Generic theorem proving using hol2p: a Category Theory inspired approach

Masters of Philosophy Thesis

of

Keisha Harriott

born on the 17th of July 1979 in Kingston Jamaica

February 12, 2014

Supervisor:

Dr. N. Voelker

University of Essex, Computer Science and Electronic Engineering, School of

Abstract

Integrating formal program verification into mainstream software development has proven to be quite challenging, due to the level of abstract mathematical machinery needed. Although there have been some successes, most existing methods do not adequately support the mechanical verification of generic programs. This thesis seeks to fill this gap by presenting a formalisation and implementation of a category theory inspired approach to generic program specification. Theorems to simplify verification of generic programs are developed along with a formal framework for reasoning. The result is theorem proving support based on type quantification and type operator variables in HOL, HOL2P. This is demonstrated by the verification the Yoenda Lemma.

Acknowledgment

I would like to thank Norbert Völker for his supervision during this research and for his invaluable comments, suggestions and countless reviews during during this research. The author would also like to thank the IFUW for part funding of this research. For my family for their support especially to my husband. I must also mention my son who kept my company for the entire time.

CONTENTS

CHAPTER 1. BACKGROUND AND MOTIVATION

Theorem provers have been used in various disciplines as a proof assistant. Mathematicians use these powerful tools to assist in proving theorems which would otherwise be laborious. Researchers in these fields have realized the upshot of automated reasoning; increased reassurance and reliability leading to renewed confidence in both users and developers.

The use of theorem provers in software verification and specification has had mixed fortunes. Although clearly beneficial in the overall goal of achieving correct and reliable software, theorem proving practices remain more or less an esoteric practice as most developers find the process too difficult and costly to incorporate in mainstream development.

There are many theorem provers available, built on a range of logical framework including First Order and Higher Order Logic [49]. Higher order logic extends first order predicate calculus in three non trivial ways:

1. variables are allowed to range over functions and predicates
2. functions can take functions as arguments and yield functions as results
3. the notation of the λ-calculus can be used to write terms which denote functions

Theorem provers for higher order logic such as Isabelle/HOL [43], Coq [53], PVS [51] and HOL Light [22] have all been used to solve non-trivial problems. Notwithstanding these achievements there is still much that needs to be done for theorem provers to be viewed as a software developer's companion.

The underlying set and type theory serves as the core of provers but is just one aspect of the entire software development process. In order to encourage the successful incorporation of provers, in mainstream development, one has to provide higher level facilities that uses the underlying logic; software developers should not have to work on low levels. This has to be a two step process, it is not sufficient to be able to express concepts logically, reasonable simple mechanisms must be provided for reasoning with these logical concepts. To some extent, such mechanism already exists; there are mechanisms for introducing some mathematical concepts, for instance inductive types and recursive function definitions.

The association of functional programs with categories has sparked a considerable amount of research interest in incorporating category theory inspired concepts in functional programming and theorem proving. This is due to the high level of abstraction that category theory has to offer; a definite requirement for generic programming and reasoning. The correlation of datatypes as objects of a category and operations as morphisms, means that reasoning on the level of objects and morphisms can be instantiated to particular types and corresponding operations. Generics would make it possible to reason about types and methods that are different only in type structure specification. This will maximize code reuse, type safety and hence reliability, and performance.

While first-order logic has syntactic categories for individuals, functions, and predicates, only quantification over individuals is permitted. Many concepts when translated into logic are, however, naturally expressed using quantifiers over functions and predicates. Higher-order logics offer a natural representation of higher-order quantification [28].

1.1. Motivation

Generic programming is a style of functional programming where programs are parameterized with respect to datatypes. The Generic Programming process focuses on the commonality among similar implementations of the same functions, then providing suitable abstractions so that a single, generic function can cover many concrete implementations. The abstractions themselves are expressed as requirements on the parameters to the generic function. Generic Programming has had some success, for instance, Jansen has developed a preprocessor for PolyP [27] where programs are parameterized with base functors. However the theorem proving counterpart has been lacking. Our research is therefore motivated by this lack of theorem proving support for generic program development.

To accomplish our goal of providing support for generic program development we identify a suitable mathematical representation of datatypes and functions. The Bird-Meertens formalism provided the algebraic theory for this style of programming [39, 4] and has origins in category theory. The first step would be to specify a logical framework based on this abstraction so as to achieve a readable formalism of category theory that seamlessly supports proof automation. The second step is the mechanisation of this logical framework using a theorem prover that can adequately support the formalism. We now motivate the logic and corresponding theorem prover used in its mechanisation.

Logical formalism - There are many theorem provers available, built on a range of logical framework including first order logic and higher order logic (HOL). In general the predicate calculus, defines which statements of logic are provable and consists of: formation rules, and a proof theory - axioms and transformation rules used for deriving theorems, and semantics. The most prominent provers, to date, are built on first order logic. First order logic is restrictive in its expressiveness, as there is really no interesting type system. First order logic would therefore not adequately represent generic programs. As a result there is a heighten interest in a logical framework that offers the opportunity to express a wider range of problems. Higher order logic (HOL) offers this expressiveness, and is the reason for our logic of choice.

Mechanizing category theory - Categories have been mechanized in HOL provers most popular of which uses a dependent type theory [13, 1, 50, 26]. Peter Aczel led the project, Galois [1], whose aim is to "formalise some abstract algebra in a predicative style". Formalisms based on dependent types are however very difficult to follow without specialist knowledge of dependent type theory and as such we aim to contribute a more readable formalism of category theory.

The most prevalent area of research that uses the Bird-Meertens Formalism is program optimization. Research on the verification of program transformations using higher order theorem provers, based on the use of category theory has been scarce. Although category theory inspired approaches have been used in compiler optimization in functional programming languages, majority of which is based on Haskell [37, 17, 30, 44], to my knowledge, this represent the first attempt at semi-automatic generic program transformation techniques using HOL-based provers. Our approach is further distinguished in that our transformational tool is built on generic theories, and therefore performs transformations on generic programs.

1.2. Background

To successfully develop theorem proving support for generic program development one needs to know Category Theory and Higher order logic. This section provides background knowledge of the concepts used in this research. For more a more in depth treatment the reader will be referred to a relevant source.

1.2.1. Category Theory

The material described in this section is well documented and may be found in several texts [11, 5, 48, 55, 3]. The paradigm of datatype definition used is due to Hagino [19], an important aspect of which is that datatypes are characterized by a universal property. This universal property prescribes the construction of specific recursive functions on the defined datatype. The universal property by means of which datatypes are characterized provides a conciseness that gives this approach its charm. These recursive functions are called *catamorphisms.* Catamorphisms play a prominent role in the theory of datatypes introduced in this chapter. The data types that can be defined using this paradigm are finite datatypes such as *list* and *tree* and are initial objects in a specific category of algebras.

This section introduces the applicable parts of category theory along with the adopted notation. Most of our notational conventions are standard. Deviations from standard notation occur when standard notation is not well suited for calculation and manipulation. Most of the notational conventions we adopt have been introduced by Bird [7, 8, 11]. In this section, Categories, Functors, F-Algebras and Natural Transformations are defined.

A *category*, $\mathcal{C}$, is an algebraic structure consisting of a class of objects, denoted using upper case letters, A, B, C etc., and a class of arrows, denoted by lower case letters f, g, h etc., together with three total operations and one partial operation.

Definition 1.1. *(Category) A* ***category*** $\mathcal{C}$ *can be described as a set, whose members are the objects of* $\mathcal{C}$*, satisfying the following three conditions:*

1. *Morphism : For every pair* X, Y *of objects, there is a set* $\mathbb{HOM}(X, Y)$*, called*

the morphisms from X to Y in $\mathcal{C}$. If f is a morphism from X to Y, we write $f : X \to Y$.

2. *Identity : For every object X, there exists a morphism id_X in $\mathbb{HOM}(X, X)$, called the identity on X.*

3. *Composition : The partial operation, composition, takes two arrows to another. For every triple X, Y and Z of objects, the composition is defined if and only if $f : X \to Y$ and $g : Y \to Z$. The composition of f and g is notated $(g; ; f) : X \to Z$.*

Identity, morphisms, and composition satisfy two axioms:

1. *Associativity : If $f : X \to Y$, $g : Y \to Z$ and $h : Z \to W$, then*

$$h; ; (g; ; f) = (h; ; g); ; f \tag{1}$$

2. *Identity : If $f : X \to Y$, then*

$$(id_Y; ; f) = f \text{ and } (f; ; id_X) = f \tag{2}$$

.

$\mathcal{FUN}$ is the category whose objects are sets and whose arrows are total functions with the usual composition of functions for ; ; and the identity function from S to S for id_S. f is an arrow from A to B if A contains the range of f and the set B is the domain of f.

$\mathcal{REL}$ is the category whose objects are sets and whose arrows are relations. An arrow $r : A \to B$ where r is a subset of the Cartesian product $A \times B$ where A is the source and B is the target. The identity arrow $id_A : A \rightarrow A$ is the relation $id_A = \{(a, a) \,|a \in A\}$ and the composition of arrows $r : A \rightarrow B$ and $s : B \rightarrow C$ is the arrow $t : A \rightarrow C$, where $t = \{(a,c)|(\exists b : (a,b) \in r \wedge (b,c) \in s)\}$.

A subcategory of a category $\mathcal{C}$ is a category $\mathcal{D}$ whose objects are objects in $\mathcal{C}$ and whose morphisms are morphisms in $\mathcal{C}$ with the same identities and composition of morphisms. Intuitively, a subcategory of $\mathcal{C}$ is a category obtained from $\mathcal{C}$ by "removing" some of its objects and arrows.

Definition 1.2. *(Sub-category) A* ***subcategory*** *$\mathcal{D}$ of a category $\mathcal{C}$ is a category for which:*

1. *All the objects of $\mathcal{D}$ are objects of $\mathcal{C}$ and all the arrows of $\mathcal{D}$ are arrows of $\mathcal{C}$*
2. *The source and target of an arrow of $\mathcal{D}$ are the same as its source and target in $\mathcal{C}$ (in other words, the source and target maps for $\mathcal{D}$ are the restrictions of those for $\mathcal{C}$). It follows that for any objects A and B of $\mathcal{D}$, $\mathbb{HOM}_{\mathcal{D}}(A,B) \subseteq \mathbb{HOM}_{\mathcal{C}}(A,B)$*
3. *If A is an object of $\mathcal{D}$ then its identity arrow id_A in $\mathcal{C}$ is in $\mathcal{D}$*
4. *If $f : A \rightarrow B$ and $g : B \rightarrow C$ in $\mathcal{D}$, then the composite (in $\mathcal{C}$) $g\,;;\,f$ is in $\mathcal{D}$ and is the composite in $\mathcal{D}$*

We also have a relation between two categories through what is called a *functor*. A functor is a structure preserving map, a *homomorphism*, between categories. That is given two categories $\mathcal{C}$ and $\mathcal{D}$, a functor consist of two mappings: one maps objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and the other maps arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$.

Definition 1.3. *(Functor) A* ***Functor*** $\mathsf{F} : \mathcal{C} \to \mathcal{D}$ *from a category* $\mathcal{C}$ *to* $\mathcal{D}$ *is a morphism of categories; consisting of a pair of mappings* $\mathsf{F}_0 : \mathcal{C}_0 \to \mathcal{D}_0$ *and* $\mathsf{F}_1 : \mathcal{C}_1 \to \mathcal{D}_1$ *for which the following holds:*

1. *If* $f : A \to B$ *in* $\mathcal{C}$*, then* $\mathsf{F}_1 : \mathsf{F}_0(A) \to \mathsf{F}_0(B)$ *in* $\mathcal{D}$
2. *For any object* A *of* $\mathcal{C}$*, then* $\mathsf{F}_1(id_A) = id_{\mathsf{F}_0(A)}$
3. *If* $g;;f$ *is defined in* $\mathcal{C}$*, then* $\mathsf{F}_1(g);;\mathsf{F}_1(f) = \mathsf{F}_1(g;;f)$

An ***Endofunctor*** *of a category* $\mathcal{C}$*, is a functor from* $\mathcal{C}$ *to* $\mathcal{C}$*.*

One example of a functor is the *identity functor* $id : \mathcal{C} \to \mathcal{C}$, which leaves objects and arrows unchanged. Another example of a functor is the *product functor.* The *product functor* $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ on a category $\mathcal{C}$ is defined by taking the arrows f and g into the Cartesian product of f and g and taking objects A and B into the binary product of A and B. Type constructors may also be interpreted as functors. The following example illustrate the list type constructor when viewed as a functor.

Example 1.1. *Let* A*,* B*, list* A *and list* B *be objects in a category* $\mathcal{C}$*. There is a the functor* F *that can be described as the pair of mappings,* F_0 *and* F_1*. Now* F_0 *maps the object* A *to object list* A *and maps the object* B *to list* B*. This is called list type constructor in functional programming. The function* F_1*, takes the arrow* $A \to B$*, denoted* f *in the diagram below, and maps it to the arrow list* $A \to$ *list* B*, denoted* $F(f)$*. This is the type functor, known more popularly as* `map`*.*

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "F_0(A)"'] & B \arrow[d, "F_0(B)"] \\
listA \arrow[r, "F_1(f)"'] & ListB
\end{tikzcd}$$

F-Algebras Type constructors are a special class of functors that specifies initial algebras. We examine the use of functors in the specification of a category of F-Algebras. These definitions will be useful later when we formalise the concept of catamorphisms.

Definition 1.4. *(F-Algebra) Let $\mathcal{C}$ be a category and $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An F-Algebra is a pair (F, a) where $a : \mathsf{F}A \to A$ is an arrow of the category $\mathcal{C}$. The object A is the carrier of the algebra and the functor F is the signature of the algebra.*

Definition 1.5. *(Algebra homomorphisms) A homomorphism between F-Algebras (A, a) and (B, b) is an arrow $f : A \to B$ of A such that the diagram commutes i.e. such that*

$$a;;f = \mathsf{F} \circ b\, f$$

$$\begin{tikzcd} FA \arrow[r, "a"] \arrow[d, "Ff"'] & A \arrow[d, "f"] \\ FB \arrow[r, "b"'] & B \end{tikzcd}$$

This construction gives a category $(\mathsf{F} : A)$ of F-Algebras.

Example 1.2. *(Category of Algebras) The category of F-algebras over F is defined by:*

- *Objects: pairs $(\mathsf{F}$,$a)$ i.e. arrows a of F such that source $a = \mathsf{F}$ (target a).*
- *Arrows: triples $(a, b, f) : a \to b$ where a and b are F-algebras and f : target $a \to b$ is a homomorphism from a to b.*
- *Identity: $id_a = (id_{target\ a}, a, a)$.*
- *Composition: $(g, a_1, a_2);;(f, a_2, a_3) = (g;;f, a_1, a_3)$.*

- *Source: Given arrow $f : a \to b$ then source $f = a$*

- *Target: Given arrow $f : a \to b$ then target $f = b$*

Definition 1.6. *(Isomorphims)*

Let C be a category, and let X, be objects of $\mathcal{C}$. A morphism $f : X \to Y$ is an isomorphism if there exists a morphism $g : Y \to X$ such that the follow holds:

$$g;;f = id_Y \tag{3}$$

$$f;;g = id_X \tag{4}$$

where id_X denotes the identity morphism on X.

Example 1.3. *Consider the endofunctor $\mathsf{F} : \mathcal{SET} \to \mathcal{SET}$ which takes S into $1 + S$. The natural numbers N form an algebra for F. The F-structure is given by the function $(zero, succ) : 1 + \mathbb{N} \to \mathbb{N}$ where zero is the function picking out 0 and succ is the successor function. This is also the algebra for F. $\mathbb{N}$ has many subsets which are all fixed up to isomorphism for F. Initial objects are determined uniquely up to a unique isomorphism.*

Definition 1.7. *(Initial Object) An object I in a category $\mathcal{C}$ is initial if for each object A of $\mathcal{C}$ there is exactly one arrow of type $I \to A$. Inital objects, often denoted α, are unique up to unique ismorphisms.*

Definition 1.8. *(Catamorphism) The* ***Catamorphism****, h is defined as a homomorphism from an initial algebra, α to h, and is denoted by $(|h|)$.*

Natural Transformations Let $\mathsf{F}, \mathsf{G} : \mathcal{A} \to \mathcal{B}$ be functors between two categories $\mathcal{A}$ and $\mathcal{B}$. By definition, a natural transformation to F from G is a collection of arrows

$\phi_B : FB \to GB$, one for each object $\mathbf{B}$ of $\mathcal{B}$. These arrows are called the components of ϕ. A transformation is called natural if

$$\mathsf{F}h;;\phi_B = \mathsf{F}h;;\phi_A \tag{5}$$

for all arrows $h : A \to B$ in $\mathcal{B}$. In a diagram, this equation can be pictured as

$$\begin{tikzcd} FB \arrow[r, "\phi_B"] \arrow[d, "Fh"'] & GB \arrow[d, "Gh"] \\ FA \arrow[r, "\phi_A"'] & GA \end{tikzcd}$$

We write $\phi : \mathsf{F} \to \mathsf{G}$ to indicate that a transformation ϕ to F from G is natural.

Example 1.4. *For example consider the function inits that returns all prefixes of its arguments:*

$$inits[a_1, a_2, \ldots, a_n] = [[], [a_1], [a_1, a_2], \ldots, [a_1, a_2, \ldots, a_n]] \tag{6}$$

For each set A there is an arrow $inits_A : list\ A \to list\ (list\ A)$. Now,
$map\ (map\ f) \circ inits = inits \circ map\ f$ and therefore inits is a natural transformation.

Category Theory Notation We follow the general notations used in category theory literature with one exception. In general $\circ$ is used for morphisms composition. We use $\circ$ to represent function composition and therefore when referring to compositions of morphisms at the abstract category level we choose to use $;;$. For a full list of notations used for category theory please refer to table 1 below:

Table 1: Category Theory Notation

English	Category Theory Notation
Set S	$\mathbb{S}$
Category C	$\mathcal{C}$
Objects	$A,\ B,\ C\ \cdots$
Arrow	$f, g, h, \cdots$ or $f : A \to B$
Composition in Categories	$;;$
Functor	F, G
Catamorphism of h	$(\!|h|\!)$
Natural transformation	ϕ

1.2.2. Higher Order Logic

We now turn to detail the logic of choice Higher Order Logic (HOL). Key features needed for our formalism are discussed such as syntax, language and type system.

Syntax The syntax of HOL is that of simple-typed λ-calculus. *Types* are either variables α, or applications $(\tau_1, ..., \tau_n)t$. There are two primitive types, *ind* denoting the set of *individuals* (a set with infinitely many distinct elements) and *bool* denoting the two element set of boolean values. There is also the primitive type operator $fun(\to)$. All new types are defined using the two primitive types and the primitive type operator fun. *Terms* are either typed constants c_τ or variables x_τ, applications $(e_1 e_2)$ and abstractions $\lambda x : T.e$. Terms must be well typed according to typing rules.

There are two primitive constant $=: \alpha \to \alpha \to bool$ and Hilbert's choice operator $\epsilon : (\alpha \to bool) \to \alpha$.

Notation Table 2 below summaries the standard notation used in HOL.

Table 2: HOL Notation

English	HOL Notation
True	T
False	F
not	$\neg$
and	$\wedge$
or	$\vee$
implication	$\Rightarrow$
equivalence	$\Leftrightarrow$
there exists	$\exists$
there exists unique	$\exists!$
for all	$\forall$
Proposition that x has the property specified by P	$P[x]$
function from x to y	$\lambda x.y$
function composition	$f \circ g$
some x such that P	$\epsilon x.P$

Types Every well-formed term must have a type that is consistent with the types of its sub terms. HOL have a set-theoretic semantics in which types are mapped to sets and terms to elements. Writing $tm : ty$ indicates explicitly that the term tm has type ty. HOL uses Milner's type inferencing algorithm [42] to assign consistent types to logical terms. The user of HOL therefore is rarely required to explicitly provide type information for terms. Types are very important as they prevent programmers from writing flawed code thereby improving reliability. Grouping the data manipulated by programs into types, is one way of preventing operations from being applied to scenarios in which they are not defined such as, returning the head or tail of an integer. Type restrictions are also very effective at thwarting basic attacks on security vulnerabilities such as the infamous buffer overflows attacks.

Types in higher order logic may be categorised as one of the following: type constants, type variables, and compound types. Type constants are identifiers that name sets of values. Examples are the two primitive types *bool* and *ind*, which denote the set of booleans and the set of "individuals" (an infinite set) respectively. Another example is the type constant *num*, which denotes the set of natural numbers. The type *num* is not primitive but is defined in terms of *ind*. Type variables are used to stand for "any type"; they are written α, β, γ, etc. Types that contain "type variables" are called polymorphic types. A substitution instance of a polymorphic type *ty* is a type obtained by substituting types for all occurrences of one or more of the type variables in *ty*. Compound types are expressions built from other types using type operators. They have the form: $(ty_1, ty_2, \cdots, ty_n)op$, where ty_1 through ty_n are types and *op* is the name of an $n-ary$ type operator. An example is the binary type operator *fun*, which denotes the function space operation on types. The compound type $(num, bool)fun$ (also written as $num \rightarrow bool$), is the type of all functions from *num* to *bool*.

Formally, all proofs in higher order logic can be formulated using type variables, the primitive type constants *bool* and *ind*, and the primitive type operator *fun*. In keeping with the aim of making theorem proving accessible to everyday software developers, it is desirable to extend this syntax of types to facilitate the addition new type constants and type operators, this is achieved using type definitions.

Hilbert's ϵ-operator The syntax and informal semantics are as follows. If $P[x : ty]$ is a boolean term involving a variable x of type *ty* then $\epsilon x.P[x]$ denotes some value,

v say, of type ty such that $P[v]$ is true. If there is no such value (i.e. $P[v]$ is false for each value v of type ty) then $x.P[x]$ denotes some fixed but arbitrarily chosen value of type ty. Thus, for example, $\epsilon n.4 < n \wedge n < 6$ denotes the value 5, $\epsilon n.(\exists m.n = 2m)$ denotes an unspecified even natural number, and $\epsilon n.n < n$ denotes an arbitrary natural number. Hilbert's $\epsilon - operator$ is formalised in higher order logic by the following theorem:

$$\forall P.(\exists x.Px) \Rightarrow P(\epsilon x.Px) \tag{7}$$

Therefore if P is a predicate and $\exists x.Px$ is a theorem of the logic, then so is $P(\epsilon x.Px)$. The ϵ-operator can therefore be used to obtain a logical term which provably denotes a value with a given property P from a theorem merely stating that such a value exists [32]. As consequence of the use of ϵ choice operator in type specification emptiness is not allowed, i.e. logical types must denote non-empty sets.

1.2.3. Higher Order Logic provers

HOL-Light HOL Light is a prover whose implementation of the underlying logic is accessible and simple. HOL Light is open source and was coded in metalanguage often called ML. This presents an opportunity for relatively easy modification. There are also special-purpose tools that are available to aid in overall theory development. For example there is code to automate the definition of inductive relations and types as well as to define recursive functions with arbitrary well-founded measures. There are also domain-specific tools such as decision procedures for linear arithmetic, over naturals, integers and reals [24, 21]. HOL traditional type system does not directly facilitate type parameterization. This is desirable for our work as it would add additional level of genericity.

Notation The HOL Light syntax for higher order logic is straightforward. For example the principle of induction on the natural numbers

$$\forall P : P(0) \land (\forall n : P(n) \Rightarrow P(S(n))) \Rightarrow (\forall n.P(n)) \tag{8}$$

would be written as

Listing 1: Principle of induction on the natural numbers

```
# !P. P 0 /\ (!n. P n ==> P(SUC n)) ==> (!n. P n)
```

where `SUC` is a predefined constant, $\forall$ is written as `!` and $\land$ is written as `/\`.Table 3 summarises conventional notations used for propositional (or Boolean) connectives, together with HOL's ASCII approximations and their approximate English reading.

Table 3: HOL Light Notation

English	HOL Light Notation
True	`T`
False	`F`
not	`~`
and	`/\`
or	`\/`
implication	`==>`
equivalence	`<=>`
there exists	`?`
there exists unique	`?!`
for all	`!`
Proposition that x has the property specified by `P`	`P(x)` or `P x`
function composition	`f o g`
Lambda expressions $\lambda x.x + 1$	`\x.x+1`
some x such that P	`@x.P`

Terms In HOL Light, mathematical expressions in higher order logic are enclosed in backquotes. These expressions have the OCaml type term. For example, if you enter ‘x + 1‘ at the OCaml toplevel (followed by two semicolons and return):

Listing 2: Term x + 1

```
# ‘x + 1‘;;
    val it : term = ‘x + 1‘
```

HOL theorem prover provides a number of operations for manipulating terms. For example `subst` will replace one term by another at all its occurrences in another term, e.g. replace ‘1‘ by ‘2‘ in the term ‘x + 1‘. The syntax is analogous to the logical notation $[2 = 1](x + 1)$ or (x + 1)[2 = 1] that one often sees:

Listing 3: `subst`

```
# subst [‘2‘,‘1‘] ‘x + 1‘;;
val it : term = ‘x + 2‘
# subst [‘y + 2‘,‘x:num‘] ‘x + 5 ∗ x‘;;
val it : term = ‘(y + 2) + 5 ∗ (y + 2)‘
```

The reason for entering ‘x:num‘ rather than just ‘x‘ lies in HOL’s type system, explained next.

Types A powerful feature of HOL is that every term must be have a well defined type. The type indicates what kind of object the term represents (a number, a set, a function, etc). The possible types of terms are represented using another symbolic datatype `hol_type`, and these will similarly be automatically parsed and printed within backquotes with a colon as the first character.

Listing 4: Types in HOL Light

```
# ‘:num‘;;
val it : hol_type = ‘:num‘
```

The type of a term can be obtained by applying the `type_of` operator to it.

Listing 5: `type_of`

```
# type_of ‘1‘;;
val it : hol_type = ‘:num‘
# type_of ‘x + 1‘;;
val it : hol_type = ‘:num‘
# type_of ‘x + 1 < x + 2‘;;
val it : hol_type = ‘:bool‘
```

The type of the terms ‘1‘ and ‘x + 1‘ is `:num`, meaning that they represent natural numbers, i.e. nonnegative whole numbers. (In mathematical syntax we would write $1 \in \mathbb{N}$ and $1 + n \in \mathbb{N}$ and to capture the information in HOL's type assignment.) On the other hand, the term ‘x + 1 < x + 2‘ is of type bool (Boolean), meaning that it is an assertion that may be true or false (in this case it happens to be true). If HOL is able to assign a type to a term, but it is not determined uniquely, a general type will be assigned automatically:

Listing 6: Automatic yype assignment

```
# ‘x‘;;
Warning: inventing type variables
val it : term = ‘x‘
# type_of it;;
val it : hol_type = ‘:?48538‘
```

One the other hand a type can be imposed on any term by writing ‘:<type>‘ after it:

Listing 7: Forced type assignment

```
# ‘x:num‘;;
val it : term = ‘x‘
# ‘x:bool‘;;
val it : term = ‘x‘
```

Variables may share the same name yet have different types and are considered to be completely different.) No annotations were needed in the composite term ‘x + 1‘ because HOL automatically allocates type ‘num‘ to the constant 1, and infers the same type for x because the two operands to the addition operator must have the same type. But you can attach type annotations to sub-terms of composite terms where necessary or simply desired for emphasis:

Listing 8: type matching

```
# ‘(x:num) = y‘;;
val it : term = ‘x = y‘
# ‘(x:num) + 1‘;;
val it : term = ‘x + 1‘
```

Theorems A boolean term may be proved true by applying a well defined set of syntactic rules and initial axioms. A special type `thm`, for theorems, is used to represent such terms. HOL Light defines a set of axioms and *inference rules*. An inference rule is an ocaml function returning something of type `thm`. For example consider the simplest inference rule, the refelexivity of equality.

Listing 9: type matching

```
# REFL ‘x:real‘;;
val it : thm = |− x = x
```

```
# let th1 = REFL `x + 1`;;
val th1 : thm = |- x + 1 = x + 1
```

A slightly more complicated primitive inference rule is INST (instantiation), which sets the variable(s) in a theorem to some particular term(s). This is a logically valid step because a HOL theorem with (free) variables holds for all values they may have:

Listing 10: type matching

```
# let th3 = INST [`2`,`x:num`] th1;;
val th3 : thm = |- 2 + 1 = 2 + 1
```

Moreover `INST`, will refuse to substitute for non-variables, which in general is not a logically valid step. For example, the fact that $2n = n + n$ does not imply that we can substitute n for $2n$ while remaining valid:

Listing 11: type matching

```
# INST [`2`,`2 * n`] th2;;
Exception: Failure "dest_var: not a variable".
```

A theorem can only be constructed by proving it. Proving non-trivial theorems at this low level is rather painful therefore, HOL Light comes with a variety of more powerful inference rules that can prove some classes of non-trivial theorems automatically. We now describe these proof methods.

Proof Methods Proofs in HOL-Light are semi-automatic which means automatic search procedures can be combined with manual proofs. Proofs in HOL Light are typically developed interactively at the OCaml toplevel. Proofs operate on goals which may be broken down into sub goals during the proof process. A *goal* captures a claim of the form $p_1, \cdots, p_n \vdash q$ that we are currently trying to prove but have not proved yet. HOL-Light keeps track of the goals and subgoals on using a *goalstack*. A tactic is a function that (essentially) takes a goal and produces a list of subgoals, such that a proof of all subgoals produces a proof of the original goal. Different proof styles are supported. Through a simple set of forward inferences, one can construct various high-level proofs. One can prove theorem in a backward fashion using tactics, use more orthodox mathematical proof style. This proof process is also simplified by the inclusion of special-purpose procedures. HOL Light already contains predefined tactics that deals with many complicated proof steps.

Example 1.5. *Suppose we want to prove the following* $x \neq 0 \Rightarrow 1 \leq x$*.*

We set up our goal with the ocaml function `g`.

Listing 12: The goal stack

```
# g `~(x=0) ==> 1 <= x`;;
Warning: Free variables in goal: x
val it : goalstack = 1 subgoal (1 total)

`~(x = 0) ==> 1 <= x`
```

Proofs take the form of natural deduction and one can introduce and eliminate assumptions with *introduction* and *elimination* rules [21]. In addition there are *simplification* and *rewrite rules*. These are equations, sometimes conditional, which can be used to substitute equal terms in a goal. Rewriting can be optionally done using higher order unification and matching [23]. Higher order matching attempts to find a common substitution for the two term, and where successful makes the appropriate substitutions. The proof process is also simplified by the inclusion of special purpose procedures. The above goal may be proved immediately with the tactic `ARITH_TAC`.

This is a tactic for proving arithmetic goals needing basic rearrangement and linear inequality reasoning only.

Listing 13: Arithmetic tactic

```
# e ARITH_TAC[];;
val it : goalstack = No subgoals
```

As another example, consider MESON, a powerful automated tool call that is very useful and convenient. Although, deciding validity in quantification theory is an undecidable problem, MESON uses an automated proof search method called 'model elimination' [33, 52] that often succeeds on valid formulas. MESON is very useful tool in automating the proof of quite intricate but essentially straightforward reasoning with quantifiers, such as in the following puzzle:

Listing 14: Theorem proving in HOL-Light

```
# MESON[]
`((?x. !y. P(x) <=> P(y)) <=> ((?x. Q(x)) <=> (!y. Q(y)))) <=>
((?x. !y. Q(x) <=> Q(y)) <=> ((?x. P(x)) <=> (!y. P(y))))`;;
val it : thm =
  |- ((?x. !y. P x <=> P y) <=> (?x. Q x) <=> (!y. Q y)) <=>
     (?x. !y. Q x <=> Q y) <=>
     (?x. P x) <=>
     (!y. P y)
```

1.2.4. HOL2P

HOL Light is suitable for a fair number of theorem proving applications. However there are limitations with regards to generic reasoning as there are no built in facility to easily manipulate type parameterisation. This means that HOL Light would not be suitable for generic program verification. As an example consider a predicate *functor* which asserts that a HOL function ϕ is a functor from the category of HOL functions to itself. From the definition of *functor* the following holds:

$$\phi\ id = id \tag{9}$$

$$\phi(f \circ g) = \phi\ f \circ \phi\ g \tag{10}$$

An example of a *functor* is the *list* map $map\_list :: (\alpha \to \beta) \to \alpha list \to \beta list$

$$map\_list\ f\ [] = [] \tag{11}$$

$$map\_list\ f\ (consxxs) = cons\ (fx)\ (map\_list\ f\ xs) \tag{12}$$

$$map\_list\ id = id \tag{13}$$

$$map\_list\ f \circ g = map\_list\ f \circ map\_list\ g \tag{14}$$

There are two problems if one tries to define a general HOL predicate *functor*:

1. From abstracting the type constructor list in the *map_list* example, one would expect functor to be parameterised with a unary type operator variable. There are no type operator variables in HOL.

2. In general, the three occurrences of ϕ in equation (2) all have different types. It is not possible to have differently typed instances of one variable in HOL.

HOL Light is therefore not expressive enough to reason about generic programs. Type quantification was added to the HOL-Light theorem prover by Voelker [54] and called HOL2P. This means that HOL2P has the necessary constructs for us to build our framework for generic reasoning. We now briefly discuss the HOL2P theorem prover.

HOL2P Type system - The type system of HOL2P extends simple typed HOL with:

- Universal types ($\Pi\alpha.T$) which bind a type variable α in a type T.
- Type variables with an arity greater than zero, these are referred to as "type operator variables"

Universal types allows for the parameterization of functions with polymorphic arguments that can be parameterized with types. The syntax of HOL2P types are as follows ($n \geq 0$):

$T ::= (\alpha :: \mathit{rank})$ - type variable (rank $=$ *large* or rank $=$ *small*)

$|\ \ (T_1, \cdots, T_n)\tau$ - type constructor application

$|\ \ (T_1, \cdots, T_n)\theta$ - type operator variable application

$|\ \ \Pi(\alpha :: \mathit{small}).T$ - universal type

The *rank* of a type is either large or small. A small type does not contain any large types and universal types. Small HOL2P types correspond to the normal HOL types.

HOL2P Terms - HOL2P extends HOL terms with type abstraction and type applications:

t ::= $(v : T)$ - variable

| $(c : T)$ - constant

| $t\ t$ - application

| $\lambda(v : T).\, t$ - abstraction

| $\Lambda\alpha.t$ - type abstraction

| $t\ [T]$ - type application

There are two important restrictions on the formation of terms:

- R1 - the type variable α must not occur freely in the type of any free variable of t in the formation of a type abstraction term $\Lambda\alpha.t$
- R2 - in a type application term $t[T]$, T must be a small type

R2 is vital to avoiding inconsistencies such as those created by Russell's paradox. Stratification of types by ranks according to the depth of universal types was implemented in $HOL - \omega$ [25]. This stratification allows terms to contain universal types of lower ranks.

A direct upshot of parameterization of terms with type operators is genericity. This expressiveness allows categorical concepts, such as initial algebras, to be applied on the level of polymorphic HOL functions. This level of abstraction is necessary for generic reasoning and is used extensively in our formalisation.

Explicit type (operator) instantiation for parsing - When entering terms that involve (possibly implicit) type operator variables, it is often necessary to add explicit type (operator) variable instantiations or type annotations; this is achieved using `TYINST`. `TYINST` is purely an auxiliary device that guides type inference. This is so because the automatic type inference and term instantiation will match a type operator variable only with a type constructor or another type operator variable.

Proofs - HOL2P theorem prover extends the proof tactics of HOL-Light with specialized tactics aim at automatically reasoning with type operator variables. For instance universal types and type operator variables in the goal can be automatically removed and assumptions placed in the list of assumptions using `TY_ALL_E_TAC`. This in essence rewrites the goal in a form that is more readily manipulated by existing HOL-Light tactics.

Listing 15: Type quantification elimination

```
# example will be inserted
```

Notation We shall omit type information when this can be determined from context. Table 4 below summaries some general notation used in this thesis. Further notation will be stated at the point of introductions:

Table 4: General Notation

English	Thesis Notation
HOL2P (OCAML) function	`sq`
HOL2P theorem proving methods	`ARITH`
Function type constructor	$\Rightarrow$
Type assignment	:
Lambda abstraction	$\lambda xyz.body$

CHAPTER 2. AUTOMATED CATEGORY-THEORY INSPIRED TYPES

The general approach used to implement recursive type definition packages, in higher order logic, is to first accept as input some informal specification of the recursive type by the user. The package then automatically generates the type definition and characteristic theorems, and adds them to the current theory. There are variants to the approach used in implementation of the type definitional packages. All the various existing approaches may be categorized as one of the following:

- **Axiomatic** The theorems and properties are generated syntactically only and introduced into the current theory as axioms. This approach is used in the PVS prover [45]
- **Inherent** The underlying logic is extended in order to support the new construct e.g. [47]
- **Definitional** The new construct is expressed in terms of existing objects and concepts, and the desired properties are derived from existing definitions and theories the Isabelle/HOL prover [41]

We introduce another approach, one based on categories. Our category theory framework, presented in Chapter (**??**), is extended to include a treatment of datatypes. Firats we review of the use of initial algebra semantics used in program construction as discussed in [38]. This serves to provide an understanding of the starting point of our automation. Datatypes are represented as algebras where the datatype is the carrier of the algebra. This algebra is interpreted categorically, that is they are viewed as functors, see section 1 example 1.1. This has the advantage over other representations as it is well suited for generic reasoning. The objectives of this chapter are to:

1. to automate the algebraic representation of recursive types from their specifications
2. to automate type functor construction using catamorphisms
3. to automatically generate datatype specific theorems from type specifications
4. to provide a rigorous formalism of the categorical initial algebra semantics as presented in [11]

The structure of this chapter is summarize below:

- **Section 2.2** Discusses singleton types as initial objects of categories
- **Section 2.3** Defines the product and coproduct type combinators
- **Section 2.4** Formalises the category of F-Algebras
- **Section 2.5** Describes the automatic generation of catamorphisms from type specification
- **Section 2.6** Discuses the Universal Extension Principle as a proof principle
- **Section 2.7** Summarizes contributions, related work and future research

2.1. Introduction

2.1.1. Program construction using initial algebras

The idea of using initiality to reason about programs can be traced back to 1969 [12] to work done by Burstall and Landin. Goguen cemented this by using initial algebras to specify the formal semantics of programming languages [18]. The project CIP evolved after and was motivated by the search of a common semantic basis for programming languages. Algebraic abstract types was used for the formulation of formal problem specifications and demonstrated with a real-life, large-scale application, the (formal) specification of the (kernel of the) program transformation system CIP-S [6].

Around the same time the notion of F-Algebras appeared in the category theory research and several researchers demonstrated the advantages of its use in program semantics in [29, 36] and later on in [31]. Hagino in his PhD thesis [19] presented a detailed construction of a categorical interpretation of a datatype. Malcom has been credited with unifying the category theory and the programming semantics research community [34]. Fokkinga has a detailed treatment of types as algebras [14] and subsequently described the categorical treatment of types that satisfies equational laws [15]. An alternative approach to types with laws is also presented in [35]. The focus has been on programming language semantics and program derivation. One noticeable omission is research on mechanized reasoning using initial algebras. We aim to fill this gap.

2.1.2. Problem statement and our goals

This chapter is geared towards specifying a suitable representation of types that for our category theory framework described in chapter (**??**). We will focus only on finite types such as `list` and `trees` which are called regular types. We will also consider regular nested types such as `rtree`. Infinite types nor types with laws will not be included in our discussion. Our focus is on automation and as such we aim to automate the initial algebraic representation, catamorphisms, type functors and associated theorems such as fusion laws. We begin with a discussion on the simplest datatype, that with just one element.

2.2. Singleton types

There is one datatype with only one element, namely, *initial object* that is of interest when specifying finite algebraic types. Our formalism of initial objects is also specified at both layers of our framework. We start our formalism with initial objects in 6-tuple categories. The initial object predicate takes two parameters, the 6-tuple representing the *layer-1* category, and an object type ($\alpha\ T$). This might not be what one would expect as an alternative would be to use a HOL2P polymorphic function as the second parameter. The main reason for using an object as this represent actual objects of the category as oppose to a class of objects represented by the a polymorphic function. This however means that when one comes to representing the algebras then one would not directly be able to represent the polymorphic list function, for instance and in that regard this definition falls short. This is not detrimental as (T) can be instantiated to a universal type when needed. An object I is initial if for each object A of **C** there is exactly one arrow of type $I \to A$.

Definition 2.1. *IOBJ1*

$(obj : (\Lambda\alpha.\ \alpha\ T \to bool),$

$hom \quad : (\Lambda\alpha\ \beta.\ (\alpha,\beta)\ H \to bool),$

$id \ : (\Lambda\alpha.\ \alpha\ T \to (\alpha,\alpha)\ H),$

$;; \ : (\Lambda\alpha\ \beta\ \gamma.\ (\alpha,\beta)\ H \to (\beta,\gamma)\ H \to (\alpha,\gamma)\ H), src : (\Lambda\alpha\ \beta.\ (\alpha,\beta)\ H \to \alpha\ T),$

$tgt : (\Lambda\alpha\ \beta.\ (\alpha,\beta)\ H \to \beta\ T))$

$(alpha \quad : \quad (\Lambda\alpha.\ \alpha\ T))$

$=$

$(\Pi\alpha.\ obj(alpha_\alpha)) \wedge$

$(\Pi\beta.\forall\ B : \beta\ T.\ obj\ b \Rightarrow (\exists!\ z : (\alpha, \beta)\ H.hom\ z \wedge src\ z = alpha_\alpha \wedge tgt\ z = \ b))$

Isomorphisms are defined as follows chapter (1) section 1.6:

Definition 2.2. *ISO1*

$(obj : \ (\Lambda\alpha.\ \alpha\ T \rightarrow bool),$

$hom \quad : \ (\Lambda\alpha\ \beta.\ (\alpha, \beta)\ H \rightarrow bool),$

$id \ : \ (\Lambda\alpha.\ \alpha\ T \rightarrow (\alpha, \alpha)\ H),$

$;; \ : (\Lambda\alpha\ \beta\ \gamma.\ (\alpha, \beta)\ H \rightarrow (\beta, \gamma)\ H \rightarrow (\alpha, \gamma)\ H), src : \ (\Lambda\alpha\ \beta.\ (\alpha, \beta)\ H \rightarrow \alpha\ T),$

$tgt : \ (\Lambda\alpha\ \beta.\ (\alpha, \beta)\ H \rightarrow \beta\ T))$

$(i \quad : \quad (\alpha,\ \beta) H)$

$=$

$hom \quad i \wedge$

$(\exists z : (\beta, \alpha) H.hom\ z \wedge src\, i = tgt\ z \wedge tgt\, i = src\, z \wedge i;;z = id(src\, i) \wedge z;;i = id(tgt\, i)))$

Initial objects are unique up to unique isomorphism.

Listing 16: Initial objects are unique up to unique isomorphism

```
val ( IOBJ1_ISOM1 ) : thm =
  |- CTGY1 (obj,hom,id,(;;),src,tgt) /\
     IOBJ1 (obj,hom,id,(;;),src,tgt) a /\
     IOBJ1 (obj,hom,id,(;;),src,tgt) b
     ==> (?h. ISOM1 (obj,hom,id,(;;),src,tgt) h /\
                src[:'A][:'B] h = a /\ tgt[:'A][:'B] h = b)
```

2.3. Type Combinators - Products and Co-products

2.3.1. Products

In category theory, the *product* of two objects A and B consists of an object and two arrows. The object is written $A \times B$ and the arrows are written $fst : A \times B \to A$ and $snd : A \times B \to B$. In HOL, products in the **FUN** are defined as `FST` and `SND` respectively and the morphism component of the product functor as `<*>`. These three things are required to satisfy the following universal property: for each pair of arrows $f : C \to A$ and $g : C \to B$ there exists an arrow $\langle f, g\rangle : C \to A \times B$ such that $fst \cdot \langle f, g\rangle = f$ and $snd \cdot \langle f, g\rangle = g$. In our formalisation below $\langle f, g\rangle$ is called `SPLIT f g`.

Listing 17: Product functor

```
val ( PROD_FUN ) : thm = |- f <*> g = (\(x,y). f x,g y)
val ( FST ) : thm = |- FST (x,y) = x
val ( SND ) : thm = |- SND (x,y) = y
val ( SPLIT_DEF ) : thm = |- SPLIT f g = (\x. f x,g x)
val ( SPLIT_CANCEL ) : thm = |- FST o SPLIT f g = f /\ SND o SPLIT f g = g
```

The following are some useful properties of products that are often used in proofs.

Listing 18: Properties of products

```
val ( PROD_ID ) : thm = |- ID <*> ID = ID
val ( PROD_FUSION ) : thm = |- f1 <*> g1 o f2 <*> g2 = (f1 o f2) <*> (g1 o g2)
val ( PROD_EQ_PROD ) : thm = |- f1 <*> g1 = f2 <*> g2 <=> f1 = f2 /\ g1 = g2
val ( PROD_CANCEL ) : thm = |- FST o f1 <*> g1 = f1 o FST /\ SND o f1 <*> g1 = g1 o SND
```

We also obtain some useful properties of `SPLIT` and formalise its relationship to products:

```
val ( SPLIT_COMP ) : thm = |- SPLIT f g o h = SPLIT (f o h) (g o h)
val ( PROD_SPLIT ) : thm = |- f <*> g = SPLIT (f o FST) (g o SND)
val ( PROD_COMP_SPLIT ) : thm = |- f1 <*> g1 o SPLIT f2 g2 = SPLIT (f1 o f2) (g1 o g2)
val ( SPLIT_FST_SND ) : thm = |- SPLIT FST SND = ID
val ( FST_AND_SND_EQ_IMP_EQ ) : thm =
     |- FST o f = FST o g /\ SND o f = SND o g ==> f = g
```

2.3.2. The Co-product

In category theory, then *Co-products* also consists of one object and two arrows for each A and B. The object is denoted $A+B$,, and the two arrows *inl* and *inr*. Given f and g, there is a unique arrow $[f,g]$ satisfying the following universal property: $[f,g] \cdot inl = f$ and $[f,g] \cdot inr = g$. *inl* and *inr*. In HOL, products in the category FUN are defined as `INL` and `INR` and in our formalisation below $[f,g]$ is called `CASE f g`.

Listing 19: Definition of `CASE` and <+>

```
let SUM_FUN = new_recursive_definition sum_RECURSION
 `(f <+> g) (INL x) = INL (f x) /\
  (f <+> g) (INR y) = INR (g y)`;;

let CASE = new_recursive_definition sum_RECURSION
   `CASE f g (INL x) = f x /\
    CASE f g (INR y) = g y`;;
```

The cancellation properties are formalised below:

Listing 20: Theorems relating <+> and `CASE`

```
let CASE_CANCEL = prove (
  `CASE f g o INL = f /\ CASE f g o INR = g`);;

let CASE_FUSION = prove (
   `f o CASE g h = CASE (f o g) (f o h)`);;
```

The relationship between `CASE` and the sum functor is formalised by the following theorems:

Listing 21: Theorems relating `CASE` and `<+>`

```
val ( SUM_CASE ) : thm = |- f <+> g = CASE (INL o f) (INR o g)


val ( CASE_COMP_SUM ) : thm =
  |- CASE f1 g1 o f2 <+> g2 = CASE (f1 o f2) (g1 o g2)
```

Listing 22: Relationship between `SPLIT` and `CASE`

```
val ( SPLIT_CASE ) : thm =
  |- SPLIT (CASE f1 g1) (CASE f2 g2) = CASE (SPLIT f1 f2) (SPLIT g1 g2)
```

2.3.3. Polynomial Functors

Functors built up from constants, products and co-products are said to be polynomial. The class of polynomial functors are defined inductively by the following,

- The identity functor id and the constant functors K_A for varying A are polynomial
- If F and G are polynomial, then so are their composition $F;;G$, their point wise sum $F+G$ and their point wise product $F \times G$. These point wise functors are defined by $(F+G)h = Fh + Gh$ and $(F \times G)h = Fh \times Gh$

2.4. Category of F-Algebras

An algebra is a endomorphism of a category, $\mathcal{C}$ of the form $\mathsf{F}A \rightarrow A$, where F is the endofunctor. This is of particular interest as algebras can be used to represent type constructors. The category of F-Algebras provides a rich playground for the modeling of primitive recursive types and primitive recursive functions. The existence of an initial F-Algebra, *alpha*, means that for any other F-Algebra $f : \mathsf{F}A \to A$, there is a unique homomorphism to f from *alpha*. This is characterized by the universal property $h = (|f|) \iff h;;alpha = f;;(\mathsf{F}\ h)$.

We begin by formulating the category of F-Algebras. Our first approach was to define this directly as a *layer-1* category. This was inevitable as the objects of the this category are a sub-set of HOL2P types, and therefore we need the type operator variable (T) to ensure that objects of this category are only F-Algebras. This approach was quite cumbersome and we decided to define the 6-tuple indirectly, using `CTGY` and a functor, F. The `FALG_CTGY` takes two parameters, a 3-tuple *layer-2* category and a *layer-2* functor F:

- *layer-2* category.
- *layer-2* functor.

Definition 2.3. *FALG_CTGY*

$(hom :\ (\Lambda\alpha\,\beta.(\alpha,\beta)\ H \to bool),$

$id :\ (\Lambda\alpha.(\alpha,\alpha)\ H),$

$;;:\ (\Lambda\alpha\,\beta\,\gamma.(\alpha,\beta)\ H \to (\beta,\gamma)\ H \to (\alpha,\gamma)\ H),$

$(F :\ (\Lambda\alpha\beta.(\alpha,\beta)\ H \to (\alpha\ F, \beta\ F)\ H)))$

$=$

$((\Pi\alpha.\ \forall a : (\alpha\ F,\alpha)\ H.\ hom\ a),$

$(\Pi\alpha\,\beta.\ \forall f : (\alpha\ F,\alpha)\ H.\forall g : (\beta\ F,\beta)\ H.\forall h : (\alpha,\beta)\ H.f;;h = (Fh);;g),$

$(\Pi\alpha.\ \forall a : (\alpha\ F,\alpha)\ H.(a,\ a,\ id_\alpha)),$

$(\Pi\alpha\,\beta\,\chi.\ \forall a : (\alpha\ F,\alpha)\ H.\forall b : (\beta\ F,\beta)\ H.\forall x : (\alpha,\beta)\ H.\forall s : (\beta\ F,\beta)\ H.\forall c : (\chi\ F,\chi)\ H.\forall y :$
$(\beta\ F,\chi)\ H.(a,\ c, x;;y)),$

$(\Pi\alpha\,\beta.\ \forall a : (\alpha\ F,\alpha)\ H.\forall b : (\beta\ F,\beta)\ H.\forall c : (\alpha,\beta)\ H.\ a),$

$(\Pi\alpha\,\beta.\,\forall a : (\alpha\,F, \alpha)\;H.\forall b : (\beta\,F, \beta)\;H.\forall c : (\alpha, \beta)\;H.\,b))$

The category of F-Algebras is defined on the 6 parameters as follows:

- Predicate $obj : \Lambda\alpha.\,\forall a : (\alpha\,F, \alpha)\;H$ ensures that only F-Algebras are objects of the sub-category. This objects are also homorphisms of the parent category

- Predicate $hom : \Lambda\alpha\,\beta.\,\forall f : (\alpha\,F, \alpha)\;H.\forall g : (\beta\,F, \beta)\;H.\forall h : (\alpha, \beta)\;H$ determines which elements of $(\alpha, \beta)\;H$ are morphisms of the sub-category F-Algebras. We are forced to use a triple to represent homomorphism as we need to make explicit the source and target algebras.

- Function $id : \Lambda\alpha.\,\forall a : (\alpha\,F, \alpha)\;H$ denotes the identity morphism.

- Operator $;; : \Lambda\alpha\beta\chi.\forall a : (\alpha\,F, \alpha)\,H.\forall b : (\beta\,F, \beta)\,H.\forall x : (\alpha, \beta)\,H.\forall s : (\beta\,F, \beta)\,H.\forall c : (\chi\,F, \chi)\;H.\forall y : (\beta\,F, \chi)\;H$ denotes morphism composition.

- Function $src : \Lambda\alpha\,\beta.\,\forall a : (\alpha\,F, \alpha)\;H.\forall b : (\beta\,F, \beta)\;H.\forall c : (\alpha, \beta)\;H$ associates a morphism with its source object.

- Function $tgt : \Lambda\alpha\,\beta.\,\forall a : (\alpha\,F, \alpha)\;H.\forall b : (\beta\,F, \beta)\;H.\forall c : (\alpha, \beta)\;H$ associates a morphism with its target object.

We now define the universal property homomorphisms. The all important catamorphism is defined as a homomorphism from the initial algebra.

Listing 23: Definition of homomorphism

```
val ( FHOMO_THM ) : thm =
  |- !hom id g (;;) F h f.
          FHOMO_THM (hom,id,(;;)) F (g,f,h) <=>
          (;;)[:('A)_F][:'A][:'B] g h =
          (;;)[:('A)_F][:('B)_F][:'B] (F[:'A][:'B] h) f
```

If `f` is a homomorphism from `alpha` then `f` is a catamorphism. Catmorphisms uses hilberts choice operator to select some homomorphism that statisfies the universal property above, where g is the initial algebra.

Listing 24: Definition of catamorphisms

```
val ( CATA ) : thm =
  |- !hom id (;;) f F alpha.
          cata (hom,id,(;;)) F alpha f =
          (@h. (;;)[:('Z)_F][:'Z][:'Y] alpha h =
               (;;)[:('Z)_F][:('Y)_F][:'Y] (F[:'Z][:'Y] h) f /\
               hom[:'Z][:'Y] h)
```

When insantated with **Fun** we have the following theorem

Listing 25: Definition of catamorphisms in **Fun**

```
# val ( CATA_FUN ) : thm =
  |- !f F alpha. cata CATFUN F alpha f = (@h. h o alpha = f o F[:'Z][:'Y] h)
```

We also need to define what it means for an algebra to be initial in the category of F-Algebras.

Listing 26: Initial algebras

```
val ( IALG ) : thm =
  |- !hom id (;;) F alpha.
          IALG (hom,id,(;;)) F alpha <=>
          hom[:('Z)_F][:'Z] alpha /\
          (!! 'B. !f. hom[:('B)_F][:'B] f
                    ==> (?!h. (;;)[:('Z)_F][:'Z][:'B] alpha h =
                              (;;)[:('Z)_F][:('B)_F][:'B] (F[:'Z][:'B] h) f /\
                              hom[:'Z][:'B] h))
```

This is also instantiated to the **Fun**.

```
# IALG_FUN;;
val it : thm =
  |- !F alpha.
          IALG CATFUN F alpha <=>
          (!! 'B. !f. ?!h. h o alpha = f o F[:'Z][:'B] h)
```

We can now proceed to verify the universal property, we show that $alpha;;h \Rightarrow (\mathsf{F}\ h);;f \Rightarrow h = (|f|)$

Listing 27: Universal property of catamorphism

```
val ( EQ_CATA ) : thm =
  |- CTGY (hom,id,(;;)) /\
      FNCTR (hom,id,(;;)) F /\
      IALG (hom,id,(;;)) F alpha /\
      hom[:('G)_F][:'G] f /\
      hom[:'T][:'G] h /\
      (;;)[:('T)_F][:'T][:'G] alpha h =
      (;;)[:('T)_F][:('G)_F][:'G] (F[:'T][:'G] h) f
      ==> cata (hom,id,(;;)) F alpha f = h
```

And this is also instantiated to the **Fun**

Listing 28: EQ_CATA instantatied to

```
# EQ_CATA_FUN;;
val it : thm =
  |- !F alpha f h.
          FNCTR CATFUN F /\
          IALG CATFUN F alpha /\
h o alpha = f o F[:'T][:'G] h
==> cata CATFUN F alpha f = h
```

Proof of the contrapositive of the universal property of catamorphisms is given as

Listing 29: Universal property of catamorphism

```
# CATA_IALG;;
val it : thm =
  |- IALG (hom,id,(;;)) F alpha /\ hom[:('B)_F][:'B] f
     ==> hom[:'T][:'B] (cata (hom,id,(;;)) F alpha f) /\
         (;;)[:('T)_F][:'T][:'B] alpha (cata (hom,id,(;;)) F alpha f) =
         (;;)[:('T)_F][:('B)_F][:'B]
         (F[:'T][:'B] (cata (hom,id,(;;)) F alpha f))f



# CATA_IALG_FUN;;
val it : thm =
  |- !F alpha f.
         IALG CATFUN F alpha
         ==> cata CATFUN F alpha f o alpha =
             f o F[:'T][:'B] (cata CATFUN F alpha f)
```

2.5. Deriving catamorphisms from type specifications

The general specification in HOL for inductive datatypes is of the general form:

$$(\alpha_1, \cdots, \alpha_n)rty ::= C_1 ty_1^1 \cdots ty_1^{k_1} | \cdots | C_m ty_m^1 \cdots ty_m^{k_m} \tag{15}$$

where the α_i are type variables and rty is the name of the type constant or type operator being defined , constructors C_i are distinct, and ty_m^k, are admissible types containing at most the type variables $(\alpha_1, \cdots, \alpha_n)$. Now ty^k_m is admissible if ty^k_m is one of the following:

- An existing non-recursive type
- A recursive occurrence
- A nested recursion involving existing types

In HOL-Light Harrison's defined a ML function, `define_type` that automatically define user-specified recursive types [23]. This function takes as input an informal type specification. in the form of equation (15). A string of this form describes an n-ary type operator rty; if n is zero then rty is a type constant. Each constructor C_i are identifiers whose arguments are types, ty^k_m. The type ty^k_m is either a (logical) type expression valid in the current theory, in which case ty^k_m must not contain $(\alpha_1, \cdots, \alpha_n)rty)$, or just the identifier $(\alpha_1, \cdots, \alpha_n)rty$ itself. If one or more of the type expressions ty^k_m is the type $(\alpha_1, \cdots, \alpha_n)rty$ itself, then the equation specifies a recursive data type. In any specification, at least one constructor must be non-recursive, a base case, i.e. all its arguments must have types which already exist in the current theory. Each of the types ty^k_m above may be built from the type being defined using other recursive type operators already defined, e.g. `list`. Moreover, one can actually have a mutually recursive family of types, where the format is a sequence of specifications in the above form separated by semicolons:

$$\begin{aligned}
op1 &= C1_1 ty...ty | C1_2 ty...ty | ... | C1n_1 ty...ty; \\
op2 &= C2_1 ty...ty | ... | C2_n 2ty...ty; \\
&\vdots \\
opk &= Ck_1 ty...ty | ... | ... | Ck_n kty...ty
\end{aligned}$$

Before including the new type in the current theory, the ML function checks if the user input string respects the rules stated above. Once the type specification is checked and is correct the corresponding type definition for the type operator or type operators is added to the current type theory (or type context, Γ). It makes appropriate definitions for the constructors C_i and automatically proves and returns two theorems, inductive and recursive theorems. Roughly, the first theorem allows one to prove properties over the new (family of) types by (mutual) induction, while the latter allows one to defined functions by recursion.

There are two general approaches used in the implementation of recursive datatypes packages in HOL provers. One developed by Melham [40] and the other is based on Knsater-Tarski's fix point theorem [2, 46].

Now that we have defined a catamorphism, the process of deriving the catamorphism for each specific datatype becomes a matter of putting the correct pieces together. What this means is that we need to derive the initial algebra and the corresponding base functor for each specified type. We also need to derive the correct type instantiation for the functor type operator variable F. Once we have these pieces then we can instantiate the general theorem to automatically derive the catamorphism of each initial type. The catamorphim is then used to automatically derive the type functor.

We now describe the process of automatically deriving the catamorphism for each type, and type functors, from the type specification.

2.5.1. Deriving initial algebras

Before we demonstrate the process of deriving initial algebras we give some examples. The most common recursive type is the `list` type, and this is defined as follows:

Listing 30: `list` type specification

```
let list_INDUCT, list_RECURSION =
 define_type "list = NULL | CONS A list";;
```

This declares `[NULL, CONS]` of type : $\mathsf{F}_A\ \mathit{list}\ A \to \mathit{list}\ A$ to be an initial algebra. The `list` base functor denoted F_A is defined by $\mathsf{F}_A(B) = 1 + (A \times B)$ and $\mathsf{F}_A(f) = id_1 + (id_A \times f)$, a pair of morphisms, one on the objects and the other on morphisms. The `list` type is parameterised with only one type variable A. We can also declare types with more than one type variables.

Listing 31: Definition of recursive type with more than one type variable (R

```
let (ex1_INDUCTION, ex1_RECURSION) =
  define_type "ex1 = C1 R | C2 S ex1";;
```

Datatypes may also be defined using previously defined datatypes and in these cases they are called *nested types*. There are two types of nesting *non-regular* and *regular* nested types. A *non-regular nested* datatype is a parametrised datatype whose declaration involves different instances of the accompanying type parameters, for example `bush A = BNUL | BCONS A bush(bush A)` [9]. These types are not currently admissible in HOL-Light. The *regular datatype* on the other hand where the nested datatype is parametrised with an instance of the currently defined recursive type. This is permissible, and the example of the rose tree is defined.

Listing 32: `rtree` type specification

```
let rtree_INDUCT, rtree_RECURSION =
  define_type "rtree = NODE_RTREE A (rtree)list";;
```

We only discuss regular nested types in this thesis and may often drop the "regular". Harrison already checks that the type specification is admissible and creates the constructors as constants in the current theory. We use these constants to automatically derive the initial algebraic representation. The initial algebra is expressed, as a case wise polynomial expression over these constructors, that is $C_1 + C_2 \cdots + C_n$. We however have to ensure that the expression is in a form suitable for `CASE` as defined in our framework see Listing 19. We use the following method to achieve this:

```
function FORMAT(constructors)
    for each constructor do
        if constructor is nullary then
            define a lambda abstraction, parameterised with a variable of type : 1
        else
            for each constructor argument do
                create a new appropriately typed variable $a_n$
            end for
            create a lambda abstraction (curried) over the variables $a_1 \cdots a_n$
        end if
    end for
end function
```

The formatted constructors are the used as parameters for the `CASE`. Therefore the initial algebra for the `list` algebra is given below.

Listing 33: `list` algebra

```
# fst (create_initial_alpha `:('A)list`);;
  val it : term * hol_type = (`CASE (\e:1. []) (\(a0,a1). CONS a0 a1)`
```

This algorithm also correctly derives the initial algebra for types parameterised with more than one type variable.

Listing 34: `ex1` algebra

```
# fst (create_initial_alpha `:('R,'S)ex1`);;
val it : term = `CASE (\a. C1 a) (\(a0,a1). C2 a0 a1)`
```

As well as for nested types

Listing 35: `rtree` algebra

```
# fst (create_initial_alpha `:('A)rtree`);;
val it : term * hol_type = `\(a0,a1). NODE_RTREE a0 a1`
```

To verify the initiallity condition for these types we must prove that they satisfy the the initial algebra theorem. This states that for all morphisms f there exist a unique morphism h from `list_ALPHA` to f. This theorem is given below for the case of `list`

Listing 36: Initial algebra theorem for list

```
# IALG_THM `:('A)list`;;
val it : thm =
  |- IALG CATFUN list_IFUN list_ALPHA <=>
     (!! 'B. !f. ?!h. h o list_ALPHA = f o ID <+> ID <*> h)
```

We can show that each admissible type is initial in the category of F-Algebra. The proof of this is based on simplification using the base functor and the universal property of catamorphisms along with some standard first order logic that is handled by MESON_TAC.

Listing 37: list is initial

```
# ALPHA_IS_INITAL `:('A)list`;;
val it : thm = |- IALG CATFUN list_IFUN list_ALPHA
```

2.5.2. Deriving base functors

The generic method create_initial_alpha returns a pair, a HOL term and a HOL type. The second argument a HOL2P type and models the base functor mapping on objects. This provides the type instantiation for the type operator variable (F). We discuss the automatic derivation of first the object component of the base functor followed by the morphism component. As an illustrative example, consider the derivation the object mapping of the base functor for list given below

Listing 38: list base functor object mapping

```
# snd (create_initial_alpha `:('A)list`);;
  val it : term * hol_type = (`:(% 'T .1+'A1#'T)`)
```

This is derived using the following procedure:

```
function BASEFUNCTOROBJECTMAPPING(constructors)
    for each constructor do
        if constructor is nullary then
            type of the functor expression is : 1
        else
            for each constructor argument do
                if type variable say : $A$ then
                    create a new free type variable say : $A1$
            end if
            if recursive type then
                create a new bond type variable say : $T$
            end if
        end for
    end if
  end for
end function
```

Applying this procedure to our example with more than one type variable results in a newly created type variable each type variable, for example.

Listing 39: `ex1` base functor object mapping

```
# snd (create_initial_alpha `:('R,'S)ex1`);;
val it : hol_type = `:(% 'T .'R1+'S1#'T)`
```

One notices a slight complication, the presence of a nested recursion. As we are only considering simple nesting, these types are always parameterised with the recursive type that is currently being defined. Therefore by rewrite rule (**??**) we obtain the following additional rule.

- If type of argument is a nested type say ty then create a new type $(T)ty$

We can able to derive the base functor for our `rose` tree example

Listing 40: `rtree` base functor object mapping

```
# fst (create_initial_alpha `:('A)rtree`);;
  val it : term * hol_type = `:(% 'T .'A1#('T)list)`)
```

The final result is then type that is universally quantified over the currently defined recursive type. We can now formally state the procedure for the derivation of the object mapping of the type base functor.

```
function baseFunctorObjectMapping(constructors)
    for each constructor do
        if constructor is nullary then
            type of the functor expression is : 1
        else
            for each constructor argument do
                if type variable say : $A$ then
                    create a new free type variable say : $A1$
                end if
                if recursive type then
                    create a new bond type variable say : $T$
                end if
                if nested type say $ty$ then
                    create a new type $(T)ty$
                end if
            end for
        end if
    end for
end function
```

The morphism mapping of the base functor is also very important as it encodes the type's structural information. This is derived by the generic method `create_base_functor`. This method returns a pair of HOL2P terms, the first of which is the base functor.

Listing 41: `list` base functor morphsim mapping

```
# fst(create_base_functor `:('A)list`);;
val it : term * term = `(\\ 'B 'C. (\f. (ID:1->1) <+> (ID:'A->'A) <*> (f:'B->'C)))`
```

```
function baseFunctorMorphismMapping(constructors)
    for each constructor do
        if constructor is nullary then
            type of the functor morphsim $id : 1 \to 1$
        else
            for each constructor argument do
                if type variable say : $A$ then
                    create a morphsim $id : A \to A$
                end if
                if recursive type then
                    create a morphsim $f : B \to C$
                end if
            end for
        end if
    end for
end function
```

This procedure is also sufficient for deriving the base functor for types with more than one type variable as shown in our example below.

Listing 42: `ex1` base functor morphsim mapping

```
# fst (create_base_functor `:('R,'S)ex1`);;
val it : term = `(\\ 'B 'C. (\f. ID <+> ID <*> f))`
```

This method however fails in the case of nested types. In this instance the morphism is actually embedded into the structure of the nested type. We will therefore need the corresponding a nested type functor before we can map of this morphism. Therefore our procedure must be modified with the following case:

- If (nested type) ty then create a morphism $ty_{map}\ f$

Listing 43: `rtree` base functor morphsim mapping

```
# fst (create_base_functor `:('A)rtree`);;
val it : term * term = `(\\ 'B 'C. (\f. ID <*> list_TYPE_FUNCTOR f))`
```

It is noted that the type functor derivation has not yet been discussed. At this point it is sufficient to note that the type functor is automatically generated when the algebraic `list` type is defined, see section (2.5.4) for further clarification. We can note state the process used to automatically derive the morphism mappings of the base functors.

```
function baseFunctorMorphismMapping(constructors)
    for each constructor do
        if constructor is nullary then
            type of the functor morphsim $id : 1 \to 1$
        else
            for each constructor argument do
                if type variable say : $A$ then
                    create a morphsim $id : A \to A$
                end if
                if recursive type then
                    create a morphsim $f : B \to C$
                end if
                if nested type say $ty$ then
                    create a morphsim $ty_{map}\ f : B \to C$
                end if
            end for
        end if
    end for
end function
```

We also show that the base functor is a funcor and with some renaming have the following generic theorem.

Listing 44: `list` base functor is a functor

```
# BASEFUN_FUNCTOR `:('A)list`;;
val it : thm = |- FNCTR CATFUN list_IFUN
```

2.5.3. Catamorphism

The derivation of the catamorphism for each type now becomes a matter of instantiating definitional axiom in listing (25), initial algebra, `alpha`, and its associated base functor. These instantiations are carried out by the method `CATA_DEF`. This method actually returns a pair of HOl2P terms and the first of this which is of interest.

Listing 45: Definition of catamorphisms

```
# fst(CATA_DEF `:('A)list`);;
val it : term =
  (`cata CATFUN (\\ 'B 'C. (\f. ID <+> ID <*> f)) (CASE (\e. []) (\(a0,a1). CONS a0 a1)) f`
```

By means of some renaming we are able to rewrite the above in a form that is more human readable.

Listing 46: catamorphisms

```
# CATA_THM `:('A)list`;;
val it : thm = |- list_CATA = (\f. cata CATFUN list_IFUN list_ALPHA f)
```

And to reinforce what we have just done we shall use the `list_CATA` to write the sum function for `list`.

Listing 47: catamorphisms

```
# let list_SUM = `list_CATA (CASE (\e:1.0) (UNCURRY (+)))`;;
val list_SUM : term = `list_CATA (CASE (\e. 0) (UNCURRY (+)))`
# type_of list_SUM ;;
val it : hol_type = `:(num)list->num`
```

Some programmers might find it impossible to write programs categorically. Fortunatly the catamorphic definition of a recursive function can be automatically derived and this is dicussed of Chapter (**??**).

There are some powerful theorems that can be derived from catamorphism. Take for instance the reflection law. This theorem states that the catamorphism of an initial algebra is equivalent to the identity morphism.

Listing 48: The reflection law

```
val ( REFLECTION_LAW ) : thm =
  |- !hom id (;;) alpha F.
          CTGY (hom,id,(;;)) /\
          FNCTR (hom,id,(;;)) F /\
          IALG (hom,id,(;;)) F alpha
          ==> cata (hom,id,(;;)) F alpha alpha = id[:'T]
```

When instantiated to the **Fun** this is simplified to the following.

Listing 49: The fusion law

```
val ( REFLECTION_LAW ) : thm =
  |- !alpha F.
          FNCTR CATFUN F /\ IALG CATFUN F alpha
          ==> cata CATFUN F alpha alpha = (\r. r)
```

Now for each type we have shown that is alpha is initial and that the base functor is a functor. It follows that by instantating the following and further simplification we are able to automatically derive the reflection law for each type. This can then be used in proofs generically by type parameterisation.

Listing 50: The fusion law

```
# REFLECTION_DEF `:('A)list`;;
val it : thm = |- list_CATA list_ALPHA = (\r. r)
```

2.5.4. Deriving type functors

Type functors are derived using catamorphisms. Previously we introduced base functors denote $\mathsf{F}_A(B)$. In this definition A is fixed on the definiton. If we parameterise A then we can write $\mathsf{F}(A,B)$, in which case F is a bi-functor with the collection of initial algebras $\alpha_A : \mathsf{F}(A, \mathsf{T}\ A) \to \mathsf{T}A$. The type functor T is defined by $\mathsf{T}f = (\!|\alpha;;\mathsf{F}(f,id)|\!)$. This means that once the bi-functor is defined for each type then the type functor definition will follow.

Deriving bi-functors We mentioned earlier that `create_base_functor` returned a pair of HOL2P terms the first of which is the morphism mapping of the base functor. The second term is the base bifunctor. This is formed by a lambda abstraction the identity morphisms on the type variables in addition to those in the functor. Therefore the `list` bi-functor is given by

Listing 51: `list` base bi-functor type functor

```
# snd (create_base_functor `:('A)list`);;
val it : term = `(\\ 'A' 'B 'A 'C. (\f0 f. ID <+> (f0:'A->'A) <*> f))`
```

Recall that each type variable resulted in a corresponding identity morphism and therefore if the following example where there are two type variables there is actually three parameters.

Listing 52: `list` base bi-functor type functor

```
# snd (create_base_functor `:('R, 'S)ex1`);;
val it : term = `(\\ 'R 'R' 'C 'S 'B 'S'. (\f1 f0 f. (f0:'R->'R) <+> (f1:'S->'S) <*> f))`
```

Bi-functors in the case of `rtree` also follows directly from this additional parameterisation.

Listing 53: `list` base bi-functor type functor

```
# snd (create_base_functor `:('A)rtree`);;
val it : term = `(\\ 'A' 'B 'A 'C. (\f0 f. (f0:'A -> 'A) <*> list_TYPE_FUNCTOR f))`
```

Deriving type functors Let F be a bi-functor with the collection of algebras $\alpha_A : \mathsf{F}(A, \mathsf{T}\,A) \rightarrow \mathsf{T}A$. The construction of the map functor is defined by

$$\mathsf{T}\,f = (|\alpha \circ \mathsf{F}(f, id)|) \tag{16}$$

Listing 54: `list` type functor

```
val it : thm =
  |– list_TYPE_FUNCTOR =
     (\f0. list_CATA
           (list_ALPHA o list_BIFUN[:'A'][:('A')list][:'A][:('A')list] f0 ID))
```

Listing 55: `ex1` type functor

```
# TYPE_FUNCTOR `:('R,'S)ex1`;;
val it : thm =
  |– ex1_TYPE_FUNCTOR =
     (\f1 f0.
          ex1_CATA
          (ex1_ALPHA o ex1_BIFUN[:'R][:'R'][:('R','S')ex1][:'S][:('R','S')ex1][:'S'] f1 f0 ID))
```

Listing 56: `rtree` type functor

```
# TYPE_FUNCTOR `:('A)rtree`;;
val it : thm =
  |– rtree_TYPE_FUNCTOR = (\f0. rtree_CATA (rtree_ALPHA o rtree_BIFUN f0 ID))
```

To prove that the type functor is a functor we must prove the following:

Listing 57: `list` type functor is a functor

```
# TYPE_FUNCTOR_FUNCTOR_THM `:('A)list`;;
val it : thm =
  |– FNCTR CATFUN (\\ 'A 'A'. list_TYPE_FUNCTOR) <=>
     (!! 'A. list_TYPE_FUNCTOR ID = ID) /\
     (!! 'A 'B 'C. !f g.
                        list_TYPE_FUNCTOR (g o f) =
                        list_TYPE_FUNCTOR g o list_TYPE_FUNCTOR f)
```

To prove this we instantiate the `FNCTR_ID_FUN` and theorem with the type functor and we obtain the following

Listing 58: `list` type functor preserves identity

```
# TYPE_FUNCTOR_ID_THM `:('A)list` ;;
val it : thm =
  |- FNCTR CATFUN (\\ 'A 'A'. list_TYPE_FUNCTOR)
     ==> (!! 'A. list_TYPE_FUNCTOR ID = ID)
```

We also instantiate the corresponding `FNCTR_COMP_FUN` theorem.

Listing 59: `list` type functor preserves composition

```
# TYPE_FUNCTOR_COMP_THM `:('A)list`;;
val it : thm =
  |- FNCTR CATFUN (\\ 'A 'A'. list_TYPE_FUNCTOR)
     ==> (!! 'A 'B 'C. !f g.
                          list_TYPE_FUNCTOR (g o f) =
                          list_TYPE_FUNCTOR g o list_TYPE_FUNCTOR f)
```

The proof that the type functor is a functor is a matter of simplification using the two theorems above.

Listing 60: `list` type functor preserves composition

```
# TYPE_FUNCTOR_IS_FUNCTOR_THM `:('A)list`;;
val it : thm = |- FNCTR CATFUN (\\ 'A 'A'. list_TYPE_FUNCTOR)
```

2.6. Universal Extension Principle

To introduce the universality principle we will start off with a recursively defined datatype that does not use type variables. We will speak only of a simple recursive datatype formed by the naturals and called `num` and its corresponding `fold`. The `num` datatype is defined by:

```
data num = zero | succ num
```

This introduces a type `num` that is parameterized with two constructors: a nullary constructor `zero` and the constructor `succ` that takes one argument. The argument to `succ` is a non-negative number `n`. This is recursively defined as the argument is the same type as the type being declared.

Take for instance the well-known function `sum` , which sums two numbers.

sum : num $\to num \to num$

$sum\ (m, 0) = m\ \wedge$

$sum\ (m, (succ\ n)) = succ(sum(m, n))$

This recursive definition reads quite easily. It says, whenever the number m is `sum` with `zero`, the result is m. Whenever m is `sum` with the `succ` of n the result is the `succ` of the `sum(m,n)`. The function `sum` is an instance of a general family of functions that may defined on `num`. If we abstract over the general pattern we have the following equations, parameterized by a recursively defined function F,

$$\Delta(F) =: F \circ 0 = z\ \wedge F \circ succ = s \circ F \tag{17}$$

F is parameterized by the constructors of `num`. Notice the introduction of two function variables; a variable z and a unary function variable $s : num \to num$. In the instance of `sum` . These specific bindings of (s, z) for `sum` in the categorical sense represent a homomorphism from the initial algebra `num` to the algebra `sum`. Furthermore, lets say another function `plus` was defined with these bindings z and $s = (\lambda v.\ succ\ v)$ then the functions `sum` and `plus` are equal.

Now this may be abstracted one step further. The function variables, z and $s : A \to A$ where A is some type. Given bindings for s and z, a function satisfying Π is a homomorphism from the initial algebra $(Fnum, num)$ of the functor $FA = 1 + A$ with signature $(succ, 0)$ to the algebra on $(\mathsf{F}B, B)$ with signature (s, z). Since the algebra of naturals is defined as the initial algebra in this category, there exists- by the definition of "initial" - exactly one such homomorphism for each choice of (s, z). Recall that a unique homomorphism is called a catamorphism. Therefore this is a means for defining functions on the naturals. Moreover, given two functions $f, g : num \to A$, the following holds

$$\Delta(f) = \Delta(g) \Rightarrow f = g \tag{18}$$

This is the *Unique Extension Property* for the naturals. We now use induction to verify this property

Proof. base case:

f(0) $=$ g (0)

$\equiv \{\Pi(f) \; and \; \Pi(g)\}$

$z = z$

$\equiv \{reflexivity \; of \; (=)\}$

$true$

inductive case:

f $\circ$ $succ \circ n = g \circ succ \circ n$

$\equiv \{\Pi(f) \; and \; \Pi(g)\}$

$s \circ f \circ n = s \circ g \circ n$

$\Leftarrow \{Leibniz\}$

$f \circ n = g \circ n$

$\equiv \{Induction\ Hypothesis\}$

$true$

$\square$

The list fold is common and is defined as standard in many functional programming languages. Harrison also defines list `fold` as apart of HOL-Light. Below we describe a method to automatically derive the fold combinator from the structure of the recursive type.

2.6.1. Deriving `fold` combinator from primitive recursion

The above sum example was used to present an informal guide to the UEP. The keen reader would also be quick to point out the informal use of `fold` in the presentation. We now describe the automatic derivation of the fold combinator from the recursion theorem. The general specification in HOL for recursive datatypes is of the general form **??**: The general recursion theorem is given by the following:

$\forall f_1 \cdots f_n . \exists! fn : (\alpha_1, \cdots, \alpha_n) rty \to bool.$

$\forall x_1^1 \cdots x_1^{k_1} . fn(C_1 x_1^1 \cdots x_1^{k_1}) = f_1(fn x_1^1 \cdots (fn x_1^{k_1}) x_1^1 \cdots x_1^{k_1} \wedge$

$\vdots$

$\forall x_m^1 \cdots x_m^{k_m} . fn(C_m x_m^1 \cdots x_m^{k_m}) = f_m(fn x_m^1 \cdots (fn x^{k_m}) x_m^1 \cdots x_m^{k_m} \wedge$

In order to derive the `fold` operator from the recursion theorem we first reduce the above to Skolem normal form, in order to remove the existential quantification of the fold operator, this is performed as the first step in the automation of

$fold$ definition from the recursion theorem. As an example, the logical statement $\forall x \exists y \forall z.P(x, y, z)$ is not in Skolem normal form because it contains the existential quantifier $\exists y$. The process of reducing this formula to skolem normal form replaces y with $f(x)$, where f is a new function symbol, and removes the quantification over y. That is $\forall x \forall z.P(x, f(x), z)$ "for every x there exists a y such that P(x,y,z)" is converted into the equivalent form "there exists a function f mapping every x into a y such that, for every x it holds R(x,f(x),z)". The result of skolemizing the recursion theorem is shown below

$$
\begin{aligned}
&\exists fn.\\
&\forall f_1 \cdots f_n.\\
&\forall x_1^1 \cdots x_1^{k_1}.fnf_1 \cdots f_n(C_1 x_1^1 \cdots x_1^{k_1}) = f_1(fnx_1^1 \cdots (fnx_1^{k_1})x_1^1 \cdots x_1^{k_1} \wedge\\
&\vdots\\
&\forall x_m^1 \cdots x_m^{k_m}.fnf_1 \cdots f_n(C_m x_m^1 \cdots x_m^{k_m}) = f_m(fnx_m^1 \cdots (fnx^{k_m})x_m^1 \cdots x_m^{k_m} \wedge
\end{aligned}
$$

We can now specialize the fn and from here on refer to this as $fold_\alpha$ thus removing the existential quantification, and also the universal quantifiers.

$$
\begin{aligned}
&fold_{(\alpha_1,\ldots,\alpha_n)rty} f_1 \cdots f_n(C_1 x_1^1 \cdots x_1^{k_1}) = f_1(fnx_1^1 \cdots (fnx_1^{k_1})x_1^1 \cdots x_1^{k_1} \wedge\\
&\vdots\\
&fold_{(\alpha_1,\ldots,\alpha_n)rty} f_1 \cdots f_n(C_m x_m^1 \cdots x_m^{k_m}) = f_m(fnx_m^1 \cdots (fnx^{k_m})x_m^1 \cdots x_m^{k_m} \wedge
\end{aligned}
$$

This form of full primitive recursion, is called paramorphisms in the category world. We first focus on the more popular catamorphic form and describe how to derive the a special class of paramorphic functions, catamorphisms. The definition of the $fold$ combinator from the corresponding recursion theorem is generated from by `FOLD_DEF` parameterized by the recursive type. For instance the following is 61 shows the *fold* combinator for lists.

Listing 61: `list` fold from primitive recursion

```
# FOLD_DEF `:('A)list`;;
val it : thm =
|- list_FOLD NIL' CONS' [] = NIL' /\
   (!a0 a1. list_FOLD NIL' CONS' (CONS a0 a1) =
            CONS' a0 (list_FOLD NIL' CONS' a1))
```

The *fold* combinator for a binary tree is similarly generated by a function call to `FOLD_DEF`.

Listing 62: `ex1` fold from primitive recursion

```
# FOLD_DEF `:('R, 'S)ex1`;;
val it : thm =
  |- (!a. ex1_FOLD f0 f1 (C1 a) = f0 a) /\
     (!a0 a1. ex1_FOLD f0 f1 (C2 a0 a1) = f1 a0 (ex1_FOLD f0 f1 a1))
```

Listing 63: `rtree` fold from primitive recursion

```
# FOLD_DEF `:('A)rtree`;;
val it : thm =
  |- (!a0 a1.
          rtree_FOLD f0 f1 f2 (NODE_RTREE a0 a1) =
                f0 a0 (rtree_list_FOLD f0 f1 f2 a1)) /\
     rtree_list_FOLD f0 f1 f2 [] = f1 /\
     (!a0 a1.
          rtree_list_FOLD f0 f1 f2 (CONS a0 a1) =
                f2 (rtree_FOLD f0 f1 f2 a0) (rtree_list_FOLD f0 f1 f2 a1))
```

2.6.2. Universality of fold

The universal property of $fold$ can be stated as the following equivalence of the recursive definition and $fold_(\alpha_1,,\alpha_n)rty$

Theorem 2.1. *The characteristic equation, i.e the recursion theorem is equivalent to* $fold_(\alpha_1,,\alpha_n)rty$

$$h(C_1 x_1^1 \cdots x_1^{k_1}) = f_1(fnx_1^1 \cdots (fnx_1^{k_1})x_1^1 \cdots x_1^{k_1} \wedge$$
$$h(C_2 x_2^1 \cdots x_2^{k_2}) = f_2(fnx_2^1 \cdots (fnx_1^{k_2})x_2^1 \cdots x_2^{k_2} \wedge$$
$$\vdots$$
$$h(C_m x_m^1 \cdots x_m^{k_m}) = f_m(fnx_m^1 \cdots (fnx^{k_m})x_m^1 \cdots x_m^{k_m} \wedge$$

$$\Leftrightarrow$$

$$h = fold \quad f_1\ f_2 \cdots f_m$$

The proof is conceptually simple and based in a proof by induction [10]. If one examines closely, this process converts the recursion into two explicit assumptions; the inductive cases. Therefore by verifying these two assumptions, and this need not be by induction, we are able to capture the inductive proof process once, and reuse it many times. One can say the universal principle is the counterpart to `fold`. The fold operator encapsulates recursion on recursive types; in the same manner the universal principle encapsulates proofs by induction on recursive types.

We generalise this proof by parameterizing the recursive type in the induction tac. The universal

Listing 64: Universality of fold for list

```
# FOLD_UNIVERSAL `:('A)list`;;
val it : thm =
|- h [] = NIL' /\
  (!a0 a1. h (CONS a0 a1) = CONS' a0 (h a1)) <=>
  h = list_FOLD NIL' CONS'
```

Listing 65: Universality of fold for list

```
# FOLD_UNIVERSAL `:('R, 'S)ex1`;;
val it : thm =
  |- (!a. h (C1 a) = f0 a) /\ (!a0 a1. h (C2 a0 a1) = f1 a0 (h a1)) <=>
     h = ex1_FOLD f0 f1
```

2.6.3. Universality of catamorphisms

There is a relationship between the parameters to fold and the casewise expression that parameterises catamorphism. Informally each f_i of fold corresponds to the $i+1$ expression of `CASE`. So for instance with `list` the following follows:

Theorem 2.2. $h \circ (\alpha_1 \cdots \alpha_n)rty = f \circ F^{\mathcal{C}}_{(\alpha_1 \cdots \alpha_n)rty}(id, h) \iff$
$h = fold^{\mathcal{C}}_{(\alpha_1 \cdots \alpha_n)rty}\ ((f \circ inl)x^1_1 \cdots x^{k_1}_1)\ ((f \circ inr \circ inl)x^1_2 \cdots x^{k_2}_2) \cdots ((f \circ inr)x^1_n \cdots x^{k_n}_n)$

This theorem states that for any arbitrary initial algebraic type the function *fold* $f_1\ f_2 \cdots f_n$ is a unique solution to the defining catamorphism. This theorem underpins the `CATA_UNIVERSAL` method, which is automatically generated by our datatype package.

Listing 66: list catamorphism universal principle

```
# CATA_UNIVERSAL `:('A)list`;;
val it : thm =
  |- h o list_ALPHA = f o list_IFUN <=>
     h = list_FOLD ((f o INL) one) (\a0 a1. (f o INR) (a0,a1))
```

The main use of the universal property is as a proof principle of fold as it encapsulates a common pattern of induction. As well as being an alternative for recursive proofs the universal extension property can also be used to guide the transformation of recursive functions to their equivalent catamorphic form. We describe the automatic derivation of catamorphisms from the primitive recursion theorem [20]

2.7. Conclusions

We first start with the simplest datatype, one with only one element then proceed to briefly discuss the tupling of existing datatypes to form new types. This is achieved by taking the product and sum of existing types.

We use a paradigm of datatype definition attributed to Hagino [19], an important aspect of which is that datatypes are characterized by a universal property. We can use this property, called *catamorphisms*, as a definitional property of recursive functions. Catamorphisms play a prominent role in the theory of datatypes used in this paper. The datatypes that can be defined using this paradigm are finite datatypes such as `list` and `tree` and are expressed as initial objects in a the category of algebras.

2.7.1. Contributions

We presented a categorically inspired datatype package that automates the representation of recursive datatypes as initial algebras. Our treatment includes regular datatypes as well as regular nested types. We automatically define the catamorphsim for each type as categorical method of defining recursive functions. Using catamorphisms, we subsequently define type functors, and also some associated laws, such as fusion and reflection laws. In addition to this automation, our main contribution is in the approach. Our theorems and laws are parameterised with the type and is therefore generic by definition. We achieve this using type quantification. Our formalism and associated proofs also provides a rigorous verification of the initial algebra semantics using type quantification in the HOL2P theorem prover. We are unaware of any other verification of this kind using HOL provers.

2.7.2. Related Work

Owre and Shankar describes their implementation of an abstract datatype mechanism in PVS [45].

2.7.3. Future work

- One immediate extension to this work is the relational treatment of datatypes in this framework.
- Our presentation deals with the treatment of regular types as well as regular nested types. One possible area of research is the treatment of non-regular nested types [9]
- We have only dealt with initial algebras but terminal objects can also be formalised in this framework with not much effort. One would however need to research the automation of such types and associated theorems and its use in mechanized reasoning.
- One can also research the categorical treatment of types with laws again with a focus on automation and its benefit to mechanized reasoning
- One could also investigate the extent to which types that are not initial algebras can fit into our categorical framework. Gibbons seems to suggest that there might be some sort of representation is possible, see [16]

REFERENCES

[1] Peter Aczel. Notes towards a formalisation of constructive galois theory. Technical report, 1994.

[2] Flemming Andersen and Kim Dam Petersen. Recursive boolean functions in HOL. In *1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 367–377. IEEE Computer Society, August 1991.

[3] M. A. Arbib and E. G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press, 1975.

[4] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[5] M. Barr and C. Wells. *Category theory for computing science.* Prentice Hall International, 1990.

[6] F. L. Bauer et al. *The Munich project CIP. Volume 1: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[7] R. S. Bird. An introduction to the theory of lists. F36:5–42, 1987. NATO ASI Series.

[8] R. S. Bird. Lectures on constructive functional programming. (PRG-69), 1988.

[9] Richard Bird and Lambert Meertens. Nested datatypes. *Lecture Notes in Computer Science*, 1422:52–??, 1998.

[10] Richard J. Bird. *Introduction to Functional Programming using Haskell.* Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.

[11] Richard S. Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall, 1997.

[12] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[13] Peter Dybjer and Verónica Gaspes. Implementing a category of sets in alf. Technical report, 1994.

[14] M. M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[15] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6(1):1–32, 1996.

[16] J. Gibbons. An initial-algebra approach to directed acyclic graphs. *Lecture Notes in Computer Science*, 947:282–??, 1995.

[17] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. Technical report, University of Glasgow, October 1993.

[18] J. A. Goguen. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery (JACM)*, 24(1):68–95, January 1977.

[19] Tatsuya Hagino. A category theoretic approach to data types. Master's thesis, University of Edinburgh, Department of Computer Science, 1987. CST-47-87 (also published as ECS-LFCS-87-38).

[20] J. Harrison. Inductive definitions: Automation and application. *Lecture Notes in Computer Science*, 971:200–??, 1995.

[21] John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[22] John Harrison. Floating point verification in HOL light: The exponential function. *Formal Methods in System Design*, 16(3):271–305, 2000.

[23] John Harrison. *The HOL Light manual*, 2000. Version 1.1.

[24] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66, Munich, Germany, 2009. Springer-Verlag.

[25] Peter V. Homeier. The HOL-omega logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2009.

[26] Gérard Huet and Amokrane Saïbi. Constructive category theory. In *In Proceedings of the joint clics-types workshop on categories and type theory, Goteborg.* MIT Press, 1998.

[27] Patrik Jansson. Functional polytypic programming use and implementation, May 21 1997.

[28] J. Lambek and P. J. Scott. Introduction to higher order categorical logic, 1986.

[29] Joachim Lambek. Deductive systems and categories – I. syntactic calculus and residuated categories. *Mathematical Systems Theory*, 2(4):287–318, 1968.

[30] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions, November 10 1995.

[31] Daniel J. Lehmann and Michael B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.

[32] A. C. Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol.* Gordon and Breach Science Publishers, New York, 1969.

[33] Donald W. Loveland. Erratum: "mechanical theorem-proving by model elimination". *J. ACM*, 16(1):646, 1969.

[34] G. Malcolm. *Algebraic data types and program transformation.* PhD thesis, Groningen University, 1990.

[35] E. G. Manes, editor. *Proceedings of the AAAS Symposium on Category Theory Applied to Computation and Control (San Francisco, California)*, number 25 in Lecture Notes in Computer Science. Springer-Verlag, 1975.

[36] Ernest Manes and Michael Arbib. *Algebraic Approaches to Program Semantics.* Springer-Verlag, 1986.

[37] Simon David Marlow. Deforestation for higher-order functional programs, 1995.

[38] Ursula Martin and Tobias Nipkow. Automating Squiggol. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 233–247. North-

Holland, 1990.

[39] Lambert Meertens. Calculate polytypically!, September 03 1996.

[40] Thomas F. Melham. Automating recursive type definitions in higher order logic. Technical Report 146, Computer Laboratory, University of Cambridge, September 1988.

[41] Thomas F. Melham. A package for inductive relation definitions in HOL. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *TPHOLs*, pages 350–357. IEEE Computer Society, 1991.

[42] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[43] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[44] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO, March 31 1997.

[45] Sam Owre and Natarajan Shankar. Abstract datatypes in PVS. Nasa/cr-97-206264, 1997.

[46] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. *Lecture Notes in Computer Science*, 814:148–??, 1994.

[47] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. Technical report CMU-CS-89-209, School of Computer Science, Carnegie-Mellon Univ., Pittsburgh, CA, 1989.

[48] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, Mass., 1991.

[49] S. Agerholm. A HOL Basis for Reasoning about Functional Programs. Brics rs-94-44, issn 0909-0878, Department of Computer Science, University of Aarhus, Denmark, December 1994. http://www.daimi.aau.dk/BRICS/RS/94/44/BRICS-RS-94-44/BRICS-RS-94-44.html.

[50] Supervised Am'ilcar Sernadas, Alexandra Carvalho, and Paulo Mateus. Category theory in coq, June 15 1998.

[51] N. Shankar. PVS: Combining specification, proof checking, and model checking. *Lecture Notes in Computer Science*, 1166:257–??, 1996.

[52] Mark E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler, September 15 1987.

[53] The Coq Development Team. The coq proof assistant reference manual, version 6.2. Technical report, INRIA, Roquencourt, France, 1998.

[54] Norbert Völker. HOL2P - A system of classical higher order logic with second order polymorphism. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 334–351. Springer, 2007.

[55] R. Walters. *Categories and Computer Science.* Cambridge University Press, Cambridge, 1986.