$ is equal to $\mathsf{fold}\, f$, then by definition of fold, $h$ is a homomorphism from $(\mu F, in)$ to $(A, f)$. Therefore $h \cdot in = f \cdot Fh$. This proves the '$\Rightarrow$' direction of the theorem.

For the other direction, assume $h : \mu F \to A$ satisfies the equation $h \cdot in = f \cdot Fh$. Then $h$ is a homomorphism from $(\mu F, in)$ to $(A, f)$. Since there exists only one such homomorphism (namely $\mathsf{fold}\, f$), we have $h = \mathsf{fold}\, f$. □

Although we defined fold as an operator mapping algebras over $F$ to arrows of $\mathcal{C}$, we did not actually specify an *algorithm* to *compute* $\mathsf{fold}\, f$, given an algebra $(A, f)$. We only know that $\mathsf{fold}\, f$ is the unique homomorphism from $(\mu F, in)$ to $(A, f)$, but this may be the most specific way of describing $\mathsf{fold}\, f$ that we have.

Therefore defining fold in NUPRL is not straightforward. One possible approach is to prove the existence of a function *fold_fun* from $\mathit{Algebra}(F)$, the type of all algebras over $F$, into the type $\mathcal{C}_{Arr}$ of all arrows in $\mathcal{C}$, such that $\mathit{fold\_fun}(A, f) \cdot in = f \cdot F(\mathit{fold\_fun}(A, f))$ for every algebra $(A, f)$. The actual `fold` operator would then be defined as (the first component of) the extract of a proof of this theorem.

```
* DISP fold_df
<h:arrow:*>=fold[<C:category:*>,<F:functor:*>,<I:algebra:*>]
    (<f:algebra:*>)
== fold(<C>; <F>; <I>; <f>; <h>)

* ABS fold
h=fold[C,F,I](f) ==
(F_dom_dom h = I_obj) c∧ (F_dom_cod h = f_obj) c∧
(h F_dom_op I_arr = f_arr F_dom_op (F_M h))
```

Figure 5.2: Display Form `fold_df` and Abstraction `fold`

We decided to define `fold` in a different way that completely avoids dealing with proof extracts. In NUPRL, `fold` is defined as a relation $Algebra(F) \rightarrow \mathcal{C}_{Arr} \rightarrow \mathbb{P}$ such that `fold((A,f),h)` holds if and only if $h \cdot in = f \cdot Fh$. We use NUPRL's display form facility [Jac94] to display `fold((A,f),h)` as an equation `h = fold(A,f)` (see Figure 5.2).

```
* THM fold_wf
∀C:Cat{i}.  ∀F:Functor{i}(C,C).
∀I:{I:Algebra(F)| algebra_category(F)-initial(I)} .
∀f:Algebra(F). ∀h:F_dom_Arr.
h=fold[C,F,I](f) ∈ ℙ
```

Figure 5.3: Theorem `fold_wf`

The well-formedness theorem `fold_wf` which is shown in Figure 5.3 simply proves that `h = fold(A,f)` is a well-formed proposition. The proof is about twelve steps long. In particular, we have to verify that $in$ and $h$ and $Fh$ and $f$ are composable arrows when $h : \mu F \rightarrow A$. Since the composition of two arrows is defined only if the second arrow's domain is equal to the first arrow's codomain, the equation $h \cdot in = f \cdot Fh$ is not well-formed for arbitrary arrows $h$. We ensured that $h$ has the proper domain and codomain by using NUPRL's `c∧` operator in the definition of `fold`. This operator can be characterized by the following proof rule:

$$\frac{\Gamma \vdash \phi \qquad \Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \;\texttt{c}\wedge\; \psi},$$

i.e. we may assume $\phi$ in the proof of $\psi$.

Now it is not hard to prove in NUPRL that for every algebra $(A, f)$, there exists a unique arrow $h$ such that $h = \mathsf{fold}\, f$. The theorem `fold_exists_unique` proving this

```
* THM fold_exists_unique
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀I:Algebra(F).
algebra_category(F)-initial(I) ⇒
    (∀f:Algebra(F). ∃!h:F_dom_Arr.  h=fold[C,F,I](f) )
```

Figure 5.4: Theorem `fold_exists_unique`

is shown in Figure 5.4. The existence of $h$ follows from the existence of a homomorphism from $(\mu F, in)$ to $(A, f)$, and the uniqueness of $h$ follows from the uniqueness of this homomorphism. Note the difference between Figure 5.3 and Figure 5.4 in the way we stated that $I$ is an initial algebra. In NUPRL using a hypothesis of the form $x \in \{y \in T \mid P[y]\}$ gives us $P[x]$ as a *'hidden'* hypothesis—that is, we cannot use it unless we can prove that either $P[x]$ or the proof goal has no computational content (i.e. is *'squash-stable'*). In the proof of `fold_exists_unique` however, we need the fact that $I$ is an initial algebra to get our hands on a homomorphism from $I$ to $(A, f)$. So neither the hypothesis nor the proof goal are squash-stable in this case, and therefore having '$I$ is an initial algebra' as a hidden hypothesis is not strong enough, i.e. we must make it an explicit antecedent to the theorem. This means that the computational content will be a function that expects an argument that is evidence of $I$ being initial. Proving the existence of $h$ requires about 20 proof steps, and proving its uniqueness requires about 66 steps in NUPRL. Together with a few preparatory steps, the proof is about 96 steps long.

## 5.2 When is an Arrow a Catamorphism?

The universal property of fold provides a technically complete answer to this question. An arrow $h : \mu F \to A$ is a catamorphism if and only if $h \cdot in = f \cdot Fh$ for some arrow $f : FA \to A$. However, usually only the arrow $h$ is given—how would we know if an arrow $f$ exists such that the above equation holds? And more importantly, how would we construct $f$ from $h$?

No general answer seems to be known to this question. The composition of a catamorphism and a homomorphism is a catamorphism [Bir95], and other results are known for other specific kinds of arrows. In this section we prove a result from [MFP91] in NUPRL: That every left-invertible arrow is a catamorphism.

**Definition 5.2.1 (Left-Invertible).** Suppose $\mathcal{C}$ is a category and $f$ is an arrow in $\mathcal{C}$. We say $f$ is *left-invertible* (in $\mathcal{C}$) if and only if there exists an arrow $g$ in $\mathcal{C}$ such that $g \cdot f = id(dom(f))$.

Of course we made the implicit assumption $dom(g) = cod(f)$ in the above definition, because otherwise $g \cdot f$ is not defined. For the same reason, we have to make this assumption explicit in the NUPRL abstraction `left_invertible`, which is shown in Figure 5.5. The associated well-formedness theorem, showing that `left_invertible` is a proposition, is then proved in two steps.

```
* ABS left_invertible
left-invertible[C](f) ==
    ∃g:{g:C_Arr| C-composable(f,g)} .  g C_op f = C_id (C_dom f)
```

Figure 5.5: Abstraction `left_invertible`

Equipped with this definition, we can now prove that every left-invertible arrow is a catamorphism.

**Theorem 5.2.2.** *Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor with an initial algebra $(\mu F, in)$, and $h : \mu F \to A$ is a left-invertible arrow in $C$. Then*

$$h = \mathsf{fold}\, f$$

*for some arrow $f : FA \to A$.*

*Proof.* Let $g : A \to \mu F$ be an arrow in $\mathcal{C}$ such that $g \cdot h = id(\mu F)$ (such a $g$ exists since $h$ is left-invertible). Then we have

$$\begin{array}{ll} & h \cdot in \\ = & \quad \{ \text{ unit law } \} \\ & h \cdot in \cdot id(F(\mu F)) \\ = & \quad \{ \text{ functors } \} \\ & h \cdot in \cdot F(id(\mu F)) \\ = & \quad \{ \text{ assumption } \} \\ & h \cdot in \cdot F(g \cdot h) \\ = & \quad \{ \text{ functors } \} \\ & h \cdot in \cdot (Fg \cdot Fh) \\ = & \quad \{ \text{ associativity } \} \\ & (h \cdot in \cdot Fg) \cdot Fh. \end{array}$$

Therefore $h = \mathsf{fold}(h \cdot in \cdot Fg)$ by the universal property of $\mathsf{fold}$. □

$$\begin{tikzcd}
F(\mu F) \arrow[r, "Fh"'] \arrow[d, "in"'] & FA \arrow[l, bend right, "Fg"'] \arrow[d, dotted, "f"] \\
\mu F \arrow[r, "h"'] & A \arrow[l, bend right, "g"']
\end{tikzcd}$$

Figure 5.6: Every Left-Invertible Arrow is a Catamorphism

The diagram in Figure 5.6 illustrates the proof idea. We need an arrow $f : FA \to A$ that makes the diagram commute. Clearly $h \cdot in \cdot Fg$ does the trick.[2]

The NUPRL theorem `left_invertible_implies_fold` is shown in Figure 5.7. The formal proof takes about 70 steps, mainly because of several well-formedness goals that need to be verified.

```
* THM left_invertible_implies_fold
∀C:Cat{i}.  ∀F:Functor{i}(C,C).
∀I:{I:Algebra(F)| algebra_category(F)-initial(I)} .
∀h:{h:F_dom_Arr| F_dom_dom h = I_obj} .
left-invertible[F_dom](h) ⇒ (∃f:Algebra(F). h=fold[C,F,I](f) )
```

Figure 5.7: Theorem `left_invertible_implies_fold`

Since this theorem proves the existence of an object (namely of an arrow $f$ such that $h = \mathsf{fold}\, f$), the proof extract—which is shown in Figure 5.8—is also worth a look. The extract is a function with five arguments: a category $\mathcal{C}$, a functor $F$, an initial algebra $I$, an arrow $h$, and a proof that $h$ is left-invertible, denoted as `%` in the extract. In NUPRL, a proof that $h$ is left-invertible is technically a pair ($g$, `%1`), where $g$ is an arrow and `%1` is a proof of $g \cdot h = id(dom(h))$. Similarly, the value returned by the function in this extract is a pair $((A, f),$ `%%`$)$, where $(A, f)$ is an algebra such that $h = \mathsf{fold}\, f$, and `%%` is a proof of this equality.

This becomes more evident if we reduce those $\lambda$-terms in the extract for which we know the argument. A few reduction steps give us the term shown in Figure 5.9. Now we can clearly see the witness term: It is the algebra `<F_dom_cod h, h F_dom_op`

[2]The diagram also illustrates another observation: We do not actually need $h$ to be left-invertible. The proof works when we only assume that $Fh$ is left-invertible. This is implied by the left-invertibility of $h$, but the converse is not true, so it is actually a weaker condition. If $g \cdot Fh = id(F(\mu F))$ for some arrow $g$, then $h = \mathsf{fold}(h \cdot in \cdot g)$.

```
λC,F,I,h,%.
let <g,%1> = %
in
(λ%2.(λ%3.(λ%4.(λ%5.(λ%6.(λ%7.
    <<F_dom_cod h, h F_dom_op (I_arr F_dom_op (F_M g))>
    , Ax
    , Ax
    , Ax>)
Ax)
Ax)
Ax)
Ax)
((λ%3.Ax) ext{functor_dom_cod_equal}{i:l}))
((λ%2.%2 C C F) ext{functor_dom_wf}{i:l})
```

Figure 5.8: Extract of `left_invertible_implies_fold`

`(I_arr F_dom_op (F_M g))>`. This is of course the same witness that our informal proof above used, only in Nuprl notation.

```
λC,F,I,h,%.
let <g,%1> = %
in
    <<F_dom_cod h, h F_dom_op (I_arr F_dom_op (F_M g))>
    , Ax
    , Ax
    , Ax>
```

Figure 5.9: Simplified Extract of `left_invertible_implies_fold`

## 5.3 Anamorphisms

*Anamorphisms* are the dual notion to catamorphisms. While catamorphisms are homomorphisms from an initial algebra in the category of algebras, anamorphisms are defined as cohomomorphisms to a terminal coalgebra in the category of coalgebras. A formalization of this category in Nuprl was presented in Chapter 4.

We say a coalgebra $(\nu F, out)$ is *terminal* if and only if it is a terminal object in the category of coalgebras; that is, for every coalgebra $(A, f)$, there exists a unique cohomomorphism $h : (A, f) \to (\nu F, out)$.

$$\begin{tikzcd}
A \arrow[r, dotted, "\mathsf{unfold}\, f"] \arrow[d, "f"'] & \nu F \arrow[d, "out"] \\
FA \arrow[r, "F(\mathsf{unfold}\, f)"'] & F(\nu F)
\end{tikzcd}$$

Figure 5.10: $F(\mathsf{unfold}\, f) \cdot f = out \cdot (\mathsf{unfold}\, f)$

**Definition 5.3.1.** Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor and $(\nu F, out)$ is a terminal coalgebra. Then for every coalgebra $(A, f)$,

$$\mathsf{unfold}\, f$$

is defined as the unique cohomomorphism from $(A, f)$ to $(\nu F, out)$.

Figure 5.10 illustrates this situation. We say an arrow $h$ is an *anamorphism* if and only if it can be written as $\mathsf{unfold}\, f$ for some arrow $f$. Recall the universal property of $\mathsf{fold}$; a similar universal property holds for the $\mathsf{unfold}$ operator:

**Theorem 5.3.2 (Universal Property of unfold).** *Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor and $(\nu F, out)$ is a terminal coalgebra. Furthermore, suppose that $(A, f)$ is a coalgebra and that $h : A \to \nu F$. Then*

$$h = \mathit{unfold}\, f \iff Fh \cdot f = out \cdot h.$$

*Proof.* Suppose $h : A \to \nu F$ is equal to $\mathsf{unfold}\, f$. Then $h$ is a cohomomorphism from $(A, f)$ to $(\nu F, out)$. Hence $Fh \cdot f = out \cdot h$.

For the '$\Leftarrow$' direction, suppose $h : A \to \nu F$ satisfies the equation $Fh \cdot f = out \cdot h$. Then $h$ is a cohomomorphism from $(A, f)$ to $(\mu F, out)$. Since there exists only one such cohomomorphism (namely $\mathsf{unfold}\, f$), we have $h = \mathsf{unfold}\, f$. $\square$

Using this universal property, we define `unfold`, similar to `fold` before, as a relation $\mathit{Coalgebra}(F) \to \mathcal{C}_{Arr} \to \mathbb{P}$. NUPRL's display form facility is used to display `unfold` as an equality nevertheless (see Figure 5.11).

The well-formedness theorem `unfold_wf` (see Figure 5.12) is proved in about twelve steps; the proof is dual to the proof of `fold_wf`. Again we use the `c`$\wedge$ operator in the definition of `unfold` to ensure that $h$ has the proper domain and codomain, so that $Fh \cdot f$ and $out \cdot h$ are both well-defined.

```
* DISP unfold_df
<h:arrow:*>=unfold[<C:category:*>,<F:functor:*>,<T:coalgebra:*>]
    (<f:coalgebra:*>)
== unfold(<C>; <F>; <T>; <f>; <h>)

* ABS unfold
h=unfold[C,F,T](f) ==
(F_dom_dom h = f_obj) c∧ (F_dom_cod h = T_obj) c∧
((F_M h) F_dom_op f_arr = T_arr F_dom_op h)
```

Figure 5.11: Display Form `unfold_df` and Abstraction `unfold`

```
* THM unfold_wf
∀C:Cat{i}.  ∀F:Functor{i}(C,C).
∀T:{T:Coalgebra(F)| coalgebra_category(F)-terminal(T)} .
∀f:Coalgebra(F). ∀h:F_dom_Arr.
h=unfold[C,F,T](f) ∈ ℙ
```

Figure 5.12: Theorem `unfold_wf`

As we did for fold above, we can now prove the existence of a unique arrow $h$ such that $h = \mathsf{unfold}\, f$ for every coalgebra $(A, f)$. The proof of `unfold_exists_unique`, which is shown in Figure 5.13, is dual to the proof of `fold_exists_unique` and also about 96 steps long.

```
* THM unfold_exists_unique
∀C:Cat{i}.  ∀F:Functor{i}(C,C). ∀T:Coalgebra(F).
coalgebra_category(F)-terminal(T) ⇒
    (∀f:Coalgebra(F). ∃!h:F_dom_Arr.  h=unfold[C,F,T](f) )
```

Figure 5.13: Theorem `unfold_exists_unique`

In the following section, we prove a result dual to the one that every left-invertible arrow is a catamorphism: that every *right-invertible* arrow is an anamorphism.

## 5.4 When is an Arrow an Anamorphism?

Again, a technically complete—but nevertheless unsatisfactory—answer is provided by the universal property of unfold. An arrow $h : A \to \nu F$ is an anamorphism if and only if $Fh \cdot f = out \cdot h$ for some arrow $f : A \to FA$. This answer is unsatisfactory because it is not at all clear how to check if such an arrow $f$ exists, and neither is it clear how to express $f$ in terms of $h$ even if we know that such a $f$ exists.

E. Meijer, M. Fokkinga, and R. Paterson [MFP91] dualized their result that every left-invertible arrow is a catamorphism to anamorphisms: Every right-invertible arrow is an anamorphism.

**Definition 5.4.1 (Right-Invertible).** Suppose $\mathcal{C}$ is a category and $f$ is an arrow in $\mathcal{C}$. We say $f$ is *right-invertible* (in $\mathcal{C}$) if and only if there exists an arrow $g$ in $\mathcal{C}$ such that $f \cdot g = id(cod(f))$.

The corresponding NUPRL abstraction is shown in Figure 5.14. The well-formedness theorem for it simply states that this abstraction is a proposition, and is proved in two steps.

```
* ABS right_invertible
right-invertible[C](f) ==
    ∃g:{g:C_Arr| C-composable(g,f)} .  f C_op g = C_id (C_cod f)
```

Figure 5.14: Abstraction `right_invertible`

**Theorem 5.4.2.** *Suppose $\mathcal{C}$ is a category, $F : \mathcal{C} \to \mathcal{C}$ is a functor with a terminal coalgebra $(\nu F, out)$, and $h : A \to \nu F$ is a right-invertible arrow in $C$. Then*

$$h = \mathsf{unfold}\, f$$

*for some arrow $f : A \to FA$.*

*Proof.* Let $g : \nu F \to A$ be an arrow in $\mathcal{C}$ such that $h \cdot g = id(\nu F)$ (such a $g$ exists since $h$ is right-invertible). Then we have

$$\begin{array}{ll} & out \cdot h \\ = & \quad \{ \text{ unit law } \} \\ & id(F(\nu F)) \cdot out \cdot h \\ = & \quad \{ \text{ functors } \} \\ & F(id(\nu F)) \cdot out \cdot h \\ = & \quad \{ \text{ assumption } \} \\ & F(h \cdot g) \cdot out \cdot h \\ = & \quad \{ \text{ functors } \} \\ & (Fh \cdot Fg) \cdot out \cdot h \\ = & \quad \{ \text{ associativity } \} \\ & Fh \cdot (Fg \cdot out \cdot h). \end{array}$$

$$
\begin{tikzcd}
A \arrow[r, "h"'] \arrow[d, dotted, "f"'] & \nu F \arrow[l, bend right, "g"'] \arrow[d, "out"] \\
FA \arrow[r, "Fh"'] & F(\nu F) \arrow[l, bend right, "Fg"']
\end{tikzcd}
$$

Figure 5.15: Every Right-Invertible Arrow is an Anamorphism

Therefore $h = \mathsf{unfold}(Fg \cdot out \cdot h)$ by the universal property of $\mathsf{unfold}$. $\square$

This situation is illustrated by the commutative diagram shown in Figure 5.15.[3] The NUPRL theorem `right_invertible_implies_unfold` shown in Figure 5.16 is proved in about 70 steps.

```
* THM right_invertible_implies_unfold
∀C:Cat{i}.  ∀F:Functor{i}(C,C).
∀T:{T:Coalgebra(F)| coalgebra_category(F)-terminal(T)} .
∀h:{h:F_dom_Arr| F_dom_cod h = T_obj} .
right-invertible[F_dom](h) ⇒ (∃f:Coalgebra(F). h=unfold[C,F,T](f) )
```

Figure 5.16: Theorem `right_invertible_implies_unfold`

Figure 5.17 shows the simplified extract of the proof. We can clearly see our witness term in NUPRL notation: The witness term is given by the coalgebra `<F_dom_dom h, (F_M g) F_dom_op (T_arr F_dom_op h)>`.

```
λC,F,T,h,%.
let <g,%1> = %
in
    < <F_dom_dom h, (F_M g) F_dom_op (T_arr F_dom_op h)>
    , Ax
    , Ax
    , Ax>
```

Figure 5.17: Simplified Extract of `right_invertible_implies_unfold`

In the following chapter we further study the case when $h$ is an arrow in the category of sets, i.e. a (total) function.

[3] The diagram also shows that again we do not actually need the right-invertibility of $h$, but only the weaker condition that $Fh$ is right-invertible.

# Chapter 6

# When is a Function a Catamorphism?

In the previous chapter we formally proved sufficient conditions for when an arbitrary arrow is a catamorphism or an anamorphism. In this chapter we want to further study the special case when $h$ is an arrow in the category of sets (i.e. a function). The questions that we are trying to answer are still the same: Given an arrow $h$ of the appropriate type, is $h$ a catamorphism? If so, how can we construct an arrow $g$ such that $h = \mathsf{fold}\, g$?

## 6.1 A Non-Constructive Result

For the special case of the category $\mathcal{SET}$, with sets as objects and functions as arrows, J. Gibbons, G. Hutton, and T. Altenkirch [GHA01] proved the following necessary and sufficient condition for when an arrow is a catamorphism.

**Theorem 6.1.1 (Gibbons, Hutton, Altenkirch).** *Suppose $F : \mathcal{SET} \to \mathcal{SET}$ is a functor with an initial algebra $(\mu F, in)$, $A$ is a set, and $h : \mu F \to A$. Then*

$$(\exists g : FA \to A. \quad h = \mathsf{fold}\, g) \iff \ker(Fh) \subseteq \ker(h \cdot in).$$

Here $\ker f$, the *kernel* of a function $f : A \to B$, is defined as a binary relation on $A$ containing all pairs of elements in $A$ that are mapped to the same element in $B$.

**Definition 6.1.2 (Kernel).** Suppose $f : A \to B$. Then

$$\ker f = \{(a_1, a_2) \in A \times A \mid f(a_1) = f(a_2)\}$$

is the *kernel* of $f$.

Figure 6.1 shows the NUPRL abstraction `kernel`. The well-formedness theorem for it is proved in a single step by the AUTO tactic.

```
* ABS kernel
ker[A,B] f == {aa:A × A| f aa.1 = f aa.2}
```

Figure 6.1: Abstraction `kernel`

Theorem 6.1.1 gives an exhaustive answer to our first question: $h$ is a catamorphism if and only if $\ker(Fh) \subseteq \ker(h \cdot in)$. Unlike the universal property of $\mathsf{fold}$, this property only depends on $h$, so we do not have to know a function $g$ with $h = \mathsf{fold}\, g$ beforehand to verify it.

Unfortunately however, only the proof of the '⇒' direction of Theorem 6.1.1 is constructive.[1] The proof of the '⇐' direction that is given in [GHA01] uses the law of excluded middle several times: "However, our proofs are set-theoretic, and make essential use of classical logic and the Axiom of Choice; hence our results do not generalize to categories of constructive functions."

Thus it is possible that the '⇐' direction of Theorem 6.1.1 tells us a function $h$ can be written as $\mathsf{fold}\, g$ for some function $g$, but even though we know such a $g$ *exists*, we are still not able to *compute* it. What we would like to have is an algorithm to compute $g$ from $h$—or equivalently, a constructive proof of the '⇐' direction of Theorem 6.1.1.

## 6.2 A Necessary Condition

As said above, the '⇒' direction of Theorem 6.1.1 can be proved constructively. In this section we present a constructive proof (which is essentially the same as the proof of the '⇒' direction given in [GHA01]), together with a formalization of the theorem in NUPRL.

**Theorem 6.2.1.** *Suppose $F : \mathcal{SET} \to \mathcal{SET}$ is a functor with an initial algebra $(\mu F, in)$, $A$ is a set, and $h : \mu F \to A$. Then*

$$(\exists g : FA \to A.\quad h = \mathsf{fold}\, g) \implies \ker(Fh) \subseteq \ker(h \cdot in).$$

The corresponding NUPRL theorem `fold_implies_kernel_inclusion` is shown in Figure 6.2. The key to its proof is the observation that the existence of 'postfactors' implies the inclusion of kernels. We say $g : B \to C$ is a *postfactor* of $f : A \to B$ for $h : A \to C$ if and only if $h = g \cdot f$.

[1] It is no surprise that this direction has a constructive proof: The right side of the theorem is a subset relation, so it does not contain any computational information. In other words, there is nothing to construct.

```
* THM fold_implies_kernel_inclusion
∀F:Functor{i'}(large_category{i},large_category{i}).
∀I:{I:Algebra(F)| algebra_category(F)-initial(I)} .
∀h:F_dom_Arr.
(∃g:Algebra(F). h=fold[large_category{i},F,I](g) )
⇒ ker[F_O I_obj,large_category{i}_cod (F_M h)] (F_M h).2.2
    ⊆ ker[F_O I_obj,large_category{i}_cod (h F_dom_op I_arr)]
    (h F_dom_op I_arr).2.2
```

Figure 6.2: Theorem `fold_implies_kernel_inclusion`

**Lemma 6.2.2.** *Suppose that $f : A \to B$ and $h : A \to C$, where $A$, $B$, $C$ are sets. Then*

$$(\exists g : B \to C. \quad h = g \cdot f) \implies \ker f \subseteq \ker h.$$

*Proof.* Assume that $g : B \to C$ with $h = g \cdot f$. Then

$$\begin{array}{ll} & (a_1, a_2) \in \ker f \\ \iff & \quad \{ \text{ kernels } \} \\ & f(a_1) = f(a_2) \\ \implies & \quad \{ \text{ substitutivity } \} \\ & g(f(a_1)) = g(f(a_2)) \\ \iff & \quad \{ \ h = g \cdot f \ \} \\ & h(a_1) = h(a_2) \\ \iff & \quad \{ \text{ kernels } \} \\ & (a_1, a_2) \in \ker h. \end{array}$$

□

The NUPRL version of this lemma is shown in Figure 6.3. It is proved in about eight steps. However, we formalized arrows in the category of types not just as functions, but as triples $(A, B, f : A \to B)$ (see Chapter 4). Therefore we prove a second version of the lemma for arrows in the category of types. Even though this second lemma (see Figure 6.4) is just a 'lifted' version of `postfactor_implies_kernel_inclusion` and its proof relies on the first version of the lemma, the proof is about 36 steps long.

Using Lemma 6.2.2, the proof of Theorem 6.2.1 becomes quite simple.

```
* THM postfactor_implies_kernel_inclusion
∀A,B,C:𝕌.  ∀f:A ⟶ B. ∀h:A ⟶ C.
(∃g:B ⟶ C. h = g o f) ⇒ ker[A,B] f ⊆ ker[A,C] h
```

Figure 6.3: Theorem `postfactor_implies_kernel_inclusion`

```
* THM postfactor_implies_kernel_inclusion_cat
∀A,B,C:large_category{i}_Obj.  ∀f:Mor[large_category{i}](A,B).
∀h:Mor[large_category{i}](A,C).
(∃g:Mor[large_category{i}](B,C). h = g large_category{i}_op f)
⇒ ker[A,B] f.2.2 ⊆ ker[A,C] h.2.2
```

Figure 6.4: Theorem `postfactor_implies_kernel_inclusion_cat`

*Proof.*

$$
\begin{array}{ll}
 & \exists g : FA \to A. \quad h = \mathsf{fold}\, g \\
\Longleftrightarrow & \quad \{ \text{ universal property } \} \\
 & \exists g : FA \to A. \quad h \cdot in = g \cdot Fh \\
\Longrightarrow & \quad \{ \text{ Lemma 6.2.2 } \} \\
 & \ker(Fh) \subseteq \ker(h \cdot in).
\end{array}
$$

□

The formal proof in NUPRL requires about 70 steps. Most of those steps are needed to discharge the well-formedness goals that are created by the instantiation of the `postfactor_implies_kernel_inclusion_cat` lemma.

## 6.3 A Sufficient Condition

The proof that is given for the '⇐' direction of Theorem 6.1.1 in [GHA01] is not constructive. However, by analyzing the proof, we were able to identify additional conditions that allow us to prove the '⇐' direction constructively. But before we state these conditions, we have to deal with two problems that are not caused by the differences between classical and constructive logic, but by the differences between set theory and type theory.

So far we simply formalized sets as types in NUPRL. Up to this point this has not caused any problems. Types are also propositions in NUPRL, and we can prove the following equivalence (see Figure 6.5):

$$P \iff (\exists x : P. \quad True).$$

```
* THM prop_iff_exists
∀P:ℙ.  P ⟺ (∃x:P. True)
```

Figure 6.5: Theorem `prop_iff_exists`

This suggests formalizing $A \neq \emptyset$ as $A$ (which is equivalent to $(\exists x : A. \quad True)$), and formalizing $A = \emptyset$ as $\neg A$ (which is equivalent to $\neg(\exists x : A. \quad True)$). Now the problem is that there is no unique empty type. When two sets contain no elements, we can conclude that they are both equal to the empty set $\emptyset$, and therefore equal to each other. This conclusion is one step in the proof of the '$\Leftarrow$' direction. However, when two types contain no elements, they can still be different. Types are equal only when they have the same 'structure'; unlike equality of sets, equality of types is not extensional. The types *Void* and $\{x : \mathbb{Z}.\ x < x\}$, for example, are distinguished as types even though they are extensionally equal, i.e. neither is inhabited. Thus we will have to find a different proof that does not rely on extensional type equality.

The second problem is that $\ker(Fh) \subseteq \ker(h \cdot in)$ is not well-formed in NUPRL unless $\ker(Fh)$ is actually a subset of $\ker(h \cdot in)$. The subtype relation $A \subseteq B$ is defined as $(\forall x : A.\ x \in B)$. If provable, the inhabitant is always equivalent to $\lambda x.Ax$, where $Ax$ is a NUPRL constant that is the inhabitant of a membership goal. Therefore $x \in B$ is either true or not well-formed. So to prove that $\ker(Fh) \subseteq \ker(h \cdot in)$ is well-formed, we have to prove that for every $x \in \ker(Fh)$, $x \in \ker(h \cdot in)$ is well-formed—which means we have to prove it is true, which means we have to prove $\ker(Fh) \subseteq \ker(h \cdot in)$. But this subset relation is not true (and hence not provable) in general! If it was, we would not need it as an assumption to our proof.

This was not a problem when we stated the '$\Rightarrow$' direction of Theorem 6.1.1 since $\ker(Fh) \subseteq \ker(h \cdot in)$ was a conclusion then (hence something we could prove), but it clearly is a problem with the '$\Leftarrow$' direction. We cannot even solve this problem by making (reasonable) additional assumptions. No matter how we try to state that every element of $\ker(Fh)$ is an element of $\ker(h \cdot in)$, the well-formedness problem remains.[2] We finally decided to use NUPRL's FIAT tactic to prove the well-formedness of $\ker(Fh) \subseteq \ker(h \cdot in)$. Since we only use FIAT on a well-formedness subgoal however, this does not affect the correctness of the proof extract. If $\ker(Fh) \subseteq \ker(h \cdot in)$ is in fact true, we still get a valid algorithm out of the proof that computes a function

[2] For the same reason, there is no well-formedness theorem for the `subtype` abstraction in the standard NUPRL libraries.

$g$ with $h = \mathsf{fold}\, g$ given a function $h$. This being said, we state this section's main result.

**Theorem 6.3.1.** *Suppose $F : \mathcal{SET} \to \mathcal{SET}$ is a functor with an initial algebra $(\mu F, in)$, $A$ is a set such that we can decide whether $A$ is empty, and $h : \mu F \to A$. Furthermore, suppose for every $b \in FA$ we can decide whether $b = (Fh)(a)$ for some $a \in F(\mu F)$. Then*

$$(\exists g : FA \to A. \quad h = \mathit{fold}\, g) \Longleftarrow \ker(Fh) \subseteq \ker(h \cdot in).$$

Figure 6.6 shows a type-theoretic formalization of this theorem in NUPRL. The proof depends on two lemmata, namely that the inclusion of kernels implies the existence of postfactors, and that the existence of a function $h : \mu F \to A$ implies the existence of a function $g : FA \to A$. We prove the former lemma first.

```
* THM kernel_inclusion_implies_fold
∀F:Functor{i'}(large_category{i},large_category{i}).
∀I:{I:Algebra(F)| algebra_category(F)-initial(I)} .
∀A:large_category{i}_Obj.  ∀h:Mor[large_category{i}](I_obj,A).
Dec(A)
⇒ (∀b:F_O A. Dec(∃a:F_O I_obj.  b = (F_M h).2.2 a))
⇒ ((∃g:Algebra(F). h=fold[large_category{i},F,I](g) )
    ⇐ ker[F_O I_obj,large_category{i}_cod (F_M h)] (F_M h).2.2
    ⊆ ker[F_O I_obj,large_category{i}_cod (h F_dom_op I_arr)]
    (h F_dom_op I_arr).2.2)
```

Figure 6.6: Theorem `kernel_inclusion_implies_fold`

**Lemma 6.3.2.** *Suppose $f : A \to B$ and $h : A \to C$. Furthermore, suppose we can decide whether $C$ is empty, and for every $b \in B$ we can decide whether $b = f(a)$ for some $a \in A$. Then*

$$(\exists g : B \to C. \quad h = g \cdot f) \Longleftarrow (\ker f \subseteq \ker h \wedge B \to C \neq \emptyset).$$

*Proof.* Assume $\ker f \subseteq \ker h$ and $B \to C \neq \emptyset$.

If $C = \emptyset$, then $B = \emptyset$ since $B \to C \neq \emptyset$, and $A = \emptyset$ since $f : A \to B$. Therefore $f = h = id(\emptyset)$, and if we choose $g = id(\emptyset)$, clearly $g : B \to C$ and $h = g \cdot f$.

If $C \neq \emptyset$, let $c$ be an arbitrary element in $C$. Let *choice* : $\{b \in B \mid \exists a \in A.\ b = f(a)\} \to A$ be a function with $f(\mathit{choice}(b)) = b$ for all $b \in B$.[3] For $b \in B$ define

[3] To prove that such a function *choice* exists, we need the Axiom of Choice. Interestingly, this axiom has a proof (!) in NUPRL that is based simply on the representation of functions and of the quantifiers ∀ and ∃.

$g(b) \in C$ as follows: If $b = f(a)$ for some $a \in A$, then $g(b) = h(\mathit{choice}(b))$. Otherwise, $g(b) = c$.

Now let $a \in A$. Since $f(\mathit{choice}(f(a))) = f(a)$ by definition of *choice*, we have $(\mathit{choice}(f(a)), a) \in \ker f \subseteq \ker h$. Hence $g(f(a)) = h(\mathit{choice}(f(a))) = h(a)$, and therefore $h = g \cdot f$. □

To give a constructive proof that the inclusion of kernels implies the existence of postfactors, we made two additional assumptions compared to the statement of this lemma in [GHA01]: that we can decide whether the codomain of $h$ is empty, and that we can decide whether an element in the codomain of $f$ is in the image of $f$. The NUPRL theorem `kernel_inclusion_implies_postfactor` is shown in Figure 6.7. The formal proof is about 43 steps long; the well-formedness of $\ker f \subseteq \ker h$ is proved by the FIAT tactic.

```
* THM kernel_inclusion_implies_postfactor
∀A,B,C:𝕌.  ∀f:A ⟶ B. ∀h:A ⟶ C.
Dec(C)
⇒ (∀b:B. Dec(∃a:A. b = f a))
⇒ ((∃g:B ⟶ C. h = g o f) ⇐ ker[A,B] f ⊆ ker[A,C] h ∧ B ⟶ C)
```

Figure 6.7: Theorem `kernel_inclusion_implies_postfactor`

Figure 6.8 shows a 'lifted' version of the lemma for arrows in the category of sets. Despite the use of the original lemma in the proof of the lifted version, the proof is about 71 steps long.

```
* THM kernel_inclusion_implies_postfactor_cat
∀A,B,C:large_category{i}_Obj.  ∀f:Mor[large_category{i}](A,B).
∀h:Mor[large_category{i}](A,C).
Dec(C)
⇒ (∀b:B. Dec(∃a:A. b = f.2.2 a))
⇒ ((∃g:Mor[large_category{i}](B,C). h = g large_category{i}_op f)
    ⇐ ker[A,B] f.2.2 ⊆ ker[A,C] h.2.2
    ∧ Mor[large_category{i}](B,C))
```

Figure 6.8: Theorem `kernel_inclusion_implies_postfactor_cat`

The second lemma that we need to prove Theorem 6.3.1 is stated below.

**Lemma 6.3.3.** *Suppose $F : \mathcal{SET} \to \mathcal{SET}$ is a functor with an initial algebra $(\mu F, in)$, and $A$ is a set such that we can decide whether $A$ is empty. Then*

$$\mu F \to A \neq \emptyset \implies FA \to A \neq \emptyset.$$

$$\begin{tikzcd}
F(\mu F) \arrow[d, "in"'] & FA \arrow[l, dashed, bend right, "Fg"'] \arrow[d, dotted, "f"] \\
\mu F \arrow[r, "h"'] & A \arrow[l, dashed, bend right, "g"']
\end{tikzcd}$$

Figure 6.9: $\mu F \to A \neq \emptyset \implies FA \to A \neq \emptyset$.

*Proof.* If $A \neq \emptyset$, then trivially $FA \to A \neq \emptyset$.

If $A = \emptyset$, then the embedding $g : A \hookrightarrow \mu F$ is a function from $A$ to $\mu F$. Thus $Fg : FA \to F(\mu F)$ by the properties of functors. Hence $h \cdot in \cdot Fg : FA \to A$.

Therefore $FA \to A \neq \emptyset$ in either case. □

Figure 6.9 illustrates the situation: Given a function $h : \mu F \to A$, we can find a function $f : FA \to A$. The functions $g : A \to \mu F$ and $Fg : FA \to F(\mu F)$ are needed only in the case $A = \emptyset$. If $A \neq \emptyset$, they may not exist—but we can construct a function $f : FA \to A$ directly then. Note that the lemma is not true for arbitrary categories. The proof of the lemma given above is different from the proof that was given in [GHA01][4], but the theorem `hom_fun_implies_algebra_fun` (which is shown in Figure 6.10) is proved along the same lines. The formal proof is about 49 steps long.

```
* THM hom_fun_implies_algebra_fun
∀F:Functor{i'}(large_category{i},large_category{i}).
∀I:{I:Algebra(F)| algebra_category(F)-initial(I)} .
∀A:large_category{i}_Obj.
Dec(A)
⇒ Mor[large_category{i}](I_obj,A)
⇒ Mor[large_category{i}](F_O A,A)
```

Figure 6.10: Theorem `hom_fun_implies_algebra_fun`

We are now ready to prove Theorem 6.3.1.

[4] mainly because of the problem mentioned earlier that there is no unique empty type, but also because we avoided using the classical version of the contrapositive, $(\neg p \implies \neg q) \implies (q \implies p)$, which is not true constructively

*Proof.*

$$
\begin{array}{cl}
& \ker(Fh) \subseteq \ker(h \cdot in) \\
\iff & \quad \{ \text{ Lemma 6.3.3}, h : \mu F \to A \ \} \\
& \ker(Fh) \subseteq \ker(h \cdot in) \wedge FA \to A \neq \emptyset \\
\implies & \quad \{ \text{ Lemma 6.3.2 } \} \\
& \exists g : FA \to A. \quad h \cdot in = g \cdot Fh \\
\iff & \quad \{ \text{ universal property } \} \\
& \exists g : FA \to A. \quad h = \mathsf{fold}\, g.
\end{array}
$$

□

Although this proof is relatively simple, a number of well-formedness goals have to be dealt with in the formal proof of the `kernel_inclusion_implies_fold` theorem. Therefore the formal proof is about 115 steps long.

Clearly we can decide whether an element in $FA$ is in the image of $Fh$ when $Fh$ is surjective (onto). We will show that $Fh$ is surjective if $h$ is. Therefore every surjective function that satisfies the condition of kernel inclusion is a catamorphism if we can decide whether its codomain $A$ is empty.[5] We could relatively easily prove this as a corollary to Theorem 6.3.1. Closer inspection of the proof of Theorem 6.3.1 however shows that when $h$ is surjective, we do not need the additional assumption that we can decide whether $A$ is empty.

**Theorem 6.3.4.** *Suppose $F : \mathcal{SET} \to \mathcal{SET}$ is a functor with an initial algebra $(\mu F, in)$, and $h : \mu F \to A$ is surjective. Then*

$$(\exists g : FA \to A. \quad h = \mathsf{fold}\, g) \Longleftarrow \ker(Fh) \subseteq \ker(h \cdot in).$$

We first prove that a function is surjective if and only if it is right-invertible in $\mathcal{SET}$.

**Lemma 6.3.5.** *Suppose $f : A \to B$. Then*

$$f \text{ is surjective} \iff f \text{ is right-invertible in } \mathcal{SET}.$$

*Proof.* For the '⇒' direction, suppose $f$ is surjective. Then there exists a function $g : B \to A$ such that $f(g(b)) = b$ for all $b \in B$ (by the Axiom of Choice). Hence $f \cdot g = id(B)$, so $f$ is right-invertible.

For the '⇐' direction, suppose $f$ is right-invertible in $\mathcal{SET}$. Then $f \cdot g = id(B)$ for some function $g : B \to A$. Now let $b \in B$. Then $f(g(b)) = (f \cdot g)(b) = (id(B))(b) = b$. Therefore $f$ is surjective. □

[5] Note that every injective (one-to-one) function is a catamorphism by Theorem 5.2.2.

Figure 6.11 shows a formalization of the lemma in NUPRL. The formal proof is about 33 steps long and makes use of the `ax_choice` lemma from the `FUN_1` library.

```
* THM surjective_iff_right_invertible
∀A,B:𝕌.  ∀f:A ⟶ B.
    Surj(A;B;f) ⟺ right-invertible[large_category{i}](<A, B, f>)
```

Figure 6.11: Theorem `surjective_iff_right_invertible`

We also state and prove a 'lifted' version of the lemma for arrows in the category of types. This lifted version is shown in Figure 6.12. Lifting the lemma requires about 11 proof steps; of course we use the lemma `surjective_iff_right_invertible` in the proof of `surjective_iff_right_invertible_cat`.

```
* THM surjective_iff_right_invertible_cat
∀f:large_category{i}_Arr
    Surj(large_category{i}_dom f;large_category{i}_cod f;f.2.2)
    ⟺ right-invertible[large_category{i}](f)
```

Figure 6.12: Theorem `surjective_iff_right_invertible_cat`

We now prove a lemma similar to Lemma 6.3.2, but for surjective functions.

**Lemma 6.3.6.** *Suppose $f : A \to B$ is surjective, and suppose $h : A \to C$. Then*

$$(\exists g : B \to C. \quad h = g \cdot f) \Longleftarrow \ker f \subseteq \ker h.$$

*Proof.* Assume $\ker f \subseteq \ker h$.

Let $choice : B \to A$ be a function with $f(choice(b)) = b$ for all $b \in B$ (such a function $choice$ exists by the Axiom of Choice since $f$ is surjective). Define $g : B \to C$ by $g(b) = h(choice(b))$ for every $b \in B$.

Now $h = g \cdot f$ by construction of $g$: Let $a \in A$. Since $f(choice(f(a))) = f(a)$ by definition of $choice$, $(choice(f(a)), a) \in \ker f \subseteq \ker h$. Therefore $g(f(a)) = h(choice(f(a))) = h(a)$. □

Figure 6.13 shows a formalization of this lemma in NUPRL. The formal proof requires about 14 steps. It is similar to the proof of `kernel_inclusion_implies_postfactor`, but slightly simpler—just like the informal proof. The well-formedness of $\ker f \subseteq \ker h$ is again proved by the FIAT tactic.

As for the `kernel_inclusion_implies_postfactor` lemma above, we prove a 'lifted' version of this lemma for arrows in the category of types. The lifted version is shown

```
* THM kernel_inclusion_implies_postfactor_surjective
∀A,B,C:𝕌.  ∀f:A ⟶ B. ∀h:A ⟶ C.
Surj(A;B;f) ⇒ ((∃g:B ⟶ C. h = g o f) ⇐ ker[A,B] f ⊆ ker[A,C] h)
```

Figure 6.13: Theorem `kernel_inclusion_implies_postfactor_surjective`

```
* THM kernel_inclusion_implies_postfactor_surjective_cat
∀A,B,C:large_category{i}_Obj.  ∀f:Mor[large_category{i}](A,B).
∀h:Mor[large_category{i}](A,C).
Surj(A;B;f.2.2)
⇒ ((∃g:Mor[large_category{i}](B,C). h = g large_category{i}_op f)
    ⇐ ker[A,B] f.2.2 ⊆ ker[A,C] h.2.2)
```

Figure 6.14: Theorem `kernel_inclusion_implies_postfactor_surjective_cat`

in Figure 6.14. Its proof is similar to the proof of the lifted lemma for functions with a decidable image (see Figure 6.8) and requires about 47 steps.

Using the two Lemmata 6.3.5 and 6.3.6, we can now prove Theorem 6.3.4.

*Proof.* We first show that $Fh : F(\mu F) \to FA$ is surjective. Since $h$ is surjective, $h$ is right-invertible by Lemma 6.3.5. Let $g : A \to \mu F$ be a function with $h \cdot g = id(A)$. Then

$$
\begin{array}{ll}
 & Fh \cdot Fg \\
= & \quad \{ \text{ functors } \} \\
 & F(h \cdot g) \\
= & \quad \{ \text{ assumption } \} \\
 & F(id(A)) \\
= & \quad \{ \text{ functors } \} \\
 & id(FA).
\end{array}
$$

Hence $Fh$ is right-invertible, and therefore surjective (again by Lemma 6.3.5). Now

$$
\begin{array}{ll}
 & \ker(Fh) \subseteq \ker(h \cdot in) \\
\Longrightarrow & \quad \{ \text{ Lemma 6.3.6 } \} \\
 & \exists g : FA \to A. \quad h \cdot in = g \cdot Fh \\
\Longleftrightarrow & \quad \{ \text{ universal property } \} \\
 & \exists g : FA \to A. \quad h = \mathsf{fold}\, g
\end{array}
$$

completes the proof. $\square$

See Figure 6.15 for a statement of this theorem in NUPRL. We use the lemma `kernel_inclusion_implies_postfactor_surjective_cat` to prove the existence of $g$, and `surjective_iff_right_invertible_cat` to prove that $Fh$ is surjective. Altogether the formal proof requires about 145 steps. FIAT is used to prove the well-formedness of $\ker(Fh) \subseteq \ker(h \cdot in)$.

```
* THM kernel_inclusion_implies_fold_surjective
∀F:Functor{i'}(large_category{i},large_category{i}).
∀I:{I:Algebra(F)| algebra_category(F)-initial(I)} .
∀A:large_category{i}_Obj.  ∀h:Mor[large_category{i}](I_obj,A).
Surj(I_obj;A;h.2.2)
⇒ ((∃g:Algebra(F). h=fold[large_category{i},F,I](g) )
    ⇐ ker[F_O I_obj,large_category{i}_cod (F_M h)] (F_M h).2.2
    ⊆ ker[F_O I_obj,large_category{i}_cod (h F_dom_op I_arr)]
    (h F_dom_op I_arr).2.2)
```

Figure 6.15: Theorem `kernel_inclusion_implies_fold_surjective`

We now have two simple conditions for when a constructive function $h$ that satisfies the condition of kernel inclusion is a catamorphism: $h$ is a catamorphism if the image of $Fh$ is decidable and we can decide whether the codomain of $h$ is empty, and $h$ is a catamorphism if $h$ is surjective.

## 6.4 Computing fold$^{-1}$: A Simple Example

In the previous section we gave a constructive proof for when a function $h$ is a catamorphism. Embedded in the proof is an algorithm to compute a function $g$ such that $h = \mathsf{fold}\, g$. Figure 6.16 shows a simplified version of the extract of the proof of theorem `kernel_inclusion_implies_fold` (the original extract was about six pages long). We can see how the function $g$ in the witness term $(A, g)$ is constructed by instantiating the `kernel_inclusion_implies_postfactor_cat` theorem. The argument `%@0` is a proof that we can decide whether $A$ is empty, `%13` proves that for every $b \in FA$ we can decide whether $b = (Fh)(a)$ for some $a \in F(\mu F)$, and `%14` finally proves the kernel inclusion.

In this section we apply the `kernel_inclusion_implies_fold` theorem to the `all` function defined in Chapter 1 to write this function as a catamorphism. Recall the definition of `all`:

```
all p L = and (map p L)
```

```
λF,I,A,h,%@0,%13,%14.
(let <g,%24> =
    (ext{kernel_inclusion_implies_postfactor_cat}{i:l}
    (F_O I_obj)
    (large_category{i}_cod (F_M h))
    (large_category{i}_cod (h F_dom_op I_arr))
    (F_M h)
    (h F_dom_op I_arr)
    (%@0)
    (λb.%13 b)
    <λ.Ax, (ext{hom_fun_implies_algebra_fun}{i:l} F I A %@0 h) >)
in
    <<A, g>, Ax, Ax, Ax>
```

Figure 6.16: Simplified Extract of Theorem `kernel_inclusion_implies_fold`

Here $L$ is a list over some type $T$, and $p : T \to \mathbb{B}$. Figure 6.17 shows a formalization of the `and` function in NUPRL. The corresponding well-formedness theorem shows that `list_and_2` is a boolean value if $L$ is a list of booleans. It is proved in about six steps by structural induction on $L$. The second function that we need to define `all`, the `map` function, is already defined in the `LIST_1` library.

```
* ABS list_and_2
∧b(L) == (letrec recfun(L) =
    case L of [] => tt | h::t => h ∧b recfun t esac ) L
```

Figure 6.17: Abstraction `list_and_2`

Before we can prove that `all` is a catamorphism, we have to show that $List(T)$ is the object of an initial algebra. Consider the functor $\mathcal{L}_T : \mathcal{SET} \to \mathcal{SET}$ again, defined by $\mathcal{L}_T(A) = \mathbf{1} + (T \times A)$ and $\mathcal{L}_T(f) = id(\mathbf{1}) + (id(T) \times f)$. Figure 6.18 shows the NUPRL abstraction defining this functor. The corresponding well-formedness theorem is shown in Figure 6.19. Verifying that `list_functor` is in fact a functor (from $\mathcal{SET}$ to $\mathcal{SET}$) takes about 54 proof steps.

This functor has an initial algebra $(\mu\mathcal{L}_T, in) = (List(T), nil + cons)$, which is defined formally in Figure 6.20. The corresponding well-formedness theorem shows that `list_functor_initial_algebra` is in fact an algebra. It is proved in about six steps.

To verify that this is an *initial* algebra, we have to show that for every other algebra $(A, f)$ there exists a unique homomorphism $h$ from $(List(T), nil + cons)$ to $(A, f)$. Since $h$ is a homomorphism, i.e. $h \cdot (nil + cons) = (id(\mathbf{1}) + (id(T) \times h)) \cdot f$, we have

```
* ABS list_functor
ListF{i}(T) ==
<large_category{i}
, large_category{i}
, λA.Unit + T × A
, λf.<Unit + T × large_category{i}_dom f
    , Unit + T × large_category{i}_cod f
    , λx.case x of inl(y) => x |
    inr(z) => let <t,a> = z in inr <t, f.2.2 a> >>
```

Figure 6.18: Abstraction `list_functor`

```
* THM list_functor_wf
∀T:𝕌.  ListF{i}(T) ∈ Functor{i'}(large_category{i},large_category{i})
```

Figure 6.19: Theorem `list_functor_wf`

```
* ABS list_functor_initial_algebra
InitialAlgebra(ListF(T)) ==
<T List
, Unit + T × T List
, T List
, λx.case x of inl(y) => [] | inr(z) => let <h,t> = z in h::t>
```

Figure 6.20: Abstraction `list_functor_initial_algebra`

$h([]) = f(\mathit{inl}\ \cdot)$ and $h(u :: v) = f(\mathit{inr}\ (u, h(v)))$ for all $u \in T$, $v \in \mathit{List}(T)$. Both that $h$ is a homomorphism and that $h$ is unique can be proved by structural induction on lists. The corresponding NUPRL theorem `list_functor_initial_algebra_is_initial` is shown in Figure 6.21. The proof of this theorem is quite technical and complicated by our inevitable formalization of algebras, homomorphisms and arrows in the category of types as tuples. With approximately 211 proof steps, it is the longest proof in this thesis. About 140 of those steps are required only to show the uniqueness of $h$.

```
* THM list_functor_initial_algebra_is_initial
∀T:𝕌.  algebra_category(ListF{i}(T))-initial(InitialAlgebra(ListF(T)))
```

Figure 6.21: Theorem `list_functor_initial_algebra_is_initial`

Using the `kernel_inclusion_implies_fold` theorem, we can now prove that the composition of `map` and `and` is a catamorphism. We do, however, need one more assumption: that we can decide for all $b \in \mathbb{B}$ whether there exists a list $L \in \mathit{List}(T)$

with $b = and(map(p; L))$.[6] If $b = \text{true}$, then $b = and([]) = and(map(p; []))$. Therefore it is sufficient if we can decide whether $p(t) = \text{false}$ for some $t \in T$: If so, then $\text{false} = and(map(p; t :: []))$. Otherwise $and(map(p; L)) = \text{true}$ for all $L \in List(T)$; this can be proved by structural induction on $L$. Figure 6.22 shows the NUPRL theorem `list_and_2_map_is_fold`.

```
* THM list_and_2_map_is_fold
∀T:𝕌.  ∀p:T ⟶ 𝔹.
Dec(∃t:T. p t = ff)
⇒ (∃g:Algebra(ListF{i}(T))
    <T List, 𝔹, λL.∧b(map(p;L))> =
    fold[large_category{i},ListF{i}(T),InitialAlgebra(ListF(T))](g) )
```

Figure 6.22: Theorem `list_and_2_map_is_fold`

The extract of this theorem (see Figure 6.23) is a function that takes three arguments: a type $T$, a predicate $p : T \to \mathbb{B}$, and a proof `%` that we can decide whether $p(t) = \text{false}$ for some $t \in T$. The `kernel_inclusion_implies_fold` theorem is used to create the witness algebra $(A, g)$. The by far longest expression in the extract ('`λb.case b of` ...') is just a proof that we can decide whether some element in $\mathcal{L}_T(\mathbb{B}) = \mathbf{1} + (T \times \mathbb{B})$ is in the image of $\mathcal{L}_T(and(map(p; \cdot)))$.

We can further unfold the extracts of the `kernel_inclusion_implies_fold` lemma and of other lemmatas that were used in its proof. Eventually, we obtain the actual function $g$ with $and(map(p; \cdot)) = \mathsf{fold}\, g$. This function (with a few simplifications made by hand) is shown in Figure 6.24. It is a triple with its first and second component being its domain and codomain, respectively. The `if-then-else` statement is used to determine whether $b \in \mathbf{1}+(T\times\mathbb{B})$ is in the image of $\mathcal{L}_T(and(map(p; \cdot)))$. Three cases need to be differentiated: $b = inl\ \cdot$, $b = inr\ (y1, \text{true})$, and $b = inr\ (y1, \text{false})$. The latter can only occur if $p(t) = \text{false}$ for some $t \in T$; whether such a $t$ exists is determined by the value of `%`. If $b$ is in the image of $\mathcal{L}_T(and(map(p; \cdot)))$, the `then` part is used to apply $and(map(p; \cdot)) \cdot (nil + cons)$ to an element $z \in \mathbf{1} + (T \times List(T))$ with $(\mathcal{L}_T(and(map(p; \cdot))))(z) = b$. Otherwise, true is returned in the `else` part. Using $and(map(p; [])) = \text{true}$ and $and(map(p; t :: [])) = \text{false}$, we could simplify the `then` part even further.

[6] We also need to be able to decide whether $\mathbb{B}$ is empty. Since $\text{true} \in \mathbb{B}$, $\mathbb{B}$ is not empty.

```
λT,p,%.
ext{kernel_inclusion_implies_fold}{i:l}
ListF{i}(T)
InitialAlgebra(ListF(T))
𝔹
<T List, 𝔹, λL.∧b(map(p;L))>
(inl tt )
(λb.case b
    of inl(x) => inl <inl · , Ax>
    | inr(y) => let <y1,y2> = y
    in
    case y2
    of inl(x) => inl <inr <y1, []> , Ax>
    | inr(y) => case %
    of inl(%2) => let <t,%3> = %2
    in
    inl <inr <y1, t::[]> , Ax>
    | inr(%3) => inr (λ%.let <a,%4> = %
    in
    case a
    of inl(x) => Ax
    | inr(y) => let <y2,y3> = y
    in
    any (rec-case(y3)
    of [] => λ%.Ax
    | u::v => %.λ%4.any (%3 <u, Ax>) Ax))
)
(λx.Ax)
```

Figure 6.23: Simplified Extract of Theorem `list_and_2_map_is_fold`

## 6.5 Two Counterexamples

We claimed that the proof of Theorem 6.1.1 given in [GHA01] is not constructive, and that we need essentially two additional assumptions to turn it into a constructive proof: Firstly, that we can decide whether the set $A$ is empty, and secondly, that we can decide whether an element in $FA$ is in the image of $Fh$.

However, decidability of the image of $Fh$ is not a necessary condition for a constructive function $h$ to be a catamorphism. We prove this by giving a function with a non-decidable image that can still be written as a fold. We then prove that $\ker(Fh) \subseteq \ker(h \cdot in)$ is not a sufficient constructive condition for a function to be a catamorphism

```
< (ListF{i}(T)_0 𝔹)
, 𝔹
, λb.if case b
    of inl(x) => tt
    | inr(y) => let <y1,y2> = y
    in
    case y2
    of inl(x) => tt
    | inr(y) => case %
    of inl(%2) => tt
    | inr(%3) => ff
then (<T List, 𝔹, λL.∧𝑏(map(p;L))> ListF{i}(T)_dom_op
    InitialAlgebra(ListF(T))_arr).2.2
    (case b
    of inl(x) => <inl ·, Ax>
    | inr(y) => let <y1,y2> = y
    in
    case y2
    of inl(x) => <inr <y1, []> , Ax>
    | inr(y) => case %
    of inl(%2) => let <t,%3> = %2
    in
    <inr <y1, t::[]>, Ax>
    | inr(%3) => any Ax.1)
else tt
fi >
```

Figure 6.24: A Function $g$ with $and(map(p;\cdot)) = \mathsf{fold}\, g$

by giving a computable function $h$ that satisfies this condition, and a proof that no function $g$ with $h = \mathsf{fold}\, g$ is computable.[7]

Recall the functor $\mathcal{L}_X$ defined in Chapter 5 which has $(List(X), nil + cons)$ as an initial algebra. Let $TM$ denote the set of all Turing machines, and let $H$ denote the set of all Turing machines that halt after a finite number of steps (when applied to the empty input). Clearly $H \subseteq TM$, and $H$ is not decidable [Tur36]. Consider the embedding $h_0 : List(H) \hookrightarrow List(TM)$. Now $\operatorname{img} \mathcal{L}_H(h_0) = \mathbf{1} + H \times List(H)$ is not decidable in $\mathbf{1} + H \times List(TM) = \mathcal{L}_H(List(TM))$. Therefore the construction given in the proof of Theorem 6.3.1 fails. However, we can still find a function $g : \mathbf{1} + H \times List(TM) \to List(TM)$ such that $h_0 \cdot in = g \cdot \mathcal{L}_H(h_0)$: Simply map $(\cdot)$ to

[7] Of course a (possibly non-computable) function $g$ with $h = \mathsf{fold}\, g$ must exist by Theorem 6.1.1.

$$\begin{tikzcd}
\mathbf{1} + (H \times \mathit{List}(H)) \arrow[r, hook, "\mathcal{L}_H(h_0)"] \arrow[d, "\mathit{nil}+\mathit{cons}"'] & \mathbf{1} + (H \times \mathit{List}(\mathit{TM})) \arrow[d, "g=\mathit{nil}+\mathit{cons}"] \\
\mathit{List}(H) \arrow[r, hook, "h_0"'] & \mathit{List}(\mathit{TM})
\end{tikzcd}$$

Figure 6.25: A Catamorphism $h$ where $\mathrm{img}(Fh)$ is not Decidable

$[]$, and a pair $(M, L)$ (where $M$ is a Turing machine in $H$, and $L$ is a list of Turing machines) to $M :: L$. That is, $g = \mathit{nil} + \mathit{cons}$ on $\mathbf{1} + H \times \mathit{List}(\mathit{TM})$ (see Figure 6.25). Now $h_0 \cdot \mathit{in} = g \cdot \mathcal{L}_H(h_0)$ is immediate. Therefore $h_0 = \mathsf{fold}\, g$ by the universal property of $\mathsf{fold}$, so $h_0$ is a catamorphism.

Our second counterexample, which proves that not every function $h$ of the appropriate type with $\ker(Fh) \subseteq \ker(h \cdot \mathit{in})$ is a catamorphism when we restrict ourselves to constructive (i.e. computable) functions, needs to be more elaborate. Consider the functor $\mathcal{L}_{\mathbb{N}} : \mathcal{SET} \to \mathcal{SET}$ with its initial algebra $(\mathit{List}(\mathbb{N}), \mathit{nil} + \mathit{cons})$. Assume $\mathit{TM} = \{M_1, M_2, M_3, \ldots\}$.[8] Define a function $h_1 : \mathit{List}(\mathbb{N}) \to \mathbb{N} + \mathit{TM}$ as follows: If $L \in \mathit{List}(\mathbb{N})$ is of even length, then $h(L)$ is defined as the sum of the elements in $L$. That is, $h(L) = \sum L$ in this case, where $\sum : \mathit{List}(\mathbb{N}) \to \mathbb{N}$ is defined by the two equations

$$\begin{aligned}
\textstyle\sum [] &= 0, \\
\textstyle\sum (n :: L) &= n + \textstyle\sum L.
\end{aligned}$$

If $L$ is of odd length, then $L$ has at least one element, so say $L = n :: L'$. In this case, let $h_1(L) = h_1(n :: L')$ be the $(n+1)$. Turing machine in $\mathit{TM}$ that halts after exactly $\sum L'$ steps (when applied to the empty input). We can compute $h_1(n :: L')$ as follows: Simulate $M_1$ for (at most) $\sum L'$ steps to check whether it halts after exactly $\sum L'$ steps. Then simulate $M_2$ for (at most) $\sum L'$ steps, then $M_3$, and so on, until we have found the $(n+1)$. Turing machine in $\mathit{TM}$ that halts after exactly $\sum L'$ steps. This algorithm is guaranteed to terminate since for every number of steps, there exist infinitely many Turing machines in $\mathit{TM}$ that terminate after exactly that many steps. Hence $h_1$ is computable.

The function $\mathcal{L}_{\mathbb{N}}(h_1) : \mathbf{1} + (\mathbb{N} \times \mathit{List}(\mathbb{N})) \to \mathbf{1} + (\mathbb{N} \times (\mathbb{N} + \mathit{TM}))$ maps a pair $(n, L)$, where $n$ is a natural number and $L$ is a list of natural numbers, to the pair

[8] The set of all Turing machines is countable, and we can enumerate this set in some (computable) way—e.g. based on the length of the representation of Turing machines.

$$\begin{tikzcd}
\mathbf{1} + (\mathbb{N} \times \mathit{List}(\mathbb{N})) \arrow[r, "\mathcal{L}_{\mathbb{N}}(h_1)"] \arrow[d, "\mathit{nil}+\mathit{cons}"'] & \mathbf{1} + (\mathbb{N} \times (\mathbb{N} + \mathit{TM})) \arrow[d, "g"] \\
\mathit{List}(\mathbb{N}) \arrow[r, "h_1"'] & \mathbb{N} + \mathit{TM}
\end{tikzcd}$$

Figure 6.26: A Function $h$ with $\ker(Fh) \subseteq \ker(h \cdot in)$ that is not a Catamorphism

$(n, h_1(L)) \in \mathbb{N} \times (\mathbb{N} + \mathit{TM})$. To prove $\ker(\mathcal{L}_{\mathbb{N}}(h_1)) \subseteq \ker(h_1 \cdot in)$, we have to verify that $h_1(L) = h_1(M)$ implies $h_1(n :: L) = h_1(n :: M)$ for all $L$, $M \in \mathit{List}(\mathbb{N})$ and $n \in \mathbb{N}$. So assume $h_1(L) = h_1(M)$. If $L$ is of even length, then $h_1(L) = \sum L$. Hence $M$ is also of even length and $\sum L = h_1(L) = h_1(M) = \sum M$. Since $h_1(n :: L)$ is the $(n+1)$. Turing machine in $\mathit{TM}$ that halts after exactly $\sum L$ steps, and $h_1(n :: M)$ is the $(n+1)$. Turing machine in $\mathit{TM}$ that halts after exactly $\sum M$ steps, we have $h_1(n :: L) = h_1(n :: M)$. If $L$ is of odd length, then $L = l :: L'$ for some $l \in \mathbb{N}$ and $L' \in \mathit{List}(\mathbb{N})$. In this case $h_1(L)$ is the $(l+1)$. Turing machine that halts after exactly $\sum L'$ steps. So $M$ is also of odd length, say $M = m :: M'$ for some $m \in \mathbb{N}$ and $M' \in \mathit{List}(\mathbb{N})$, and $h_1(M)$ is the $(m+1)$. Turing machine that halts after exactly $\sum M'$ steps. Since $h_1(L) = h_1(M)$, we have $\sum L' = \sum M'$, and $M_i \neq M_j$ for $i \neq j$ implies $l = m$. Therefore $h_1(n :: L) = \sum(n :: L) = n + \sum L = n + l + \sum L' = n + m + \sum M' = n + \sum M = \sum(n :: M) = h_1(n :: M)$.

Figure 6.26 shows the sets and functions involved in this counterexample. Since a Turing machine $M \in \mathit{TM}$ halts if and only if it is the $(n+1)$. Turing machine to halt after $s$ steps for some $n$, $s \in \mathbb{N}$ (we assume $0 \in \mathbb{N}$), we have $\mathrm{img}(h_1) = \mathbb{N} + H$ and $\mathrm{img}(\mathcal{L}_{\mathbb{N}}(h_1)) = \mathbf{1} + (\mathbb{N} \times (\mathbb{N} + H))$.

Now assume $h_1 = \mathsf{fold}\, g$ for some function $g : \mathbf{1} + (\mathbb{N} \times (\mathbb{N} + \mathit{TM})) \to \mathbb{N} + \mathit{TM}$. We use reduction to the halting problem to prove that $g$ is not computable. Assume $g$ is computable. Consider any pair $(k, M) \in \mathbb{N} \times H$, and assume $M$ is the $(n+1)$. Turing machine to halt after $s$ steps. Since $h_1 \cdot in = g \cdot \mathcal{L}_{\mathbb{N}}(h_1)$, we have $g(k, M) = k + n + s$. Therefore $g(k, M) \geq s$, so any Turing machine $M \in H$ halts after at most $g(0, M)$ steps. Thus it is enough to run a Turing machine $T \in \mathit{TM}$ for $g(0, T)$ steps to find out if $T$ halts (on the empty input); if it does not halt until then, it will never halt. This gives us a decision procedure for the halting problem—a contradiction, therefore $g$ is not computable.

# Chapter 7

# Bird's Fusion Transformation

Many algorithms can be specified as the composition of a function that constructs an intermediate data structure from the given input, and another function that traverses the intermediate data structure to extract the requested information. A simple example was given in Chapter 1.

Bird's fusion theorem [Bir95] proves that if the first function is an anamorphism and the second function is a catamorphism, these two functions can be combined into a single function, thereby eliminating the intermediate data structure constructed by the anamorphism.

This chapter presents a formalization of the fusion theorem for the special case where the underlying data structure is the type of binary trees. The formalization presented here is partially based on a formalization of Bird's fusion transformation in PVS by N. Shankar [Sha96].

## 7.1 Binary Trees

A *binary tree* (over some type $T$) is a type of data structure in which each element is attached to zero or two elements directly beneath it. We use the following inductive definition after [CLRS01].

**Definition 7.1.1 (Binary Trees).** Suppose $T$ is a type.

- A *leaf* is a binary tree over $T$.
- If $t \in T$ and $B_1, B_2$ are binary trees over $T$, then $\mathit{node}(t, B_1, B_2)$ is a binary tree over $T$.

*BinTree*$(T)$ is the type of all binary trees over $T$.

According to this definition, leafs do not carry information (i.e. elements from $T$). All information is stored in the nodes, and in the structure of the tree itself.

```
* ABS binary_tree
BinTree(T) == rec(t.Unit + T × t × t)
```

Figure 7.1: Abstraction `binary_tree`

The NUPRL abstraction defining binary trees is shown in Figure 7.1. Due to the use of the disjoint product type $+$, a binary tree in NUPRL now is equal to either *inl* $\cdot$ or *inr* $<t, B_1, B_2>$, where $t \in T$ and $B_1$, $B_2$ are binary trees. We define `leaf` as an abbreviation for *inl* $\cdot$, and `node(t,B_1,B_2)` as an abbreviation for *inr* $<t, B_1, B_2>$, as shown in Figure 7.2. The fact that `leaf` and `node(t,B_1,B_2)` are binary trees is captured by the two well-formedness theorems shown in Figure 7.3. The theorems are proved in two steps each.

```
* ABS leaf
leaf == inl ·

* ABS node
node(t,b1,b2) == inr <t, b1, b2>
```

Figure 7.2: Abstractions `leaf` and `node`

```
* THM leaf_wf
∀T:𝕌.  leaf ∈ BinTree(T)

* THM node_wf
∀T:𝕌.  ∀t:T. ∀B1,B2:BinTree(T). node(t,B1,B2) ∈ BinTree(T)
```

Figure 7.3: Well-formedness theorems for `leaf` and `node`

**Example 7.1.2.** $node(0, leaf, node(1, leaf, leaf))$ represents a binary tree with value 0 at its root node, an empty left branch, and a single node with value 1 in its right branch. Figure 7.4 shows a graphical representation of this tree.

See Figure 7.5 for a theorem stating that $node(0, leaf, node(1, leaf, leaf))$ is in fact a binary tree (over $\mathbb{N}$). The theorem is proved in a single step by NUPRL's AUTO tactic.

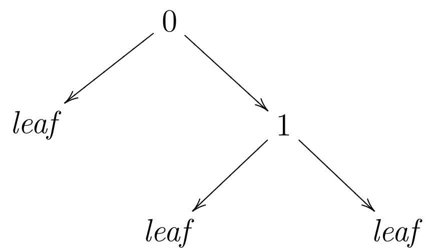



Figure 7.4: Example: A binary tree

```
* THM binary_tree_example
node(0; leaf; node(1; leaf; leaf)) ∈ BinTree(ℕ)
```

Figure 7.5: Theorem `binary_tree_example`

## 7.2 The *reduce* Operator

Suppose $T$ and $R$ are types, $c \in R$ and $g : T \times R \times R \to R$. We want to define a function $f : \mathit{BinTree}(T) \to R$ by the following recursion over binary trees:

$$\begin{aligned} f(\mathit{leaf}) &= c \\ f(\mathit{node}(t, B_1, B_2)) &= g(t, f(B_1), f(B_2)) \end{aligned}$$

The *reduce* operator is defined such that $f = \mathit{reduce}(c; g)$. Note that every function $f$ that can be written as $\mathit{reduce}(c; g)$ for some $c$ and $g$ is a catamorphism on binary trees.

**Definition 7.2.1 (reduce).** Suppose $T$ and $R$ are types, $c \in R$ and $g : T \times R \times R \to R$. Define $\mathit{reduce}(c; g) : \mathit{BinTree}(T) \to R$ recursively by

$$\mathit{reduce}(c; g)(B) = \begin{cases} c & \text{if } B = \mathit{leaf} \\ g(t, \mathit{reduce}(c; g)(B_1), \mathit{reduce}(c; g)(B_2)) & \text{if } B = \mathit{node}(t, B_1, B_2) \end{cases}$$

for all $B \in \mathit{BinTree}(T)$.

The corresponding abstraction `treereduce` is shown in Figure 7.6. We use a curried function $g : T \to R \to R \to R$ in the `treereduce` abstraction instead of a function with domain $T \times R \times R$ and codomain $R$. Avoiding the cartesian product (and consequently, tuples as function arguments) simplifies the NUPRL notation.

Since *reduce* is defined recursively, we have to verify that this recursion always terminates to make sure that $\mathit{reduce}(c; g)$ is well-defined, i.e. that $\mathit{reduce}(c; g)(B)$ is in $R$ for all binary trees $B$.

```
* ABS treereduce
reduce(c;g)(B) ==
(letrec recfun(B) = case B
of inl(x) => c
| inr(y) => let t,B1,B2 = y in g t (recfun B1) (recfun B2) )
B
```

Figure 7.6: Abstraction `treereduce`

**Lemma 7.2.2.** *Suppose $T$ and $R$ are types, $c \in R$ and $g : T \times R \times R \to R$. Then*

$$reduce(c; g)(B) \in R$$

*for all $B \in BinTree(T)$.*

*Proof.* Let $B \in BinTree(T)$. We use structural induction on $B$.

Base case ($B = leaf$): $reduce(c; g)(B) = c \in R$.

Inductive step ($B = node(t, B_1, B_2)$): By the induction hypothesis, $reduce(c; g)(B_1) \in R$ and $reduce(c; g)(B_2) \in R$. Therefore

$$reduce(c; g)(B) = g(t, reduce(c; g)(B_1), reduce(c; g)(B_2)) \in R.$$

□

The proof of the formal theorem `treereduce_wf`, which is shown in Figure 7.7, proceeds along the same lines. The RecElimination tactic is used for structural induction on $B$. The base case is then proved by the Auto tactic after we unfold the definition of `treereduce`. The induction hypothesis is used to prove the inductive step. Altogether the proof is about nine steps long.

```
* THM treereduce_wf
∀T,R:𝕌.  ∀c:R. ∀g:T ⟶ R ⟶ R ⟶ R. ∀B:BinTree(T).
reduce(c;g)(B) ∈ R
```

Figure 7.7: Theorem `treereduce_wf`

**Example 7.2.3.** The *height* of a binary tree (over an arbitrary type $T$) can be defined recursively. The height of a leaf is 0, and the height of a node is one more than the maximum of the heights of the node's left and right subtree:

$$\begin{aligned} height(leaf) &= 0, \\ height(node(t, B_1, B_2)) &= 1 + \max(height(B_1), height(B_2)). \end{aligned}$$

```
* ABS treeheight
|B| ==
(letrec recfun(B) = case B
of inl(x) => 0
| inr(y) => let t,B1,B2 = y in 1 + imax(recfun B1;recfun B2) )
B
```

Figure 7.8: Abstraction `treeheight`

See Figure 7.8 for a formal definition.

Clearly $height(B) \in \mathbb{N}$ for all binary trees $B$; this fact is proved by the theorem `treeheight_wf` shown in Figure 7.9. Again the RecElimination tactic is used for structural induction on $B$ in the proof of this theorem. The formal proof is about 27 steps long, mainly because we have to overcome a few technical difficulties caused by the use of $\mathbb{N}$ and $\mathbb{Z}$.

```
* THM treeheight_wf
∀T:𝕌.  ∀B:BinTree(T). |B| ∈ ℕ
```

Figure 7.9: Theorem `treeheight_wf`

Alternatively, *height* can be defined in terms of *reduce*. Define $g : T \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by $g(t, m, n) = 1 + \max(m, n)$. Then $height(B) = reduce(0; g)(B)$ for all binary trees $B$, as shown in Figure 7.10. The proof of this theorem is about ten steps long and uses both the RecElimination tactic and the `treeheight_wf` lemma, as well as a few other lemmata.

```
* THM treereduce_example
∀T:𝕌.  ∀B:BinTree(T). |B| = reduce(0;λt,m,n.1 + imax(m;n))(B)
```

Figure 7.10: Theorem `treereduce_example`

## 7.3 The *unfold* Operator

The *reduce* operator extracts some information from a binary tree. It provides a general pattern to define catamorphisms on binary trees. Now suppose $S$ is a type, and we want to define an operator *unfold* that *constructs* a binary tree from some input $x \in S$ as follows. First, a given predicate $p$ is applied to $x$. If $p(x)$ is true, we apply a function $f$ to $x$ that computes a node value $a$ and two new input values $y$ and $z$. *unfold* is then recursively applied to $y$ and $z$ to compute the left and right subtree of the node. If $p(x)$ is false, *unfold* simply returns *leaf*.

However, there is a problem with this 'definition'. If $y$ and $z$ are allowed to be arbitrary input values, this recursion is not guaranteed to terminate: Assume $p(x)$ is true for every input $x$, and consider the function $f : S \to S \times S \times S$, defined by $f(x) = (x, x, x)$. Then

$$\begin{aligned} \mathit{unfold}(p; f)(x) &= \mathit{node}(x, \mathit{unfold}(p; f)(x), \mathit{unfold}(p; f)(x)) \\ &= \mathit{node}(x, \\ &\qquad \mathit{node}(x, \mathit{unfold}(p; f)(x), \mathit{unfold}(p; f)(x)), \\ &\qquad \mathit{node}(x, \mathit{unfold}(p; f)(x), \mathit{unfold}(p; f)(x))) \\ &= \ldots \end{aligned}$$

To guarantee that the recursion terminates, we require $y$ and $z$ to be 'smaller' than $x$, where the 'size' of an input is just a natural number.[1]

**Definition 7.3.1 (Smaller).** Suppose $S$ is a type, $\mathit{size} : S \to \mathbb{N}$, and $x \in S$. Then we define

$$\mathit{Smaller}(S, \mathit{size}, x) = \{y \in S \mid \mathit{size}(y) < \mathit{size}(x)\}$$

to be the type of all elements in $S$ with a size less than the size of $x$.

The formal definition of `Smaller` is shown in Figure 7.11. The well-formedness theorem `smaller_wf` proves that `Smaller(S,size,x)` is a type if $S$ is a type, $\mathit{size} : S \to \mathbb{N}$, and $x \in S$. It is proved in a single step by the Auto tactic.

```
* ABS smaller
Smaller(S,size,x) == {y:S| size y < size x}
```

Figure 7.11: Abstraction `Smaller`

Now we are ready to define the type of functions that we allow as a parameter to *unfold*. Note that to compute $\mathit{unfold}(p; f)(x)$, we only need to evaluate $f(x)$ when $p(x)$ is true. Therefore the domain of $f$ does not need to be $S$, but it can be restricted to the subtype $\{x \in S \mid p(x) = \text{true}\}$.

[1] As pointed out by N. Shankar [Sha96], any well-founded ordering could be used here instead of the less-than relation on natural numbers.

**Definition 7.3.2 (WellFnd).** Suppose $S$ and $T$ are types, $p : S \to \mathbb{B}$, and $size : S \to \mathbb{N}$. Then we define

$$
\begin{aligned}
&\mathit{WellFnd}(S, p, \mathit{size}, T) = \\
&\quad \{f : \{x \in S \mid p(x) = \text{true}\} \to T \times S \times S \mid \\
&\qquad \forall x \in \{x \in S \mid p(x) = \text{true}\} : \\
&\qquad\quad f(x) \in T \times \mathit{Smaller}(S, \mathit{size}, x) \times \mathit{Smaller}(S, \mathit{size}, x)\}.
\end{aligned}
$$

In NUPRL, the dependent function type can be used to define *WellFnd* more elegantly: The codomain does not have to be a single type, but it can depend on the function argument $x$. Thus given $x$, we can require $f(x)$ to be in $T \times \mathit{Smaller}(S, \mathit{size}, x) \times \mathit{Smaller}(S, \mathit{size}, x)$. Figure 7.12 shows the corresponding abstraction `treewellfnd`.

```
* ABS treewellfnd
WellFnd(S,p,size,T) ==
x:{x:S| p[x] = tt} → (T × Smaller(S,size,x) × Smaller(S,size,x))
```

Figure 7.12: Abstraction `treewellfnd`

The well-formedness theorem for `treewellfnd` simply states that this is a type if $S$ and $T$ are types, $p : S \to \mathbb{B}$, and $size : S \to \mathbb{N}$. It is proved in a single step by NUPRL's AUTO tactic. Using the type *WellFnd* of *'well-founded'* functions, we can now precisely define *unfold*.

**Definition 7.3.3 (unfold).** Suppose $S$ and $T$ are types, $p : S \to \mathbb{B}$, $size : S \to \mathbb{N}$, and $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$. Define $\mathit{unfold}(p; f) : S \to \mathit{BinTree}(T)$ recursively by

$$
\mathit{unfold}(p; f)(x) = \begin{cases} \mathit{node}(a, \mathit{unfold}(p; f)(y), \mathit{unfold}(p; f)(z)) & \text{if } p(x) \text{ is true} \\ \mathit{leaf} & \text{if } p(x) \text{ is false} \end{cases}
$$

for all $x \in S$, where $f(x) = (a, y, z)$.

See Figure 7.13 for the definition of `treeunfold` in NUPRL.

The restrictions imposed on $f$ allow us to prove that *unfold* is well-defined, i.e. that the recursion always terminates.

**Lemma 7.3.4.** *Suppose $S$ and $T$ are types, $p : S \to \mathbb{B}$, $size : S \to \mathbb{N}$, and $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$. Then*

$$
\mathit{unfold}(p; f)(x) \in \mathit{BinTree}(T)
$$

*for all $x \in S$.*

```
* ABS treeunfold
unfold(p;f)(x) ==
(letrec recfun(x) = if p[x]
then let a,y,z = (f x) in node(a; (recfun y); (recfun z))
else leaf
fi )
x
```

Figure 7.13: Abstraction `treeunfold`

*Proof.* Let $x \in S$. We show $\mathit{unfold}(p; f)(x) \in \mathit{BinTree}(T)$ by complete induction on $\mathit{size}(x)$. Assume $\mathit{unfold}(p; f)(y) \in \mathit{BinTree}(T)$ for all $y \in S$ with $\mathit{size}(y) < \mathit{size}(x)$.

Case 1: Assume $p(x)$ is false. Then $\mathit{unfold}(p; f)(x) = \mathit{leaf} \in \mathit{BinTree}(T)$.

Case 2: Assume $p(x)$ is true. Let $f(x) = (a, y, z)$. Then $y, z \in \mathit{Smaller}(S, \mathit{size}, x)$ since $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$. Hence $\mathit{size}(y) < \mathit{size}(x)$ and $\mathit{size}(z) < \mathit{size}(x)$. Thus $\mathit{unfold}(p; f)(y) \in \mathit{BinTree}(T)$ and $\mathit{unfold}(p; f)(z) \in \mathit{BinTree}(T)$ by the induction hypothesis. Therefore

$$\mathit{unfold}(p; f)(x) = \mathit{node}(a, \mathit{unfold}(p; f)(y), \mathit{unfold}(p; f)(z)) \in \mathit{BinTree}(T).$$

□

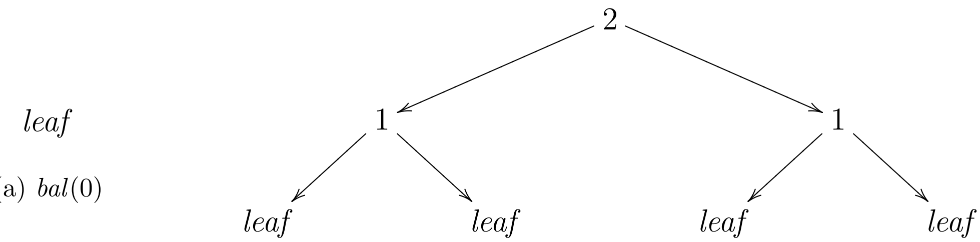
In NUPRL we state this lemma as a well-formedness theorem for `treeunfold`. This well-formedness theorem is shown in Figure 7.14. The formal proof uses NUPRL's INVIMAGEIND tactic in combination with the COMPNATIND tactic for complete induction on the size of $x$. The IFTHENELSE tactic is then used for the case split on $p(x)$. The proof is about 15 steps long.

```
* THM treeunfold_wf
∀S:𝕌.  ∀p:S → 𝔹.  ∀size:S → ℕ.  ∀T:𝕌.  ∀f:WellFnd(S,p,size,T).
∀x:S. unfold(p;f)(x) ∈ BinTree(T)
```

Figure 7.14: Theorem `treeunfold_wf`

The *unfold* operator, just like *reduce*, can be used to specify a number of algorithms. We give a simple example below, and a more elaborate example in the following chapter.

**Example 7.3.5.** We say a binary tree $B$ is *balanced* if and only if every leaf in $B$ has the same height. Consider a function $\mathit{bal} : \mathbb{N} \to \mathit{BinTree}(\mathbb{N})$ that, given a natural number $n$, creates a balanced binary tree of height $n$ in which every node is labelled




(a) $bal(0)$

(b) $bal(2)$

Figure 7.15: Example: *bal*

with its height (i.e. the root node is labelled with $n$, the two nodes directly beneath it are labelled with $n-1$, and so on). See Figure 7.15 for two examples.

More precisely, let $bal : \mathbb{N} \to BinTree(\mathbb{N})$ be defined inductively by

$$\begin{aligned} bal(0) &= leaf, \\ bal(n+1) &= node(n+1, f(n), f(n)). \end{aligned}$$

The NUPRL abstraction defining *bal* is shown in Figure 7.16. The well-formedness theorem `create_balanced_wf` proves that `create_balanced(n)` is in $BinTree(\mathbb{N})$ for every $n \in \mathbb{N}$. We use the NATIND tactic in the proof of `create_balanced_wf` for mathematical induction on $n$. The proof is about six steps long.

```
* ABS create_balanced
create_balanced(n) ==
(letrec recfun(n) = if (n =z 0)
then leaf
else node(n; (recfun (n - 1)); (recfun (n - 1)))
fi )
n
```

Figure 7.16: Abstraction `create_balanced`

Now define $p : \mathbb{N} \to \mathbb{B}$ by $p(n) \iff (n \neq 0)$, and define $g : \mathbb{N} \setminus \{0\} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ by $g(n) = (n, n-1, n-1)$. Then $bal(n) = unfold(p; g)(n)$ for all $n \in \mathbb{N}$, as proved by the theorem `treeunfold_example` shown in Figure 7.17. The proof uses NUPRL's NATIND tactic for mathematical induction on $n$. It is about 92 steps long, mainly because several well-formedness goals need to be verified.

```
* THM treeunfold_example
∀n:ℕ.  create_balanced(n) =
unfold((λn.¬b(n =z 0)); (λn.<n, n - 1, n - 1>); n)
```

Figure 7.17: Theorem `treeunfold_example`

## 7.4 The *fun* Operator

The composition of *unfold* and *reduce* can be used to specify a large number of algorithms, e.g. the QUICKSORT algorithm (see Chapter 8 for details). However, *unfold* first constructs a binary tree, and *reduce* then consumes the tree. Bird's fusion transformation allows us to replace *reduce* · *unfold* with a single operator *fun* (defined below) that does not construct an intermediate tree. This is an instance of *deforestation* [Dav87, Wad88, GJS93], a program optimization technique that fuses adjacent phases to eliminate the intermediate data structures.

**Definition 7.4.1 (fun).** Suppose $S, T, R$ are types, $p : S \to \mathbb{B}$, $size : S \to \mathbb{N}$, and $f \in WellFnd(S, p, size, T)$. Furthermore, suppose $c \in R$ and $g : T \times R \times R \to R$. Define $fun(p; f; c; g) : S \to R$ by

$$fun(p; f; c; g)(x) = \begin{cases} g(a, fun(p; f; c; g)(y), fun(p; f; c; g)(z)) & \text{if } p(x) \text{ is true} \\ c & \text{if } p(x) \text{ is false} \end{cases}$$

for all $x \in S$, where $f(x) = (a, y, z)$.

Figure 7.18 shows the corresponding NUPRL abstraction `treefun`. Again we avoid tuples as function arguments by using a curried function $g$.

```
* ABS treefun
fun(p;f;c;g)(x) ==
(letrec recfun(x) = if p[x]
then let a,y,z = (f x) in g a (recfun y) (recfun z)
else c
fi )
x
```

Figure 7.18: Abstraction `treefun`

The operator *fun*, like *reduce* and *unfold* before, is defined recursively. Therefore we need to verify that it is well-defined, i.e. that the recursion terminates for every input $x \in S$.

**Lemma 7.4.2.** *Suppose $S, T, R$ are types, $p : S \to \mathbb{B}$, $size : S \to \mathbb{N}$, and $f \in WellFnd(S, p, size, T)$. Furthermore, suppose $c \in R$ and $g : T \times R \times R \to R$. Then*

$$fun(p; f; c; g)(x) \in R$$

*for all $x \in S$.*

*Proof.* Let $x \in S$. We show $fun(p; f; c; g)(x) \in R$ by complete induction on $size(x)$. Assume $fun(p; f; c; g)(y) \in R$ for all $y \in S$ with $size(y) < size(x)$.

Case 1: Assume $p(x)$ is false. Then $fun(p; f; c; g)(x) = c \in R$.

Case 2: Assume $p(x)$ is true. Let $f(x) = (a, y, z)$. Then $y, z \in Smaller(S, size, x)$ since $f \in WellFnd(S, p, size, T)$. Hence $size(y) < size(x)$ and $size(z) < size(x)$. Thus $fun(p; f; c; g)(y) \in R$ and $fun(p; f; c; g)(z) \in R$ by the induction hypothesis. Therefore

$$fun(p; f; c; g)(x) = g(a, fun(p; f; c; g)(y), fun(p; f; c; g)(z)) \in R.$$

□

The formal well-formedness theorem is shown in Figure 7.19. Its proof is about eleven steps long and uses the InvImageInd tactic in combination with CompNatInd for complete induction on the size of $x$.

```
* THM treefun_wf
∀S:𝕌.  ∀p:S → 𝔹.  ∀size:S → ℕ.  ∀T:𝕌.  ∀f:WellFnd(S,p,size,T).
∀R:𝕌.  ∀c:R. ∀g:T → R → R → R. ∀x:S.
fun(p;f;c;g)(x) ∈ R
```

Figure 7.19: Theorem `treefun_wf`

## 7.5 Bird's Fusion Theorem for Binary Trees

As mentioned before, we want to replace *reduce* · *unfold* with *fun* to eliminate the intermediate tree. In this section we prove that *reduce* · *unfold* and *fun* are equivalent, in the sense that they compute the same function.

**Theorem 7.5.1 (Bird's Fusion Theorem for Binary Trees).** *Suppose $S, T, R$ are types, $p : S \to \mathbb{B}$, $size : S \to \mathbb{N}$, and $f \in WellFnd(S, p, size, T)$. Furthermore, suppose $c \in R$ and $g : T \times R \times R \to R$. Then*

$$(reduce(c; g) \cdot unfold(p; f))(x) = fun(p; f; c; g)(x)$$

*for all $x \in S$.*

*Proof.* Let $x \in S$. We show $(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(x) = \mathit{fun}(p;f;c;g)(x)$ by complete induction on $\mathit{size}(x)$. Assume $(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(y) = \mathit{fun}(p;f;c;g)(y)$ for all $y \in S$ with $\mathit{size}(y) < \mathit{size}(x)$.

Case 1: Assume $p(x)$ is false. Then

$$\begin{aligned}(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(x) &= \mathit{reduce}(c;g)(\mathit{unfold}(p;f)(x))\\ &= \mathit{reduce}(c;g)(\mathit{leaf})\\ &= c\\ &= \mathit{fun}(p;f;c;g)(x).\end{aligned}$$

Case 2: Assume $p(x)$ is true. Let $f(x) = (a,y,z)$. Then $y,z \in \mathit{Smaller}(S,\mathit{size},x)$ since $f \in \mathit{WellFnd}(S,p,\mathit{size},T)$. Hence $\mathit{size}(y) < \mathit{size}(x)$ and $\mathit{size}(z) < \mathit{size}(x)$. Thus $(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(y) = \mathit{fun}(p;f;c;g)(y)$ and $(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(z) = \mathit{fun}(p;f;c;g)(z)$ by the induction hypothesis. Therefore

$$\begin{aligned}&(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(x)\\ &\quad = \mathit{reduce}(c;g)(\mathit{unfold}(p;f)(x))\\ &\quad = \mathit{reduce}(c;g)(\mathit{node}(a,\mathit{unfold}(p;f)(y),\mathit{unfold}(p;f)(z)))\\ &\quad = g(a,\mathit{reduce}(c;g)(\mathit{unfold}(p;f)(y)),\mathit{reduce}(c;g)(\mathit{unfold}(p;f)(z)))\\ &\quad = g(a,(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(y),(\mathit{reduce}(c;g)\cdot \mathit{unfold}(p;f))(z))\\ &\quad = g(a,\mathit{fun}(p;f;c;g)(y),\mathit{fun}(p;f;c;g)(z))\\ &\quad = \mathit{fun}(p;f;c;g)(x)\end{aligned}$$

as required. □

Figure 7.20 shows the formal `fusion` theorem. The proof uses the usual combination of the tactics InvImageInd and CompNatInd for complete induction on the size of $x$; it is about 27 steps long.

```
* THM fusion
∀S:𝕌.  ∀p:S → 𝔹.  ∀size:S → ℕ.  ∀T:𝕌.  ∀f:WellFnd(S,p,size,T).
∀Range:𝕌.  ∀c:Range.  ∀g:T → Range → Range → Range.  ∀x:S.
reduce(c;g)(unfold(p;f)(x)) = fun(p;f;c;g)(x)
```

Figure 7.20: Theorem `fusion`

In the following chapter we apply the fusion transformation to the Quicksort algorithm.

# Chapter 8

# Example: Quicksort

The QUICKSORT algorithm was first published by C.A.R. Hoare [Hoa61] in 1961. It is "one of the fastest, the best known, the most generalized, ... and the most widely used algorithms for sorting an array of numbers" [ES95]. Both R. Bird [Bir95] and N. Shankar [Sha96] chose it as an example to apply the fusion transformation to.

Despite its speed, QUICKSORT is a relatively simple algorithm. It can be described as follows.

1. If the list is empty, there is nothing to do.
2. Otherwise pick an element from the list to be the 'partition element'.
3. Divide the other elements into those less than or equal to the partition element, and those greater than the partition element.
4. Arrange the elements in the list such that the order is the elements below the partition element, the partition element itself, and the elements above the partition element.
5. Recursively invoke QUICKSORT on the smaller elements.
6. Recursively invoke QUICKSORT on the larger elements.

As we can see from this description, QUICKSORT can be used for any type on which an order relation $\leq$ is defined.[1]

[1] Note that even when $\leq$ is not an order relation, we can still formally apply QUICKSORT. In fact, we will prove that QUICKSORT returns a permutation of its input when $\leq$ is an arbitrary relation on the type of the list elements. However, we will need to put certain constraints on $\leq$ to prove that the list returned by QUICKSORT is ordered.

## 8.1 Quicksort in Nuprl

Figure 8.1 shows an implementation of the QUICKSORT algorithm in NUPRL. We define `quicksort` as a recursive function that takes a relation $\leq$ and a list $L$ as arguments and returns a list (NUPRL's built-in data type `list` is used here). If $L$ is the empty list, denoted as [], then the empty list is returned. Otherwise the head of $L$ is picked as the partition element. Then `quicksort` is invoked recursively on a list of all elements in the tail of $L$ that are smaller than or equal to ('`below`') the head of $L$, and on a list of all elements in the tail of $L$ that are larger than ('`above`') the head of $L$. Both lists are generated by the `filter` function: `filter(p;L)` returns a list with those elements in $L$ that satisfy the predicate $p$. Finally `append` (`@`) and `cons` (`::`) are used to concatenate the two lists and the partition element in the proper order.

```
* ABS quicksort
quicksort(≤,L) ==
(letrec recfun(L) = case L of
[] => []
a::y => recfun filter((λb.b below(≤) a);y)
@ (a::(recfun filter((λb.b above(≤) a);y)))
esac )
L
```

Figure 8.1: Abstraction `quicksort`

The `quicksort` function is defined recursively. We prove that it is well-defined by complete induction on the length of the input list $L$.

**Lemma 8.1.1.** *Suppose $T$ is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$\mathit{quicksort}(\leq, L) \in \mathit{List}(T)$$

*for all $L \in \mathit{List}(T)$.*

We first prove another lemma, namely that the list returned by $\mathit{filter}(p; L)$ is at most as long as $L$.

**Lemma 8.1.2.** *Suppose $T$ is a type, and $f : T \to \mathbb{B}$. Then*

$$|\mathit{filter}(f, L)| \leq |L|$$

*for all $L \in \mathit{List}(T)$.*

The `filter` abstraction is part of the `LIST_3` library, as is the abstraction defining `list_length`.[2] Here we define *filter* as follows.

**Definition 8.1.3 (filter).** Suppose $T$ is a type, $f : T \to \mathbb{B}$, and $L \in \mathit{List}(T)$. Define $\mathit{filter}(f; L) \in \mathit{List}(T)$ recursively by

$$\mathit{filter}(f;L) = \begin{cases} [] & \text{if } L = [] \\ \mathit{filter}(f;t) & \text{if } L = h :: t \text{ and } f(h) \text{ is false} \\ h :: \mathit{filter}(f;t) & \text{if } L = h :: t \text{ and } f(h) \text{ is true} \end{cases}.$$

With this definition we can easily prove Lemma 8.1.2.

*Proof.* The proof is by structural induction on $L$.

Base case ($L = []$): $|\mathit{filter}(f;L)| = |[]| = |L|$.

Inductive step ($L = h :: t$): By the induction hypothesis, $|\mathit{filter}(f;t)| \leq |t|$. If $f(h) = \text{true}$,

$$|\mathit{filter}(f;L)| = |h :: \mathit{filter}(f;t)| = 1 + |\mathit{filter}(f;t)| \leq 1 + |t| = |L|.$$

If $f(h) = \text{false}$,

$$|\mathit{filter}(f;L)| = |\mathit{filter}(f;t)| \leq |t| = |L| - 1 < |L|.$$

□

Figure 8.2 shows the corresponding NUPRL theorem `list_length_filter`. The proof of the formal theorem uses the LISTIND tactic for structural induction on $L$, and the IFTHENELSECASES tactic for the case split on $f(h)$. The proof is about six steps long; most of the work is done by NUPRL's AUTO tactic.

```
* THM list_length_filter
∀T:𝕌.  ∀f:T → 𝔹.  ∀L:T List.  |·| filter(f;L) ≤ |·| L
```

Figure 8.2: Theorem `list_length_filter`

Given a type $T$ and a relation $\leq\, : T \times T \to \mathbb{B}$, we define $b$ *below*$(\leq)$ $a$ as $b \leq a$, and $b$ *above*$(\leq)$ $a$ as $\neg(b$ *below*$(\leq)$ $a)$ for $a, b \in T$. The corresponding NUPRL abstractions `below` and `above` are shown in Figure 8.3.

The well-formedness theorems `below_wf` and `above_wf` prove that $b$ *below*$(\leq)$ $a$ and $b$ *above*$(\leq)$ $a$ are in $\mathbb{B}$ if $T$ is a type, $\leq\, : T \times T \to \mathbb{B}$, and $a, b \in T$. They are proved in a single step each. Now we are ready to prove Lemma 8.1.1.

[2]The abstraction defining `length` in NUPRL, however, is part of the `LIST_1` library.

```
* ABS below
b below(≤) a == b ≤ a

* ABS above
b above(≤) a == ¬_b b below(≤) a
```

Figure 8.3: Abstractions `below` and `above`

*Proof.* By complete induction on the length of $L$. Assume $\mathit{quicksort}(\leq, M) \in \mathit{List}(T)$ for all $M \in \mathit{List}(T)$ with $|M| < |L|$.

Case 1: Assume $L = []$. Then $\mathit{quicksort}(\leq, L) = [] \in \mathit{List}(T)$.

Case 2: Assume $L = h :: t$, where $h \in T$ and $t \in \mathit{List}(T)$. By Lemma 8.1.2, $|\mathit{filter}(b\ \mathit{below}(\leq)\ h; t)| \leq |t| < |L|$ and $|\mathit{filter}(b\ \mathit{above}(\leq)\ h; t)| \leq |t| < |L|$. Thus

$$\mathit{quicksort}(\leq, \mathit{filter}(b\ \mathit{below}(\leq)\ h; t)) \in \mathit{List}(T)$$

and

$$\mathit{quicksort}(\leq, \mathit{filter}(b\ \mathit{above}(\leq)\ h; t)) \in \mathit{List}(T)$$

by the induction hypothesis. Therefore

$$\begin{aligned}
&\mathit{quicksort}(\leq, L) \\
&\quad = \mathit{quicksort}(\leq, \mathit{filter}(b\ \mathit{below}(\leq)\ h; t)) \\
&\qquad\quad @\ (h :: \mathit{quicksort}(\leq, \mathit{filter}(b\ \mathit{above}(\leq)\ h; t))) \\
&\quad \in \mathit{List}(T).
\end{aligned}$$

□

The NUPRL theorem `quicksort_wf` is shown in Figure 8.4. Note the use of a curried function $\leq : T \to T \to \mathbb{B}$ to avoid tuples as function arguments. The formal proof uses the LISTLENIND tactic for complete induction on the length of the list $L$. Then CASES is used to do a case split on $L = []$ and $L = h :: t$. The case $L = []$ is proved by an invocation of the LISTIND tactic, because even though we know that $L$ is equal to $[]$, we cannot substitute $[]$ for $L$ in the proof goal $\mathit{quicksort}(\leq, L) \in \mathit{List}(T)$ without creating unprovable well-formedness goals. For the same reason, we cannot simply substitute $h :: t$ for $L$ in the other case. We circumvent this problem by eliminating $L$ from all hypotheses first (by substituting $h :: t$ for $L$, or by moving them to the conclusion), and by decomposing the declaration of $L$ as a list then. With 26 steps altogether, the proof is relatively short, but surprisingly tricky.

```
* THM quicksort_wf
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀L:T List.  quicksort(≤,L) ∈ T List
```

Figure 8.4: Theorem `quicksort_wf`

## 8.2 Quicksort by Fusion

If we compare our implementation of QUICKSORT (Figure 8.1) to the `treefun` operator (Figure 7.18) defined in the previous chapter, it is almost obvious that QUICKSORT can be written as `treefun`, and hence—by the fusion theorem—that QUICKSORT is equal to the composition of an anamorphism and a catamorphism. In this section we make a few necessary definitions before we finally prove this equality.

Using a binary tree, we can split QUICKSORT into two phases. The first phase constructs an ordered binary tree that contains the same elements as the input list $L$ as follows: The partition element becomes the tree's root value. The left subtree and the right subtree are recursively constructed from a list of those elements in the tail of $L$ that are below the partition element, and from a list of those elements in $L$ that are above the partition element. The empty list $[]$ simply becomes a leaf.

The second phase flattens the ordered binary tree into an ordered list by an in-order search: First the left subtree is flattened, then the root value is inserted at the end of the list, then the right subtree is flattened.

Flattening a binary tree is a catamorphism that can easily be defined in terms of *reduce*.

**Definition 8.2.1 (flatten).** Suppose $T$ is a type. Let $g : T \times \mathit{List}(T) \times \mathit{List}(T) \to \mathit{List}(T)$ be defined by $g(a, x, y) = x@(a :: y)$. Define $\mathit{flatten} : \mathit{BinTree}(T) \to \mathit{List}(T)$ by

$$\mathit{flatten}(B) = \mathit{reduce}([]; g)(B).$$

The formal definition of `flatten` is shown in Figure 8.5. The well-formedness theorem `flatten_wf` proves that $\mathit{flatten}(B)$ is a list over $T$ for every type $T$ and every $B \in \mathit{BinTree}(T)$. It is proved in two steps by instantiating the `treereduce_wf` lemma.

Defining the first phase of QUICKSORT in terms of *unfold* requires a little more effort. Firstly we define a simple predicate $\mathit{is\_cons} : \mathit{List}(T) \to \mathbb{B}$ such that $\mathit{is\_cons}(L)$ is true if and only if $L = h :: t$ for some $h \in T$, $t \in \mathit{List}(T)$. The abstraction `is_cons` is shown in Figure 8.6.

The well-formedness theorem `is_cons_wf` states that $\mathit{is\_cons} : \mathit{List}(T) \to \mathbb{B}$ for every type $T$. It is proved in a single step by the AUTO tactic. We also prove two

```
* ABS flatten
flatten(B) == reduce([];λa,x,y.x @ (a::y))(B)

* THM flatten_wf
∀T:𝕌.  ∀B:BinTree(T). flatten(B) ∈ T List
```

Figure 8.5: Abstraction `flatten` and Theorem `flatten_wf`

```
* ABS is_cons
is_cons == λL.case L of [] => ff | h::t => tt esac
```

Figure 8.6: Abstraction `is_cons`

useful lemmata, namely that $\mathit{is\_cons}([])$ is false and that $\mathit{is\_cons}(h :: t)$ is true (see Figure 8.7). The lemmata are proved in a single step each by unfolding the definition of `is_cons` and applying the Auto tactic afterwards.

```
* THM is_cons_of_nil
is_cons [] = ff

* THM is_cons_of_cons
∀T:𝕌.  ∀u:T. ∀v:T List.  is_cons (u::v) = tt
```

Figure 8.7: Theorems `is_cons_of_nil` and `is_cons_of_cons`

We then define a function $\mathit{unjoin}(\leq) : \{L \in \mathit{List}(T) \mid \mathit{is\_cons}(L)\} \to T \times \mathit{List}(T) \times \mathit{List}(T)$ that maps a non-empty list $L$ to the triple that has $\mathit{hd}(L)$ as its first component, the list of all elements in $\mathit{tl}(L)$ that are below $\mathit{hd}(L)$ as its second element, and finally the list of all elements in $\mathit{tl}(L)$ that are above $\mathit{hd}(L)$ as its third element.

**Definition 8.2.2 (unjoin).** Suppose $T$ is a type and $\leq\, : T \times T \to \mathbb{B}$. Define $\mathit{unjoin}(\leq) : \{L \in \mathit{List}(T) \mid \mathit{is\_cons}(L)\} \to T \times \mathit{List}(T) \times \mathit{List}(T)$ by

$$\begin{aligned}&\mathit{unjoin}(\leq)(L) = \\ &\quad (\mathit{hd}(L), \mathit{filter}(\cdot\ \mathit{below}(\leq)\ \mathit{hd}(L); \mathit{tl}(L)), \mathit{filter}(\cdot\ \mathit{above}(\leq)\ \mathit{hd}(L); \mathit{tl}(L)))\end{aligned}$$

for all $L \in \mathit{List}(T)$ with $\mathit{is\_cons}(L) = \mathrm{true}$.

The Nuprl abstraction `unjoin` is shown in Figure 8.8. We want to use *unjoin* as an argument to the *unfold* operator defined in Chapter 7, so we have to verify that *unjoin* is a 'well-founded' function.

```
* ABS unjoin
unjoin(≤) ==
λx.<hd(x),
    filter((λb.b below(≤) hd(x));tl(x)),
    filter((λb.b above(≤) hd(x));tl(x))>
```

Figure 8.8: Abstraction `unjoin`

**Lemma 8.2.3.** *Suppose $T$ is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$unjoin(\leq) \in WellFnd(List(T), is\_cons, |\cdot|, T).$$

*Proof.* Clearly $unjoin(\leq) : \{L \in List(T) \mid is\_cons(L)\} \to T \times List(T) \times List(T)$. We have to verify

$$filter(\cdot\ below(\leq)\ hd(L); tl(L)) \in Smaller(List(T), |\cdot|, L)$$

and

$$filter(\cdot\ above(\leq)\ hd(L); tl(L)) \in Smaller(List(T), |\cdot|, L)$$

for all $L$ in $List(T)$ with $is\_cons(L) = \text{true}$.

Both statements follow from Lemma 8.1.2 in combination with $|tl(L)| = |L| - 1 < |L|$. □

We prove this lemma as a well-formedness theorem `unjoin_wf` in NUPRL (see Figure 8.9). The formal proof is about 24 steps long. It uses a number of lemmata, including `list_length_filter` and `length_tl`. The latter proves $|tl(L)| = |L| - 1$. It can be found in the `LIST_1` library. The final proof step for each of the two statements invokes the SUPINF tactic which handles integer inequalities in NUPRL.

```
* THM unjoin_wf
∀T:𝕌.  ∀≤:T → T → 𝔹.  unjoin(≤) ∈ WellFnd(T List,is_cons,|·|,T)
```

Figure 8.9: Theorem `unjoin_wf`

We can now define a function $mktree(\leq) : List(T) \to BinTree(T)$ that implements the first phase of QUICKSORT, that is, the generation of an ordered binary tree from a list.

**Definition 8.2.4 (mktree).** Suppose $T$ is a type, and $\leq : T \times T \to \mathbb{B}$. Define $mktree(\leq) : List(T) \to BinTree(T)$ by

$$mktree(\leq)(L) = unfold(is\_cons; unjoin(\leq))(L)$$

for all $L \in List(T)$.

The `mktree` abstraction and the associated well-formedness theorem `mktree_wf` are shown in Figure 8.10. The well-formedness theorem is proved in a single step by the Auto tactic.

```
* ABS mktree
mktree(≤)(x) == unfold(is_cons;unjoin(≤))(x)

* THM mktree_wf
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀L:T List.  mktree(≤)(L) ∈ BinTree(T)
```

Figure 8.10: Abstraction `mktree` and Theorem `mktree_wf`

Like for `is_cons` before, we prove two simple, yet useful lemmata about `mktree` that can later be used when we do structural induction on a list $L$. The first lemma proves $\mathit{mktree}(\leq)([]) = \mathit{leaf}$, and the second lemma proves $\mathit{mktree}(\leq)(u :: v) = \mathit{node}(u, \mathit{mktree}(\leq)(\mathit{filter}(\cdot\ \ \mathit{below}(\leq)\ \ u; v)), \mathit{mktree}(\leq)(\mathit{filter}(\cdot\ \ \mathit{above}(\leq)\ \ u; v))$. The lemmata are shown in Figure 8.11. The proof of `mktree_of_nil` is about seven steps long, and proving `mktree_of_cons` requires about nine steps—mainly just unfolding definitions.

```
* THM mktree_of_nil
∀T:𝕌.  ∀≤:T → T → 𝔹.  mktree(≤)([]) = leaf

* THM mktree_of_cons
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀u:T. ∀v:T List.
mktree(≤)(u::v) =
    node(u,
    mktree(≤)(filter((λb.b below(≤) u);v)),
    mktree(≤)(filter((λb.b above(≤) u);v)))
```

Figure 8.11: Theorems `mktree_of_nil` and `mktree_of_cons`

We have a second way of stating the Quicksort algorithm now: *quicksort* is equal to the composition of *mktree* and *flatten*.

**Theorem 8.2.5.** *Suppose $T$ is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$\mathit{quicksort}(\leq, L) = \mathit{flatten}(\mathit{mktree}(\leq)(L))$$

*for all $L \in \mathit{List}(T)$.*

The theorem `quicksort_by_fusion` shown in Figure 8.12 formalizes this result in Nuprl. To prove it, we first replace $\mathit{flatten} \cdot \mathit{mktree}$ with $\mathit{fun}$ using the `fusion` theorem. The ListLenInd tactic is then used to prove the resulting equality by complete induction on the length of $L$. A minor complication is introduced by the fact that the Fold tactic does not work for certain abstractions,[3] which forces us to work with the unfolded terms in some places. The proof is about 31 steps long.

```
* THM quicksort_by_fusion
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀L:T List.
    quicksort(≤,L) = flatten(mktree(≤)(L))
```

Figure 8.12: Theorem `quicksort_by_fusion`

# 8.3 A Formal Correctness Proof

Quicksort is a sorting algorithm: For every list $L$, it should return an ordered permutation of that list. We prove that Quicksort is correct by first proving that it returns an ordered list, and secondly by proving that it returns a permutation of its input. The first proof is based on the representation of *quicksort* as *flatten* · *mktree*, while the second proof uses the definition of *quicksort* directly.

## 8.3.1 Quicksort Returns an Ordered List

We say a list $L$ is *ordered* if the elements in $L$ are in ascending order (with respect to a relation $\leq$).

**Definition 8.3.1 (ordered).** Suppose $T$ is a type and $\leq : T \times T \to \mathbb{B}$. Define $\mathit{ordered}(\leq, L) \in \mathbb{B}$ recursively by

$$\mathit{ordered}(\leq, L) = \begin{cases} \text{true} & \text{if } L = [] \\ (\forall x \in t.\ h \leq x) \wedge \mathit{ordered}(\leq, t) & \text{if } L = h :: t \end{cases}.$$

By checking whether the head of the list is below *every* other element in the list (instead of just checking whether it is below the second element), we avoid having to check if there exists a second element in the list. The Nuprl abstraction defining `ordered` is shown in Figure 8.13. The well-formedness theorem `ordered_wf` proves $\mathit{ordered}(\leq, L) \in \mathbb{B}$ if $T$ is a type, $\leq : T \times T \to \mathbb{B}$ and $L \in \mathit{List}(T)$. The well-formedness theorem is proved by structural induction on $L$ using the ListInd tactic.

[3]Folding abstractions that contain `so_apply` seems to be a problem in some cases.

```
* ABS ordered
ordered(≤,L) ==
(letrec recfun(L) = case L
of [] => tt
| h::t => ∀x∈₂t.(h ≤ x) ∧_b recfun t esac )
L
```

Figure 8.13: Abstraction `ordered`

To prove that the list returned by *quicksort* = *flatten* · *mktree* is ordered, we first prove that *mktree* creates an ordered tree. Before we can define what it means for a binary tree to be ordered, we need to define a function that computes whether some predicate $P[x]$ holds for every element $x$ in a tree. The abstraction defining `tree_all_2` is shown in Figure 8.14. The name of the function ends with '`_2`' to indicate that a boolean value is returned (as opposed to a proposition in $\mathbb{P}$), thereby following the naming scheme for the `list_all` functions defined in the `LIST_3` library.

```
* ABS tree_all_2
∀x∈₂B.P[x] ==
(letrec recfun(B) = case B
of inl(y) => tt
| inr(z) => let t,B1,B2 = z in P[t] ∧_b recfun B1 ∧_b recfun B2 )
B
```

Figure 8.14: Abstraction `tree_all_2`

The well-formedness theorem `tree_all_2_wf` shows that $(\forall x \in_2 B.P[x])$ is a boolean value for every type $T$, $P : T \to \mathbb{B}$, and $B \in \mathit{BinTree}(T)$. It is proved in about eight steps; we use the RecElimination tactic in its proof for structural induction on $B$. We can now define when a binary tree is ordered.

**Definition 8.3.2 (treeordered).** Suppose $T$ is a type and $\leq : T \times T \to \mathbb{B}$. Define $\mathit{ordered}(\leq, B) \in \mathbb{B}$ recursively by

$$\mathit{ordered}(\leq, B) = \begin{cases} \text{true} & \text{if } B = \mathit{leaf} \\ (\forall z \in B_1.\ z \leq t) \wedge (\forall z \in B_2.\ \neg(z \leq t)) & \text{if } B = \mathit{node}(t, B_1, B_2) \\ \quad \wedge \mathit{ordered}(\leq, B_1) \wedge \mathit{ordered}(\leq, B_2) & \end{cases} .$$

The corresponding Nuprl abstraction `treeordered` is shown in Figure 8.15. As usual, we prove a well-formedness theorem for it: `treeordered_wf` just shows that for every type $T$, $\leq : T \times T \to \mathbb{B}$, and $B \in \mathit{BinTree}(T)$, $\mathit{ordered}(\leq, B) \in \mathbb{B}$. It is proved in about six steps by structural induction on $B$.

```
* ABS treeordered
ordered(≤,B) ==
(letrec recfun(B) = case B
of inl(x) => tt
| inr(y) => let t,B1,B2 = y in ∀z∈₂B1.(z ≤ t)
∧_b ∀z∈₂B2.(¬_b(z ≤ t))
∧_b recfun B1
∧_b recfun B2 )
B
```

Figure 8.15: Abstraction `treeordered`

**Lemma 8.3.3.** *Suppose $T$ is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$\mathit{ordered}(\leq, \mathit{mktree}(\leq)(L))$$

*for all $L \in \mathit{List}(T)$.*

Figure 8.16 shows the NUPRL theorem `ordered_mktree`. The arrow '↑' (*assert*) is used to turn the boolean value $\mathit{ordered}(\leq, \mathit{mktree}(\leq)(L))$ into a proposition, i.e. `tt` becomes *True*, `ff` becomes *False*.

```
* THM ordered_mktree
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀L:T List.
↑ordered(≤,mktree(≤)(L))
```

Figure 8.16: Theorem `ordered_mktree`

To prove the formal theorem, we need three lemmata: Firstly, that $f[x]$ holds for all $x$ in $\mathit{filter}(f; L)$ assuming $T$ is a type, $f : T \to \mathbb{B}$ and $L \in \mathit{List}(T)$. Secondly, that $P[x]$ holds for all $x \in L$ if and only if $P[x]$ holds for all $x$ in $\mathit{filter}(f; L)$ and for all $x$ in $\mathit{filter}(\neg f; L)$ assuming $T$ is a type, $P, f : T \to \mathbb{B}$, and $L \in \mathit{List}(T)$. Finally, that $f[x]$ holds for all $x$ in $L$ if and only if $f[x]$ holds for all $x$ in $\mathit{mktree}(\leq)(L)$ assuming $T$ is a type, $\leq : T \times T \to \mathbb{B}$, $f : T \to \mathbb{B}$, and $L \in \mathit{List}(T)$. The lemmata are shown in Figures 8.17, 8.18, and 8.19 respectively.

```
* THM filter_all_2
∀T:𝕌.  ∀f:T → 𝔹.  ∀L:T List.  ↑∀x∈₂filter(f;L).f[x]
```

Figure 8.17: Theorem `filter_all_2`

The `filter_all_2` lemma is proved in about eight steps by structural induction on $L$ using the LISTIND tactic. The base case is proved in a single step by the AUTO tactic. For the case $L = u :: v$, the IfThenElseCases tactic is used to do a case split on $f[u]$.

```
* THM list_all_2_filter_filter
∀T:𝕌.  ∀f,P:T → 𝔹.  ∀L:T List.
∀x∈₂L.P[x] = ∀x∈₂filter(f;L).P[x] ∧b ∀x∈₂filter((λz.¬bf[z]);L).P[x]
```

Figure 8.18: Theorem `list_all_2_filter_filter`

The `list_all_2_filter_filter` lemma is also proved by structural induction on $L$. The case $L = []$ is proved in a single step again, and for the case $L = u :: v$, we do a case split on $f[u]$ by IfThenElseCases. The resulting equalities are proved using the associativity and commutativity of $\wedge_b$. The proof is about eleven steps long.

```
* THM mktree_all_2
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀f:T → 𝔹.  ∀L:T List.
∀x∈₂L.f[x] = ∀x∈₂mktree(≤)(L).f[x]
```

Figure 8.19: Theorem `mktree_all_2`

Proving the `mktree_all_2` lemma is slightly more complicated. We start by using the ListLenInd tactic for complete induction on the length of $L$, followed by the ListInd tactic to differentiate between the two cases $L = []$ and $L = u :: v$. For the base case, we instantiate the lemma `mktree_of_nil`, and for the case $L = u :: v$, we use the `mktree_of_cons` lemma. The induction hypothesis is then used on the two lists $\mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v)$ and $\mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v)$. Finally the `list_all_2_filter_filter` lemma is used to prove the equivalence of $(\forall x \in_2 v.f[x])$ and $(\forall x \in_2 \mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v).f[x]) \wedge (\forall x \in_2 \mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v).f[x])$. The proof is about 23 steps long.

The proof of `ordered_mktree` then requires about 26 steps. It is based on complete induction on the length of $L$, using the ListLenInd tactic followed by ListInd. About 20 of those steps are needed to prove the case $L = u :: v$.

Our next step in proving that *quicksort* returns an ordered list is to show that $\mathit{flatten}(B)$ is an ordered list if $B$ is an ordered tree.

**Lemma 8.3.4.** *Suppose $T$ is a type, $\leq : T \times T \to \mathbb{B}$ is transitive and total (i.e. $x \leq y$ or $y \leq x$ for all $x, y \in T$), and $B \in \mathit{BinTree}(T)$. Then*

$$\mathit{ordered}(\leq, B) \implies \mathit{ordered}(\leq, \mathit{flatten}(B)).$$

```
* THM ordered_flatten
∀T:𝕌.
∀≤:{≤:T → T → 𝔹| Trans(T;x,y.↑≤[x;y]) ∧ Connex(T;x,y.↑≤[x;y])} .
∀B:BinTree(T).
↑ordered(≤,B) ⇒ ↑ordered(≤,flatten(B))
```

Figure 8.20: Theorem `ordered_flatten`

The corresponding NUPRL theorem `ordered_flatten` is shown in Figure 8.20.

We need a number of fairly self-evident lemmata before we can formally prove this theorem. The `list_all_2_append_lemma` lemma shown in Figure 8.21 proves that a property $P[x]$ holds for all $x$ in $L@M$ if and only if it holds for all $x$ in $L$ and for all $x$ in $M$. In other words, '∀' distributes over *append*. Using the LISTIND tactic for structural induction on $L$, the lemma is proved in about four steps.

```
* THM list_all_2_append_lemma
∀T:𝕌.  ∀P:T → 𝔹.  ∀L,M:T List.
∀x∈₂(L @ M).P[x] = ∀x∈₂L.P[x] ∧_b ∀x∈₂M.P[x]
```

Figure 8.21: Theorem `list_all_2_append_lemma`

Figure 8.22 shows a lemma proving that a list of the form $L@(t :: M)$ is ordered if and only if $L$ is ordered, $M$ is ordered, $x \leq t$ for all $x$ in $L$, and $t \leq x$ for all $x$ in $M$. To prove the lemma, we use structural induction on $L$, the `list_all_2_append_lemma` lemma and a number of other lemmata. A nested induction on $M$ and several case splits are required for the case where $L = u :: v$. The proof is about 60 steps long.

```
* THM ordered_append
∀T:𝕌.  ∀≤:{≤:T → T → 𝔹| Trans(T;x,y.↑≤[x;y])} .  ∀L,M:T List.
∀t:T.
ordered(≤,L @ (t::M)) =
    ∀x∈₂L.(x ≤ t) ∧_b ∀x∈₂M.(t ≤ x) ∧_b ordered(≤,L) ∧_b ordered(≤,M)
```

Figure 8.22: Theorem `ordered_append`

The `flatten_all_2` lemma (see Figure 8.23) shows that a property $f[x]$ holds for all $x$ in a binary tree $B$ if and only if it holds for all $x$ in *flatten*$(B)$. This lemma is similar to the `mktree_all_2` lemma proved earlier. The proof is by structural induction on $B$. It requires about 29 steps, including one instantiation of the `list_all_2_append_lemma` lemma.

```
* THM flatten_all_2
∀T:𝕌.  ∀f:T → 𝔹.  ∀B:BinTree(T). ∀x∈₂B.f[x] = ∀x∈₂flatten(B).f[x]
```

Figure 8.23: Theorem `flatten_all_2`

Figure 8.24 shows another lemma that we need, `list_all_2_implies_lemma`. It proves that if $P[x]$ and $(P[x] \implies Q[x])$ hold for all $x$ in a list $L$, then $Q[x]$ holds for all $x$ in $L$. The lemma is proved in about 13 steps by structural induction on $L$; many of those steps just deal with the fairly technical difference between boolean values and propositions.

```
* THM list_all_2_implies_lemma
∀T:𝕌.  ∀P,Q:T → 𝔹.  ∀L:T List.
↑∀x∈₂L.P[x] ∧ ↑∀x∈₂L.(P[x] ⇒b Q[x]) ⇒ ↑∀x∈₂L.Q[x]
```

Figure 8.24: Theorem `list_all_2_implies_lemma`

Our last lemma for now is shown in Figure 8.25. The `list_all_2_if_all` lemma proves that a property $P[x]$ holds for all $x$ in a list $L \in \mathit{List}(T)$ if it holds for all $x \in T$. It is proved in about six steps by structural induction on $L$.

```
* THM list_all_2_if_all
∀T:𝕌.  ∀P:T → 𝔹.  ∀L:T List.  (∀x:T. ↑P[x]) ⇒ ↑∀x∈₂L.P[x]
```

Figure 8.25: Theorem `list_all_2_if_all`

Given these lemmata, the proof of `ordered_flatten` requires about 58 steps. The RecElimination tactic is used for structural induction on $B$. The base case is then proved in about six steps simply by unfolding definitions. Proving the case $B = \mathit{node}(t, B_1, B_2)$ requires the use of the lemmata `ordered_append`, `flatten_all_2`, `list_all_2_implies_lemma` and `list_all_2_if_all`.

We proved that *mktree* always creates an ordered tree, and that *flatten* flattens an ordered tree into an ordered list. Given the `quicksort_by_fusion` theorem from Section 8.2, the proof that Quicksort always returns an ordered list is quite simple now.

To prove the `ordered_quicksort` theorem shown in Figure 8.26, we first replace $\mathit{quicksort}(\leq, L)$ with $\mathit{flatten}(\mathit{mktree}(\leq)(L))$ using the `quicksort_by_fusion` theorem. After using the `ordered_flatten` lemma then, we only have to prove that $\mathit{mktree}(\leq)(L)$ is ordered. This is proved by the `ordered_mktree` lemma. All well-formedness goals are discharged by Nuprl's Auto tactic, so the whole proof requires only about three steps.

```
* THM ordered_quicksort
∀T:𝕌.
∀≤:{≤:T → T → 𝔹| Trans(T;x,y.↑≤[x;y]) ∧ Connex(T;x,y.↑≤[x;y])} .
∀L:T List.
↑ordered(≤,quicksort(≤,L))
```

Figure 8.26: Theorem `ordered_quicksort`

### 8.3.2 Quicksort Returns a Permutation of its Input

In the previous subsection we proved that QUICKSORT always returns an ordered list. To prove that QUICKSORT is a sorting algorithm, it remains to show that the list returned by QUICKSORT is a permutation of the input list.

**Theorem 8.3.5.** *Suppose $T$ is a type, $eq : T \times T \to \mathbb{B}$ is a function with $eq(x, y) = true$ if and only if $x = y$ for all $x, y \in T$ (in other words, equality in $T$ is decidable), and $\leq : T \times T \to \mathbb{B}$. Furthermore, suppose $x \in T$ and $L \in List(T)$. Then $x$ occurs in $quicksort(\leq, L)$ exactly as often as in $L$.*

The idea of counting the occurrences of an element in $L$ and in $quicksort(\leq, L)$ is borrowed from [Sha96]. Figure 8.27 shows the NUPRL theorem `list_count_quicksort`. We used the abstractions `discrete_equality`, which can be found in the `DISCRETE` library, and `list_count` from the `LIST_3` library to state the theorem. We need a decidable equality on $T$ in order to be able to count the occurrences of a given element $x \in T$ in the two lists $L$ and $quicksort(\leq, L)$: If we could not tell whether two elements $x, y \in T$ are equal, we could not compare $x$ to the elements in $L$ and $quicksort(\leq, L)$.

```
* THM list_count_quicksort
∀T:𝕌.  ∀eq:{T=₂}.  ∀≤:T → T → 𝔹.  ∀L:T List.  ∀x:T.
|x∈quicksort(≤,L)| = |x∈L|
```

Figure 8.27: Theorem `list_count_quicksort`

We do not prove this theorem directly. Instead, we prove three lemmata first. The first lemma, `list_count_over_filter_lemma`, is shown in Figure 8.28. It proves that an element $x$ occurs in the list $filter(f; L)$ exactly as often as in $L$ if $f[x]$ is true, and zero times otherwise. The lemma is proved in about 33 steps using the LISTIND tactic for structural induction on $L$, combined with several applications of the IFTHENELSECASES tactic for case splits on $f[x]$ and—in the case $L = u :: v$—on

```
* THM list_count_over_filter_lemma
∀T:𝕌.  ∀eq:{T=₂}.  ∀f:T → 𝔹.  ∀L:T List.  ∀x:T.
|x∈filter(f;L)| = if f[x] then |x∈L| else 0 fi
```

Figure 8.28: Theorem `list_count_over_filter_lemma`

$f[u]$. The fact that we can decide whether $x$ is equal to $u$ (via the *eq* function) is crucial to the proof.

The second lemma, shown in Figure 8.29, states that an element $x$ occurs in $L$ exactly as often as in the two lists $\mathit{filter}(f; L)$ and $\mathit{filter}(\neg f; L)$ together. It is proved in about 16 steps. We apply the `list_count_over_filter_lemma` lemma twice in its proof: first to the list $\mathit{filter}(f; L)$, and then to the list $\mathit{filter}(\neg f; L)$.

```
* THM list_count_filter_filter_lemma
∀T:𝕌.  ∀eq:{T=₂}.  ∀f:T → 𝔹.  ∀L:T List.  ∀x:T.
|x∈filter(f;L)| + |x∈filter((λz.¬bf[z]);L)| = |x∈L|
```

Figure 8.29: Theorem `list_count_filter_filter_lemma`

Figure 8.30 shows the third lemma. This lemma is simply a specialized version of `list_count_over_filter_lemma` for the predicates *below* and *above*. Using the `list_count_over_filter_lemma` lemma, it is proved in two steps.

```
* THM list_count_below_above
∀T:𝕌.  ∀eq:{T=₂}.  ∀≤:T → T → 𝔹.  ∀L:T List.  ∀u,x:T.
|x∈filter((λb.b below(≤) u);L)| + |x∈filter((λb.b above(≤) u);L)|
    = |x∈L|
```

Figure 8.30: Theorem `list_count_below_above`

The proof of `list_count_quicksort` now requires about 55 steps. The ListLenInd tactic is used for complete induction on the length of $L$, followed by the ListInd tactic two differentiate between the two possible cases $L = []$ and $L = u :: v$. The case $L = []$ is proved in a single step by the Auto tactic after unfolding the definition of *quicksort*. For the case $L = u :: v$, we apply the `list_count_over_append_lemma` lemma from the `LIST_3` library to the two lists $\mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v))$ and $u :: \mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v))$. The induction hypothesis is then applied to the two lists $\mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v))$ and $\mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v))$. Finally `list_count_below_above` is used on the two lists $\mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v)$ and $\mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v)$.

This does not only complete the proof that QUICKSORT returns a permutation of its input list, but it is also the last step in our correctness proof for QUICKSORT. The next section presents an alternative approach to proving that QUICKSORT returns a permutation of its input.

### 8.3.3 Quicksort Returns a Permutation of its Input: A Second Proof

To prove that QUICKSORT returns a permutation of its input in the previous section, we counted the number of occurrences of elements in the lists $L$ and $quicksort(\leq, L)$. We cannot do this unless equality on $T$ is decidable. This is not a real restriction if $\leq$ is a decidable order relation on $T$: Then $x = y \iff (x \leq y \wedge y \leq x)$ for all $x$ and $y$ in $T$.[4] However, all theorems that we proved in the previous section only required $\leq$ to be total (i.e. $x \leq y \vee y \leq x$ for all $x, y \in T$) and transitive (i.e. $(x \leq y \wedge y \leq z) \implies x \leq z$ for all $x, y, z \in T$), and there is a different approach to proving that QUICKSORT returns a permutation of its input—an approach that does not require equality on $T$ to be decidable.

This approach is based on the inductive definition of `permutation` shown in Figure 8.31. The definition can be found in the `LIST_3` library.

```
* ABS permutation
perm(L,M) ==
(letrec perm(L)(M) = case L
of [] => case M
    of [] => True
    | h::t => False
esac
| h::t => case M
    of [] => False
    | h'::t' => ∃N,N':T List.  M = N @ (h::N') ∧ perm t (N @ N')
esac
esac )
L
M
```

Figure 8.31: Abstraction `permutation`

[4]The '⇒' direction follows from the reflexivity of $\leq$, and the antisymmetry of $\leq$ implies the '⇐' direction.

We also need two self-evident lemmata: that *permutation* is transitive, and that *permutation* distributes over append. The former is shown in Figure 8.32, and the latter in Figure 8.33.

```
* THM permutation_transitive
∀T:𝕌.  ∀L,M,N:T List.  perm(L,M) ⇒ perm(M,N) ⇒ perm(L,N)
```

Figure 8.32: Theorem `permutation_transitive`

```
* THM permutation_over_append_lemma
∀T:𝕌.  ∀A,B,X,Y:T List.  perm(A,X) ∧ perm(B,Y) ⇒ perm(A @ B,X @ Y)
```

Figure 8.33: Theorem `permutation_over_append_lemma`

We now prove a lemma similar to the `list_count_filter_filter_lemma` lemma shown in Figure 8.29: $L$ is a permutation of $\mathit{filter}(f;L)@\mathit{filter}(\neg f;L)$. This lemma, which is shown in Figure 8.34, is proved in about 23 steps by structural induction on $L$.

```
* THM permutation_filter_filter_lemma
∀T:𝕌.  ∀f:T → 𝔹.  ∀L:T List.
perm(L,filter(f;L) @ filter((λz.¬_b f[z]);L))
```

Figure 8.34: Theorem `permutation_filter_filter_lemma`

The `permutation_below_above` lemma (see Figure 8.35) simply results from applying the `permutation_filter_filter_lemma` lemma to the two predicates *below* and *above*. It is proved in about three steps.

```
* THM permutation_below_above
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀L:T List.  ∀u:T.
perm(L,filter((λb.b below(≤) u);L) @ filter((λb.b above(≤) u);L))
```

Figure 8.35: Theorem `permutation_below_above`

We can now show that $\mathit{quicksort}(\leq, L)$ is a permutation of $L$. Figure 8.36 shows the corresponding NUPRL theorem. It is proved by complete induction on the length of $L$ using the LISTLENIND tactic, followed by the LISTIND tactic to differentiate between $L = []$ and $L = u :: v$. The case $L = []$ is then proved in a single step by unfolding definitions and the AUTO tactic. Proving the case $L = u :: v$ requires approximately

```
* THM permutation_quicksort
∀T:𝕌.  ∀≤:T → T → 𝔹.  ∀L:T List.  perm(L,quicksort(≤,L))
```

Figure 8.36: Theorem `permutation_quicksort`

34 steps. A number of lemmata are instantiated in this part of the proof. Altogether, the proof is about 39 steps long.

This completes our second proof that QUICKSORT returns a permutation of its input.

# Chapter 9

# Conclusions

In this thesis we presented a formalization of program transformations and their general categorical framework in NUPRL. This chapter summarizes our results and points out possible future work.

## 9.1 Contributions

We formalized substantial parts of category theory in NUPRL. We gave formal definitions of catamorphisms and anamorphisms and formal, constructive proofs for when an arrow is a catamorphism or anamorphism. We showed that the result from [GHA01] for when a function is a catamorphism is not constructively valid, and we found conditions under which the result can be applied to constructive functions. We verified an instance of Bird's fusion theorem [Bir95] for binary trees in NUPRL, and applied it to the QUICKSORT algorithm to formally prove the algorithm's correctness.

## 9.2 Summary

In Chapter 4 we presented the notions of category theory that are needed for a definition of catamorphisms and anamorphisms. We showed that NUPRL's constructive type theory is well-suited to formalize these concepts. Using subtypes and the dependent product type, the formalization in NUPRL was straightforward, although verifying well-formedness was sometimes tedious. Many well-formedness goals arose from the fact that the composition of two arrows is defined only if the second arrow's domain is equal to the first arrow's codomain. The `category_if` lemma shows how we can greatly simplify many proofs by proving specialized well-formedness lemmas.

Catamorphisms and anamorphisms were defined in Chapter 5. For the most part, this was a straightforward extension of the formalization of category theory presented in Chapter 4. The formal definitions of fold and unfold presented a minor difficulty because we cannot express fold $g$ directly as a function of $g$. Therefore we defined fold and unfold as binary relations (as opposed to functions). We showed how NUPRL's display forms can be used to retain the common mathematical notation nevertheless. Also, dualizing our results from catamorphisms to anamorphisms was straightforward. Parts of the proofs of dual theorems could simply be copied from the original theorems in NUPRL. Manual interaction was still needed however, and a complete automatization of the dualization process is well beyond the scope of this thesis.

Perhaps our most significant theoretical results can be found in Chapter 6. In this chapter we addressed the question of when a constructive function is a catamorphism. Not only did we give a counterexample to a theorem proved in [GHA01] (thereby proving that the theorem is not constructively valid), but we also found a simple additional condition that allowed us to give a constructive proof. Whereas the results from [GHA01] could only be used to find out when two phases of a program *cannot* be fused into a single catamorphism, our results can also be used to find out when two phases *can* be fused—and for this case, we presented a transformation that actually writes the given function as a fold. Therefore our results are not only of theoretical interest, but they have practical applications in program optimization as well. It was only in this chapter that the differences between set theory and NUPRL's type theory caused any problems that required reformulation of the results in [GHA01] stated in terms of constructive category theory.

We verified an instance of Bird's fusion theorem in Chapter 7. Here we made extensive use of recursive types and functions. Many of NUPRL's powerful induction strategies where frequently used in the proofs. N. Shankar [Sha96] had to write a tactic similar to INVIMAGEIND when he verified Bird's fusion theorem in PVS. We did not have to write a single tactic in NUPRL; instead, we just employed existing tactics from the `YINDUCTIONS` library. For the most part, well-formedness goals were not an issue in this chapter.

In Chapter 8 we applied the fusion transformation to the QUICKSORT algorithm. Implementing QUICKSORT in NUPRL was no problem, but the proof of the associated well-formedness theorem presented a few technical difficulties. Other proofs were slightly more complicated than necessary because of a problem with the FOLD tactic. The overall development however was fairly straightforward. The correctness proof, although a few hundred steps long in total, was simplified and structured by the use of several lemmata, some of which we had to state and prove ourselves, and some of which were already in the basic NUPRL libraries. Our experience from Chapters 7 and 8 shows that NUPRL, as a general-purpose proof development system, is quite well suited for reasoning about program transformations as well.

## 9.3 Future Work

The partial formalization of category theory presented in Chapter 4 is only part of a much larger project: the formalization of (constructive) mathematics in NUPRL. Many have contributed to this project [Kre86, How87, For93, Jac95, CJNU97, Cal98], and work on it will certainly continue.

More research should be done on the question when a constructive function is a catamorphism or anamorphism. In Chapter 6 we gave a partial answer based on results for non-constructive functions, but not a complete classification. Also the results from Chapter 6 were not dualized to anamorphisms in this thesis.

Finally, it remains to be seen whether the transformation presented in Chapter 6, which writes certain functions as a fold, and which we manually applied to a small example in this thesis, is simple and useful enough to be actually implemented in an optimizing compiler.

# Bibliography

[Acz93] P. Aczel. Galois: A theory development project. Report for the 1993 Turin meeting on the Representation of Mathematics in Logical Frameworks, 1993.

[Acz99] P. Aczel. On relating type theories and set theories. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs (TYPES '98)*, volume 1657 of *LNCS*, pages 1–18, 1999.

[AP90] James A. Altucher and Prakash Panangaden. A mechanically assisted constructive proof in category theory. In *Proceedings of the 10th International Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer-Verlag, July 1990.

[Bax01] Ira D. Baxter. DMS: Practical code generation and enhancement by program transformation, 2001.

[BBC+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, INRIA, 1997.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[BHW97] J. Boyle, T. Harmer, and V. Winter. The TAMPR program transformation system: Design and applications. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*. Brikhauser, 1997.

[Bir84] Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4), October 1984.

[Bir95] Richard S. Bird. Functional algorithm design. In Bernhard Moller, editor, *Mathematics of Program Construction '95*, volume 947 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 1995.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, second edition, 1998.

[BM92] L. M. Barroca and J. A. McDermid. Formal methods: use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579–599, 1992.

[BS93] J. P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, 1993.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[Cal98] J. L. Caldwell. Classical propositional decidability via Nuprl proof extraction. In *Proc. 11th International Theorem Proving in Higher Order Logics Conference*, pages 105–122, 1998.

[Cal02] James Caldwell. Extracting recursion operators in Nuprl's type-theory. In A. Pettorossi, editor, *Eleventh International Workshop on Logic -based Program Synthesis, LOPSTR-02*, volume 2372 of *LNCS*, pages 124–131. Springer, 2002.

[Car98] A. Carvalho. Category theory in COQ. Technical report, 1049-001 Lisboa, Portugal, 1998.

[CJNU97] Robert L. Constable, Paul B. Jackson, Pavel Naumov, and Juan Uribe. Constructively formalizing automata theory, 1997.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[Cor00] James R. Cordy. *The TXL Compiler/Interpreter User's Guide Version 10*. TXL Software Research Inc., Kingston, Canada, January 2000.

[Dav87] M. Davis. Deforestation: Transformation of functional programs to eliminate intermediate trees. Master's thesis, Oxford University, 1987.

[EM42] Samuel Eilenberg and Saunders MacLane. Group extensions and homology. *Annals of Mathematics*, 43:757–831, 1942.

[EM45] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58:231–294, 1945.

[ES95] William F. Eddy and Mark J. Schervish. How many comparisons does Quicksort use? *Journal of Algorithms*, 19(3):402–431, November 1995.

[For93] Max B. Forester. Formalizing constructive real analysis. Technical Report TR93-1382, Computer Science Department, Cornell University, Ithaca, NY, 1993.

[GHA01] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold? In Andrea Corradini, Marina Lenisa, and Ugo Montanari, editors, *Proceedings 4th Workshop on Coalgebraic Methods in Computer Science, CMCS'01, Genova, Italy, 6–7 Apr. 2001*, volume 44(1). Elsevier, Amsterdam, 2001.

[Gir72] Jean-Yves Girard. *Interprétation Fonctionelle et Élimination des Compures de l'Arithmétic d'Ordre Supérieur*. PhD thesis, Université Paris VII, 1972.

[GJ98] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34(1), pages 273–279. ACM Press, New York, 1998.

[GJS93] A. Gill, Launchbury J., and Peyton Jones S.L. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[HE99] M. G. Hinchey and J. P. Bowen (Eds.). *Industrial-Strength Formal Methods in Practice*. Springer, 1999.

[Her00] Debra S. Herrmann. *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*. IEEE, 2000.

[Hoa61] C. A. R. Hoare. ACM Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961.

[How87] Douglas J. Howe. Implementing number theory: An experiment with Nuprl. In *8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 404–415. Springer-Verlag, 1987.

[How93] Douglas J. Howe. Reasoning about functional programs in Nuprl. In Peter F. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693, pages 145–164. Springer-Verlag, 1993.

[HS98] G. Huet and A. Saibi. Constructive category theory. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[Hut98] Graham Hutton. Fold and unfold for program semantics. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34(1), pages 280–288. ACM Press, New York, 1998.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[Jac94] Paul B. Jackson. *The Nuprl Proof Development System, Version 4.1 Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 1994.

[Jac95] Paul B. Jackson. Enhancing the nuprl proof development system and applying it to computational abstract algebra. Technical Report TR95-1509, Computer Science Department, Cornell University, Ithaca, NY, 18, 1995.

[Jon96] Simon L. Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In *European Symposium on Programming*, pages 18–44, 1996.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[Kre86] Christoph Kreitz. Constructive automata theory implemented with the Nuprl proof development system, 1986.

[LP92] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[Mac97] Saunders MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2nd edition, 1997. (1st ed., 1971).

[Mal90] G. Malcolm. Algebraic data types and program transformation. *Science of Computer Programming*, 14(2–3):255–280, 1990.

[Mar02] Jean-Pierre Marquis. Category theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab at the Center for the Study of Language and Information, Stanford University, Stanford, CA, Summer 2002.

[Mee86] Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334, North–Holland, 1986.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[Miz] The Mizar Home Page. `http://www.mizar.org`.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, 1982.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[OSR95] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. CSL, 1995.

[RB88] David Rydeheard and Rod Burstall. *Computational Category Theory (Prentice Hall International Series in Computer Science)*. Prentice Hall, 1988.

[Sha96] Natarajan Shankar. Steps toward mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1-3):33–57, 1996.

[Smi90] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[Sto96] Neil Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society, Series 2*, volume 42, pages 230–265, 1936.

[Vis01a] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[Vis01b] Eelco Visser. A survey of strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57/2 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Berlin: Springer-Verlag, 1988.

[Wan80] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[Wer97] Benjamin Werner. Sets in types, types in sets. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, (TACS '97)*, volume 1281 of *Lecture Notes in Computer Science*, pages 530–546. Springer, 1997.