Applicative Functors in Isabelle/HOL: Notes

Joshua Schneider

November 23, 2015

1 Introduction

1.1 Motivation

Interactive theorem provers emphasize the human aspect of mathematical work. Rather than relying on fully automatic proof search, the user guides the process with their own understanding [1]. This requires a sufficiently expressive interface with features beyond plain logical calculus. In particular, there exists a demand for automation and abstraction of patterns [2]. It is thus common to extend proving environments with (sometimes domain-specific) tools.

In this report we present a particular variant of lifting we have implemented for the Isabelle/HOL proof assistant [3]. The term "lifting" is used in different contexts, often informally. It vaguely refers to the transfer of mathematical objects between domains, while preserving a certain relationship. A concrete example are lifts of paths in topology. There is also an existing Isabelle package [4], which lifts definitions to quotient types. Here we consider a slightly different meaning of lifting: The transfer of operations and their properties to generic structures. Since HOL is a typed logic, it is natural to represent these structures as parametric types. Let us look at a simple example.

Example 1. For each type α there is a corresponding type $\alpha\ set$ of sets with elements from α. Addition $(+)$ is a binary operator defined on natural numbers, amongst others. How could addition of sets of natural numbers be defined, such that some relation to $+$ remains? The canonical way to combine two sets into a set of pairs is the cartesian product. Therefore, we define

$$X \oplus Y = \{x + y \mid x \in X,\, y \in Y\}. \tag{1.1}$$

We interpret $\oplus$ as the lifting of $+$ to sets. Note that similar definitions are possible for other operators like multiplication, and also other element types such as real numbers. A property of addition is associativity,

$$\forall xyz.\, (x + y) + z = x + (y + z). \tag{1.2}$$

It can be translated to sets of natural numbers,

$$\forall XYZ.\, (X \oplus Y) \oplus Z = X \oplus (Y \oplus Z), \tag{1.3}$$

where it holds as well, as one can check with a slightly laborious proof. The two sides of the latter equation can be regarded as functions with three arguments. They are the lifted counterparts of the former equation. ▲

Hinze [5] came across similar patterns and proceeded to investigate the conditions under which lifted equations are preserved. He noticed that lifting can be defined in an intuitive fashion if the target structure is an applicative functor [6]. These come with two constants, usually denoted *pure* and $\diamond$ ("ap"), which lift a single object and the notion of function application, respectively. Applicative functors must also satisfy some properties, which we restate in Section 1.4.

Every monad is an applicative functor. Monads are a common mechanism for handling effects in functional programming [7]. Such being the case, they are also useful for modelling in the context of verification. The difference between monads and applicative functors is that the latter do not allow sub-computations to depend on previous results. From there we can borrow quite a few examples of applicative functors: Sum types with one variable type (known as Either in Haskell), the reader monad or environment functor, the exception/backtracking list monad [8], the state monad, and parser combinators. Hinze originally worked on streams (infinite lists) and infinite trees [9, 10]. We conclude that there are indeed relevant applicative functors, which supports the argument that this kind of lifting is a useful abstraction.

Example 1 (continued)**.** Back to sets, we obtain an applicative functor if *pure* is the singleton set constructor $x \mapsto \{x\}$; $F \diamond X$ takes a set of functions F and a set of arguments X with compatible type, applying each function to each argument:

$$F \diamond X = \{fx \mid f \in F, x \in X\}. \tag{1.4}$$

Now we can express lifted addition directly in terms of the base operation:[1]

$$X \oplus Y = \mathit{pure}\,(+) \diamond X \diamond Y. \tag{1.5}$$

(The operator $\diamond$ is left-associative.) This definition is equivalent to the previous one (1.1), but not specific to sets anymore. ▲

As we have seen earlier, lifting can be generalized to equations. One of Hinze's results is that a fundamental relationship exists between the associative properties (1.2) and (1.3)—the lifted form can be proven for all applicative functors, not just *set*. Moreover, this is possible for other equations as well, but not all equations can be lifted in all functors. Stronger conditions are required if the list of quantified variables is different for each side of the equation. (The left-to-right order is relevant, but not the nesting within the terms.) These conditions must basically ensure that the functor does not add "too many effects" which go beyond the simple embedding of a base type. Such effects may be evoked if a variable takes an impure value, i.e., a value which is not equal to $\mathit{pure}\,x$ for any x. Hinze showed that sufficient conditions can be expressed in terms of combinators as known from combinatory logic.

These findings justify direct proofs of lifted equations. It desirable to enable such reasoning in a proof assistant.

Example 2. We try to lift the fact that zero is a left absorbing element for multiplication of natural numbers, $\forall x :: \mathit{nat}.\ 0 \cdot x = 0$, to sets. Note that the variable x occurs only on the left. But the lifted equation does not hold: If x, now generalized to *nat set*, is instantiated with the empty set, then

$$\mathit{pure}\,(\cdot) \diamond \mathit{pure}\,0 \diamond \{\} = \{\} \neq \mathit{pure}\,0.$$

[1] As customary in HOL, we treat binary operators as curried functions.

Here the effect of {} is that it cancels out everything else if it occurs somewhere in an lifted expression. This makes it impossible to lift any equation with a variable occuring only on one side to *set*. We will see that other functors permit this lifting. ▲

1.2 Contributions and Overview

Our primary goal is to provide an Isabelle/HOL proof method which reduces lifted equations to their base form. The method can be instantiated for arbitrary applicative functors. Then the user is able to prove the lifted equations directly without having to invent a specific proof strategy, or even simulate the approach taken here. Apart from the theoretical appeal of the method, the resulting proof text is usually more concise, due to the higher level of abstraction. Together with some infrastructure, the proof method forms a basic package for reasoning with applicative functors.

Hinze's work is the basis for the package. He has identified the suitability of applicative functors and shown necessary conditions for lifting of equations. However, we needed to adapt the details to the HOL environment in order to construct actual proofs. This construction is directly programmed on top of Isabelle's ML interface. We have therefore derived the correctness and some other properties of our approach formally using a simplified calculus. Moreover, we have further extended the idea of combinators as building blocks for lifting, generalizing Hinze's model conditions. Consequently, the package supports several combinator sets which functors may exhibit, each capable of lifting different sets of equations. A particular motivation for our package are arithmetic operations on streams and infinite trees. We have proven some of their properties as a usage example.

The remainder of this report is organized as follows: The present section concludes with some background information on Isabelle/HOL, applicative functors, and a concrete definition of lifting. Section 2 describes the design considerations, discusses the choice of embedding, and provides a high-level overview of the proof method. In the next two sections, we present the details of the two strategies we have implemented: Section 3 shows an implementation of the normal form of lifted expressions; Section 4 is concerned with combinators and the application of bracket abstraction to applicative terms. To do. application example, related work, conclusion

1.3 Proving with Isabelle/HOL

Isabelle was originally designed as a framework for interactive theorem proving, without being restricted to a specific logical system [11]. However, one chooses a particular object-logic in order to construct a theory and prove theorems. In this paper, we focus on the Isabelle/HOL object-logic [3]. It implements the higher-order logic which was used in the HOL system, another proving environment [12]. Isabelle/HOL (or HOL from here on) is arguably the most popular object-logic of Isabelle, as it comes with an extensive library of readily formalized mathematics. It also supports modelling of functional programs by means of datatypes and recursive functions, making it suitable for verification tasks.

The basis of HOL is a slightly extended variant of simply-typed lambda calculus. Therefore, every object (and every term representing such an object) has a certain type attached to it. We use lower-case greek letters α, β, γ as meta-variables for types. The language of types consists of base types, type variables, and compound types. Base types are represented by their name and include fundamental types like the booleans *bool* and the natural numbers *nat*. Type variables stand for an arbitrary types. In Isabelle syntax, they are distinguished by a prefixed ‘’’, e.g. $'a$, $'b$, $'c$. The polymorphism in HOL is quite restricted, though, because higher-ranked types cannot be expressed: There is no explicit quantifier on the type level. This rules out functions which take a polymorphic function as an argument and apply it to values of different types. Compound types are built up of a type constructor and a list of types. The type constructor determines the number of argument types. For example, the unary type constructor *set* denotes sets with elements of a certain type. The argument is written on the left as in *nat set*, the type of sets of natural numbers. Multiple types are written in parentheses: (α, β)*fun*. *fun* is the special type constructor for (total) functions from α to β. More commonly, the infix operator $\Rightarrow$ is used. It is right-associative, i.e. $\alpha \Rightarrow \beta \Rightarrow \gamma$ is notation for $(\alpha, (\beta, \gamma)$*fun*$)$*fun*. Note that type constructors are different from types and must always be concrete. In particular, it is not possible to use a variable in place of a type constructor!

Terms follow the standard rules of lambda calculus. Atomic terms are constants and variables. Application of a function f to an argument x is written $f\,x$. Functions with multiple arguments are commonly curried in HOL; we can drop parentheses accordingly: $f\,x\,y$ is the same as $(f\,x)\,y$. Abstraction of a term t over the variable x is written $\lambda x.\,t$. Terms must be well-typed, of course. The types of variables and polymorphic constants can usually be omitted, since they are inferred automatically. Explicit type constraints are denoted by $t :: \alpha$ and may occur anywhere in a term. While all terms are represented internally roughly as shown above, Isabelle comes with extensible notation support.

Example 3. We already introduced the type *nat*. Number literals can be used directly. Common arithmetic operators are available, like[2] *plus* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat*. We can also use infix operators:

$$\lambda(x :: nat)\, y.\, 1 + x * y$$

is a function which multiplies two natural numbers and adds one to the result. Another important type family are sets. They can be specified as finite collections $\{\}$, $\{a, b, c\}$ etc., and by using set comprehension: Let P be a predicate $\alpha \Rightarrow$ *bool*. Then $\{x.\, Px\}$ is the set of those values $x :: \alpha$ such that Px is true, and $\{fx \mid x.\, Px\}$ is the image of that set under f.

Logical formulas are centered around truth values. Thus, the usual connectives like conjunction $\wedge$ and implication $\longrightarrow$ operate on type *bool*. Quantifiers work just as expected: The term

$$\forall(x :: nat)\, y.\, x + y = y + x$$

states that addition of natural numbers is commutative. Note that $=$ is just another operator of polymorphic type $'a \Rightarrow 'a \Rightarrow$ *bool*. Internally, quantifiers

[2]These functions actually have a more generic type (they are overloaded). We will look at this later on.

are represented as constants applied to lambda abstractions, which handle the variable binding. ▲

In order to support different object-logics, Isabelle contains an immediate layer, the meta-logic Pure (see e.g. [13, Chapter 2]). It is an "intuitionistic fragment of higher-order logic" [14, p. 27]. Pure has two main uses within the Isabelle framework: representation and manipulation of deduction rules, and goal states. While our user interface is intended to be used with HOL, the underlying theorems are always represented as propositions of Pure. Therefore it plays an ancillary role in our tool. The following operators are separate from those of HOL: meta-quantification $\bigwedge$ instead of the universal quantifier $\forall$, and meta-equality $\equiv$ instead of $=$. Each of these are logically equivalent, though. In particular, the generic rewriting and simplification tools work with $\equiv$. As an example of Pure, consider the symmetry axiom for $\equiv$ in traditional rule format,

$$\frac{\Gamma \vdash x \equiv y}{\Gamma \vdash y \equiv x}.$$

Deduction in Isabelle handles the context Γ automatically, and therefore does not have to be stated in the corresponding proposition. The dependency of the conclusion on the hypothesis translates to meta-implication; the outermost meta-quantified variables (and all type variables) are turned into schematic variables, which are free variables distinguished by the prefix *?*. Thus, the symmetry axiom appears as the theorem

$$\texttt{symmetric}: \quad ?x \equiv ?y \Longrightarrow ?y \equiv ?x.$$

Schematic (type) variables are eligible for instantiation by some core functions.

Isabelle is an LCF-style prover, i.e., proofs are composed by invoking trusted, primitive inferences programmed in the implementation language SML, at least on the lowest level. For instance, the symmetry axiom is available in combination with modus ponens as the function `Thm.symmetric`, which takes a theorem $x \equiv y$ and returns a new theorem $y \equiv x$. The function is part of the large SML interface of Isabelle, which not only constitutes its implementation, but can also be used for new tools.

In contemporary use of Isabelle, user input to the system is expressed in the Isar language [15, 16, 14]. It aims to encode proofs in a way that is formal, i.e. has precise semantics, but still resembles informal patterns of reasoning. The basic organization unit in Isar is a theory. The body of a theory consists of a sequence of commands, which consecutively augment the logical context by declarations of various kinds. Other theories may be imported in the beginning, leading to a acyclic graph of theory dependencies. The set of data stored within a theory can be extended through the programming interface. This is relevant to us, because it allows us to register applicative functors in the theory context.

Commands constitute the so-called outer syntax of Isar. Terms and types occuring within them are parsed separately, according to the inner syntax. In certain cases, a command may put the theory state into proof mode. After the proof is finished, the associated goal becomes a fact, which can be further handled by some associated code. The important **lemma** family of commands works this way, for example. New commands are frequently introduced for specifications. For example, algebraic datatypes can be defined with the **datatype**

command. Finally, there are other extensible syntactical categories which are commonly used in Isar: Proof methods denote (possibly parameterized) operations on the goal state within a regular Isar proof. These operations are provided by arbitrary code and thus can be arbitrarily complex. Attributes invoke further processing steps on already proven facts, either transforming them or causing additional declarations to the context.

1.4 Applicative Functors and Lifting

Applicative functors were introduced by McBride and Paterson [6] in order to abstract a recurring theme they observed in the programming language Haskell. In fact, their findings already included some examples of lifting. They defined an applicative functor as a unary type constructor f with associated constants

$$\begin{aligned} \mathit{pure}_f &:: \alpha \Rightarrow \alpha f, \\ (\diamond_f) &:: (\alpha \Rightarrow \beta) f \Rightarrow \alpha f \Rightarrow \beta f. \end{aligned}$$

We omit the subscripts if the functor is clear from the context. Moreover, the following laws must be satisfied:

$$\begin{aligned} \mathit{pure}\, \mathit{id} \diamond u &= u && \text{(identity)} \\ \mathit{pure}\, (\cdot) \diamond u \diamond v \diamond w &= u \diamond (v \diamond w) && \text{(composition)} \\ \mathit{pure}\, f \diamond \mathit{pure}\, x &= \mathit{pure}\, (f x) && \text{(homomorphism)} \\ u \diamond \mathit{pure}\, x &= \mathit{pure}\, (\lambda f.\ f x) \diamond u && \text{(interchange)} \end{aligned}$$

We have already seen how the two constants can be used to build terms. McBride and Paterson coined the term "idiom" to refer to a particular interpretation of such terms. In line with Hinze, we will use "applicative functor" and "idiom" interchangeably.

The identity type constructor defined by $\alpha\, \mathit{id} = \alpha$ is a trivial applicative functor for $\mathit{pure}\, x = x$, $f \diamond x = f x$. We can take any abstraction-free term t and replace each constant c by $\mathit{pure}\, c$, and each instance of function application $f x$ by $f \diamond x$. The rewritten term is equivalent to t when interpreted in the identity idiom. By choosing a different idiom, we obtain a different interpretation of the same term structure. In fact, this is exactly how we define the lifting of t. However, the terms we are interested in can also contain variables: Equations such as (1.2) are universally quantified. For the purpose of lifting, we ignore quantifiers and treat their variables as free. Like in Example 1, the variables of lifted equation should range over the lifted type. Note that these can take impure values. A term consisting only of *pure* and $\diamond$ applications and free variables is then called an *idiomatic expression*.

Every idiom is a functor, of course, and thus can be mapped over. One obtains an equivalent formalization of idioms:

$$\begin{aligned} \mathit{map}_f &:: (\alpha \Rightarrow \beta) \Rightarrow \alpha f \Rightarrow \beta f, & \mathit{map}_f\ g\, x &= \mathit{pure}\, g \diamond x, \\ \mathit{unit}_f &:: () f, & \mathit{unit}_f &= \mathit{pure}\, (), \\ (\star_f) &:: \alpha f \Rightarrow \beta f \Rightarrow (\alpha, \beta) f, & x \star_f y &= \mathit{pure}\, (\lambda x y.\ (x, y)) \diamond x \diamond y. \end{aligned}$$

Some of Hinze's proofs make use of these definitions. Dealing with product types can be a bit cumbersome in HOL, though. The curried interface therefore seems to be a better choice, and we will focus solely on that.

2 Requirements and Basic Design

2.1 Requirements

Our primary goal is to implement an Isabelle/HOL proof method which reduces lifted equations to their base form. This proof method should be generic and work with arbitrary idioms. The following is the minimal set of user actions we shall support:

a) Declare applicative functors to the theory context. Given a type constructor f, the functions $pure_f$ and $\diamond_f$, and proofs for the relevant functor laws, the functor is registered with the package such that it can be used in subsequent invocations of the proof method.

b) Prove lifted equations $a'[\vec{x}] = b'[\vec{x}]$, where a' and b' are idiomatic expressions with free variables $\vec{x}$, using the base equation $\forall \vec{y}.\, a[\vec{y}] = b[\vec{y}]$. More precisely, if there is a subgoal stating the former, applying the proof methods transforms the goal state to the latter. The functor should be either detected automatically, or specified by the user.

The first requirement ensures that our package is reusable, while the second is the core functionality. However, usability is also a concern: In the realm of interactive theorem proving, it is not sufficient to just verify formal objects—we are not extending the logic, after all, but providing a shortcut for a certain principle. We must balance the clarity of the resulting proof document and the amount of work that the user has to put into developing a proof. The following features are possible extensions which may help in this regard:

c) Declare lifted constants and other terms to the theory context. Generally speaking, if a term t with free variables v_i can be expressed as $pure\, t' \diamond v_1 \diamond \cdots \diamond v_n$ for some t', then the corresponding equation can be registered. The lifting proof methods rewrites with these equations at the beginning, such that the base equation will refer to t'. This way, the user does not have to transform everything into idiomatic format first.

d) More flexibility regarding the logical structure of the input proposition. This includes bound variables (quantified by $\forall$ or $\bigwedge$), complex subgoals with premises, and cases where the conceptual variables of the lifted equation have been subsituted by some terms.

e) Related proof methods and attributes, for example for forward lifting of proven base equations.

f) Inspection and tracing output. This is particular useful if something does not work as expected.

g) Extend the notion of lifting beyond equations. It is possible to define lifting for other logical operators. For example, the cancellation law $a + b = a + c \longrightarrow b = c$ consists of two equations, joined by implication. We can interpret it in different domains, e.g., for integers and for streams of integers with element-wise addition. In this example, the law is true for both interpretations. We are not able to handle such propositions with just a method for equations, though.

The current version at the time writing supports c) and d), as well as e) and f) to some extent. To do. g) not discussed in detail, but see final section. In the remainder of this section, we will explain the design decisions we have made in order to fulfill the requirements. Furthermore, the registration infrastructure and the basic proof approach are explained.

2.2 Choice of Embedding

The basic way of proof composition in Isabelle is resolution with a proven theorem, using it as a rule of inference. Here we argue that it is not possible to prove lifting as a single theorem, and then we discuss potential remedies. We are interested in applicative functors on the type level, where application is based on the standard function space. Therefore, each idiom comes with a type family which is indexed by a distinguished type variable, and the related functions and laws are polymorphic in this index. This is the natural form of idioms in the HOL libraries; all examples in 1.4 are parametric datatypes. The proof method should work directly with equations of these idioms. Regardless of the mechanism of proof construction, it needs a type constructor as a parameter. This concept is foreign to the type system of Pure and HOL—we already referred to the fact that type constructors are fixed. Another issue is the lack of polymorphism in the inner logic: We cannot have, say, a schematic variable *?pure* and use it with different types within the same proposition or proof.

One solution is to define a custom logic, including a term language, axioms and meta theorems, and formalize it using the available specification tools. In our case, the language is that of idiomatic expressions,[3] the axioms describe an equality judgment which is compatible with the applicative functor laws, and lifting of equations is a theorem. This is a *deep embedding* of the logic [17]. With its tools for algebraic datatypes and recursive functions, reasoning about such an embedding is quite manageable in HOL. However, we want to prove propositions involving objects of HOL itself, not just their encodings in the embedded language. Some machinery, known as reflection, needs to perform the encoding and transfer back results. It must be implemented necessarily outside of the logic, but can be generic. Chaieb and Nipkow have implemented a proof procedure using a deep embedding and reflection in Isabelle [18]. They point out that their approach also functions as a verification of the proof procedure, is portable and has smaller proofs than those obtained by automating inference rules. Their reflection system does not support polymorphism to the extent we need, though.

The package for nonfree datatypes [19] is deeply embedded as well. Its constructions must work with arbitrary types. In the underlying framework, Schropp [20] proposed the use of a "pseudo-universe", a sum type combining all these types. The meta-theory of the package carries a type parameter which is instantiated with a suitable pseudo-universe for every construction. It may be possible to use the same approach for idiomatic expressions, since the number of types occuring in an idiomatic expression is finite. This number can be linear in the size of the expression, though, which bloats the intermediate terms during reflection. A bigger problem is the generic axiomatization of idioms. For instance, the identity law would refer to a function of type $\alpha \Rightarrow \alpha$ for each type

[3] Base equations can be interpreted as expressions of the identity idiom.

α in the pseudo-universe. Thus, the universe needs to be closed under function types. It is not clear to the author how this could be modelled.

A different approach, which is the one we will take, is a *shallow embedding*. The formulas (here, idiomatic terms) are expressed directly in the language of HOL. Due to aforementioned restrictions, meta-theorectical results must be provided in specialized form for each case. We use the powerful ML interface of Isabelle to program the proof construction, composing inferences according to the structure of the input equation. The handling of polymorphism is simplified, as we have full control over term and theorem instantiations. The system still verifies the soundness of the synthesized proofs. On the other hand, one has to assert externally that the construction algorithm itself is correct, i.e., complete. The main part of this report therefore justifies these algorithms.

2.3 Proof Strategy

The ML code of the package can be considered in two parts. One is concerned with registration of idioms and access to the recorded information, the other provides the actual proof methods. We start with a summary of the former component. It does not only store the idioms declared with the **applicative** command (see Section 5.1), but also deals with the high degree of polymorphism. First, we need to represent the concept of an idiom. Instead of a plain type constructor we use type schemes $f['a]$, which are simply normal HOL types with (in this case) one distinguished type variable $'a$. This variable has to be tracked externally in ML because of the lack of type quantifiers in the type system. The lifted type of α is then obtained by substitution of $'a$ by α in $f['a]$. The functions $pure_f$ and $\diamond_f$ thus have types $'a \Rightarrow f['a]$ and $f['a \Rightarrow 'b] \Rightarrow f['a] \Rightarrow f['b]$, respectively. We allow additional type variables in functor signatures, which therefore represent families of functors. These are always instantiated with the same types throughout a proof. This is useful for idioms based on sum types $\alpha+\beta$, for instance. As a further refinement, all type variables may be constrained by a sort. Sorts in Pure are type classes [13]. The sort of the distinguished variable in f must be compatible with the function type constructor.

Above parts form the signature of the idiom f. Using it, we define functions which compose and destruct types and terms related to f. One problem is identifying terms of the form $pure\ \_$ and $\_ \diamond \_$ in the first place. We do not want to restrict the term structure of the functor "constants", since some useful predefined concepts in the HOL libraries are abbreviations of compound terms. Isabelle's higher-order matching appears to be a proper solution in practice, using $pure\, v$ (or $v_1 \diamond v_2$) with new variables v (v_1, v_2) as the pattern.

The remaining code uses these functions to provide several layers of conversions and tactics. A wrapper tactic prepares the subgoal to solve. This includes dealing with universal quantifiers $\forall$ and local premises. We then rewrite the subgoal using the declared rules for lifted constants. Only those related to f are used, the reason being that overeager, unwanted unfolding may be difficult to reverse. The result of this preparation is a subgoal which is a simple equation of two idiomatic expressions. We provide two variants of the basic lifting step. They have in common that both expressions are transformed into *canonical form*:

$$pure\, g \diamond s_1 \diamond \cdots \diamond s_m = pure\, h \diamond t_1 \diamond \cdots \diamond t_n,$$

We then apply appropriate congruence rules until the subgoal is reduced to $g = h$. If either $m \neq n$ or $s_i \neq t_i$ for some i (as terms modulo $\alpha\beta\eta$-conversion), the proof method fails. Since g and h are at least n-ary functions, we can further apply extensionality, reaching the subgoal

$$\bigwedge x_1 \ldots x_n.\, g x_1 \cdots x_n = h x_1 \cdots x_n.$$

This is the transformed proof state presented to the user.

Hinze's Normal Form Lemma [5, p. 7] asserts the existence of a certain normal form for idiomatic expressions where each variable occurs only once. As it turns out, we can compute it for arbitrary expressions. This normal form is a canonical form. Therefore, the first variant simply replaces both sides of the equation with the normal form. The details of the normalization algorithm are described in Section 3. There we will also show that the transformed equation is indeed the base form of the original equation. The other implementation is a superset of the normal form approach. Instead of using the unique normal form, we construct the canonical forms explicitly such that the condition $\vec{s} = \vec{t}$ is satisfied. The idiom under consideration limits the set of equivalent canonical forms, though. The construction is related to bracket abstraction of lambda terms—we add combinators to the term in order to separate the lifted part from the variables. This is further explained in Section 4.

3 Normal Form Conversion

McBride and Paterson [6] noted that idiomatic expressions can be transformed into an application of a pure function to a sequence of impure arguments. They called this the *canonical form* of the expression. Hinze [5, Section 3.3] gave an explicit construction based on the monoidal variant of applicative functors. Transforming the terms in this way is useful for our purpose, because the arguments of the remaining *pure* terms reflect the equation that was lifted. The soundness of the algorithm depends only on the applicative laws, making it the most general approach regarding functors (but not regarding lifted equalities). We will later show that all transformations based on the applicative laws yield a unique canonical form, modulo $\alpha\beta\eta$-conversion. Therefore, we follow Hinze and denote by *normal form* this particular canonical form. The distinction is necessary, as we will consider other canonical forms in Section 4, which are justified by additional laws.

In the following, we define lifting and normalization formally, based on a syntactic representation of idiomatic terms. While this presentation is more abstract than what is actually happening in Isabelle, it makes it easier to demonstrate correctness and some other properties. Unlike Hinze, we use the *pure*/$\diamond$ formalism. Then we describe the implementation of the normalization procedure in Isabelle/ML.

3.1 The Idiomatic Calculus

In Section 1.4, we introduced idiomatic expressions built from *pure* and $\diamond$ constants of an applicative functor. This structure maps straightforward to a recursive datatype, given that there is a representation for arguments of *pure*. These must have some structure as well such that the applicative laws can be

expressed. It should also be possible to have "opaque" idiomatic subterms, which cannot (or should not) be written as a combination of *pure* and $\diamond$. This is primarily useful for variables ranging over lifted types, but as demonstrated in Example **??**, more complex terms may occur too. Therefore it makes sense to refer to general lambda terms in both cases; then we can define semantics consistently. Types are ignored here for simplicity. However, all results are compatible with the restrictions of simply typed lambda calculus.

Definition 1 (Untyped lambda terms)**.** Let $\mathcal{V}$ be an infinite set of variable symbols. We assume that f, g, x, y are disjoint variables in the following formulas. The set of untyped lambda terms is defined as

$$\mathcal{T} ::= \mathcal{V} \mid (\mathcal{T}\ \mathcal{T}) \mid \boldsymbol{\lambda}\mathcal{V}.\ \mathcal{T} \tag{3.1}$$

An equivalence relation on $\mathcal{T}$ is a $\mathcal{T}$-congruence iff it is closed under application and abstraction. Let $=_{\alpha\beta\eta}$ be the smallest $\mathcal{T}$-congruence containing α-, β-, and η-conversion. ▲

Definition 2 (Idiomatic terms)**.** The set of idiomatic terms is defined as

$$\mathcal{I} ::= \mathsf{term}\,\mathcal{T} \mid \mathsf{pure}\,\mathcal{T} \mid \mathcal{I}\ \text{`}\mathsf{ap}\text{`}\ \mathcal{I}. \tag{3.2}$$

By convention, the binary operator ‘ap‘ associates to the left. An $\mathcal{I}$-congruence is an equivalence relation closed under ‘ap‘. We overload notation and reuse $=_{\alpha\beta\eta}$ for idiomatic terms, where it stands for structural equality modulo substitution of $=_{\alpha\beta\eta}$-equivalent lambda terms. The $\mathcal{I}$-congruence $\simeq$ is induced by the rules

$$x \simeq \mathsf{pure}\,(\boldsymbol{\lambda}x.\ x)\ \text{`}\mathsf{ap}\text{`}\ x \tag{3.3}$$

$$g\ \text{`}\mathsf{ap}\text{`}\ (f\ \text{`}\mathsf{ap}\text{`}\ x) \simeq \mathsf{pure}\,\mathbf{B}\ \text{`}\mathsf{ap}\text{`}\ g\ \text{`}\mathsf{ap}\text{`}\ f\ \text{`}\mathsf{ap}\text{`}\ x \tag{3.4}$$

$$\mathsf{pure}\,f\ \text{`}\mathsf{ap}\text{`}\ \mathsf{pure}\,x \simeq \mathsf{pure}\,(f\ x) \tag{3.5}$$

$$f\ \text{`}\mathsf{ap}\text{`}\ \mathsf{pure}\,x \simeq \mathsf{pure}\,((\boldsymbol{\lambda}x.\ \boldsymbol{\lambda}f.\ f\ x)\ x)\ \text{`}\mathsf{ap}\text{`}\ f \tag{3.6}$$

$$s =_{\alpha\beta\eta} t \implies s \simeq t \tag{3.7}$$

where $\mathbf{B}$ abbreviates $\mathbf{B} \equiv \boldsymbol{\lambda}g.\ \boldsymbol{\lambda}f.\ \boldsymbol{\lambda}x.\ g\ (f\ x)$. ▲

term represents arbitrary values in the lifted domain, whereas pure lifts a value. The introduction rules for the relation $\simeq$ are obviously the syntactical counterparts of the applicative laws. Together with symmetry, substitution, etc., they give rise to a simple calculus of equivalence judgements. The intuitive meaning of $s \simeq t$ is that the terms can be used interchangeably. For example, there is a derivation for

$$\mathsf{pure}\,g\ \text{`}\mathsf{ap}\text{`}\ (f\ \text{`}\mathsf{ap}\text{`}\ x) \simeq \mathsf{pure}\,(\mathbf{B}\ g)\ \text{`}\mathsf{ap}\text{`}\ f\ \text{`}\mathsf{ap}\text{`}\ x \tag{3.8}$$

from (3.4), where g is instantied with $\mathsf{pure}\,g$, and a substitution along (3.5) on the right-hand side.

Idiomatic terms are visualized naturally as trees. This will be helpful in explaining term transformations. Figure 1 shows the conventions: Inner nodes correspond to ‘ap‘, leaves are either pure terms (boxes) or opaque terms (circles). Whole subterms may be abbreviated by a triangle. A term has canonical form if it consists of a single pure node to which a number of opaque terms (or none) are applied in sequence. Figure 2 gives a general example. A formal construction follows:

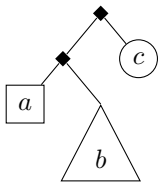

Figure 1: (pure a ‘ap‘ b) ‘ap‘ term c as a tree.

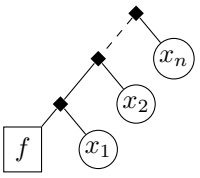

Figure 2: A term in canonical form.

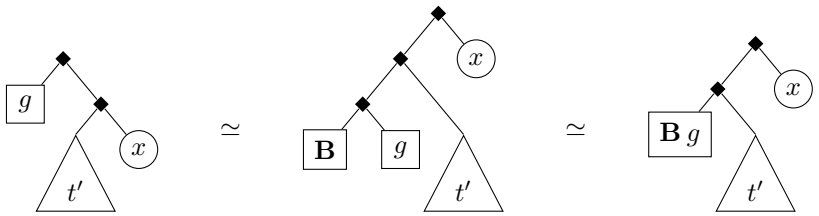

Figure 3: The “pure-rotate” step.

Definition 3 (Canonical form)**.** The set $\mathcal{C} \subset \mathcal{I}$ of idiomatic terms in canonical form is defined inductively as

$$\mathsf{pure}\, x \in \mathcal{C}, \tag{3.9}$$

$$t \in \mathcal{C} \implies t \text{ ‘ap‘ } \mathsf{term}\, s \in \mathcal{C}. \tag{3.10}$$

▲

It is not entirely obvious how a canonical form can be derived from equations (3.3)–(3.6). Rewriting blindly with these is prone to infinite recursion. Therefore we need a more controlled algorithm. As we have said before, we use *normal form* to refer to the particular canonical form derived from those equations. Consider an idiomatic term t. If t is a single pure term, then it is already in normal form. The case $t = \mathsf{term}\, x$ is also easy: Due to (3.3), we have $t \simeq \mathsf{pure}\,(\boldsymbol{\lambda} x.\, x)$ ‘ap‘ t, which is in normal form. But in the case of $t = u$ ‘ap‘ v, various steps could be performed, depending on the subterms. We simplify the situation by normalizing each subterm recursively, so we get an equivalent term u' ‘ap‘ v' where $u', v' \in \mathcal{C}$.

Now let us assume that u' is just $\mathsf{pure}\, g$. If v' is also a pure term, they can be combined along (3.5). Otherwise, the term looks like the one on the left of Figure 3. As is shown there, the term tree can be rotated such that one opaque term moves to the outer-most level. This is the same equivalence as stated in (3.8). Because the remaining part again has the shape “pure term applied to normal form”, we proceed recursively. In pattern-matching style, the transformation ‘pure-nf’ reads

$$\text{pure-nf}(\mathsf{pure}\, g \text{ ‘ap‘ } (f \text{ ‘ap‘ } x)) = \text{pure-nf}\,(\mathsf{pure}\,(\mathbf{B} g) \text{ ‘ap‘ } f) \text{ ‘ap‘ } x \tag{3.11}$$

$$\text{pure-nf}(\mathsf{pure}\, f \text{ ‘ap‘ } \mathsf{pure}\, x) = \mathsf{pure}\,(f x) \tag{3.12}$$

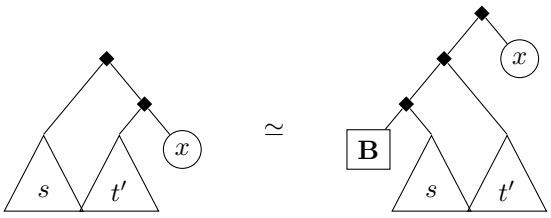

Figure 4: The "rotate" step.

Lemma 1. *For all $g \in \mathcal{T}$ and $t \in \mathcal{C}$,* pure-nf($\mathsf{pure}\, g$ ‘ap‘ t) *is well-defined, and*[4] pure-nf($\mathsf{pure}\, g$ ‘ap‘ t) $\in \mathcal{C} \simeq \mathsf{pure}\, g$ ‘ap‘ t.

Proof. We prove all claims simultaneously by induction on $t \in \mathcal{C}$, where g is arbitrary.

Case 1. Assume $t = \mathsf{pure}\, x$ for some $x \in \mathcal{T}$. Only the second equation applies, so we have

$$\text{pure-nf}(\mathsf{pure}\, g \text{ ‘}\mathsf{ap}\text{‘ } t) = \mathsf{pure}\, (g\, x).$$

All pure terms are in $\mathcal{C}$, and equivalence follows from (3.5).

Case 2. Assume $t = t' \text{‘}\mathsf{ap}\text{‘}\, \mathsf{term}\, x$ for some $t' \in \mathcal{C}$, $x \in \mathcal{T}$, and that the hypothesis holds for t' and all g. Only the first equation applies, so

$$\text{pure-nf}(\mathsf{pure}\, g \text{ ‘}\mathsf{ap}\text{‘ } t) = \text{pure-nf}(\mathsf{pure}\, (\mathbf{B} g) \text{ ‘}\mathsf{ap}\text{‘ } t') \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{term}\, x.$$

Instantiating the induction hypothesis, we find that

$$\text{pure-nf}(\mathsf{pure}\, (\mathbf{B}\, g) \text{ ‘}\mathsf{ap}\text{‘ } t') \in \mathcal{C} \simeq \mathsf{pure}\, (\mathbf{B}\, g) \text{ ‘}\mathsf{ap}\text{‘ } t'$$

is well-defined. $\mathcal{C}$ is closed under application to opaque terms (3.10), hence pure-nf($\mathsf{pure}\, g$ ‘ap‘ t) $\in \mathcal{C}$. Finally, we have

$$\begin{aligned}\text{pure-nf}(\mathsf{pure}\, g \text{ ‘}\mathsf{ap}\text{‘ } t) &\simeq \mathsf{pure}\, (\mathbf{B}\, g) \text{ ‘}\mathsf{ap}\text{‘ } t' \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{term}\, x \\ &\overset{(3.8)}{\simeq} \mathsf{pure}\, g \text{ ‘}\mathsf{ap}\text{‘ } (\mathsf{pure}\, t' \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{term}\, x) = \mathsf{pure}\, g \text{ ‘}\mathsf{ap}\text{‘ } t. \qquad \square\end{aligned}$$

Going back to $u' \text{ ‘}\mathsf{ap}\text{‘ } v'$, we assumed that u' is a pure term. The case where v' is pure instead can be translated to the former by

$$\text{nf-pure}(f \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{pure}\, x) = \text{pure-nf}\, (\mathsf{pure}\, ((\boldsymbol{\lambda} x.\, \boldsymbol{\lambda} f.\, f x)\, x) \text{ ‘}\mathsf{ap}\text{‘ } f) \tag{3.13}$$

Lemma 2. *For all $t \in \mathcal{C}$ and $x \in \mathcal{T}$,* nf-pure(t ‘ap‘ $\mathsf{pure}\, x$) *is well-defined, and* nf-pure(t ‘ap‘ $\mathsf{pure}\, x$) $\in \mathcal{C} \simeq t$ ‘ap‘ $\mathsf{pure}\, x$.

Proof. Follows from Lemma 1 and (3.6). $\square$

Finally, we look at general u', v'. A term rotation is useful again, see Figure 4. Before recursion, we must normalize the subterm $\mathsf{pure}\, \mathbf{B}$ ‘ap‘ s. But

[4] $a \in S \simeq b$ abbreviates "$a \in S$ and $a \simeq b$".

we already know how to do this: by 'pure-nf'. The base case is reached when v' is a single pure term, which is the domain of 'nf-pure'. The corresponding transformation is therefore

$$\text{nf-nf}(g \text{ `}\mathsf{ap}\text{`} (f \text{ `}\mathsf{ap}\text{`} x)) = \text{nf-nf}\,(\text{pure-nf}\,(\mathsf{pure}\,\mathbf{B} \text{ `}\mathsf{ap}\text{`} g) \text{ `}\mathsf{ap}\text{`} f) \text{ `}\mathsf{ap}\text{`} x \tag{3.14}$$

$$\text{nf-nf}(t) = \text{nf-pure}(t) \quad \text{(otherwise)} \tag{3.15}$$

Lemma 3. *For all* $s, t \in \mathcal{C}$, $\text{nf-nf}(s \text{ `}\mathsf{ap}\text{`} t)$ *is well-defined, and* $\text{nf-nf}(s \text{ `}\mathsf{ap}\text{`} t) \in \mathcal{C} \simeq s \text{ `}\mathsf{ap}\text{`} t$.

Proof. The proof is similar to the one of Lemma 1, by induction on $t \in \mathcal{C}$ and arbitrary $s \in \mathcal{C}$.

Case 1. Assume $t = \mathsf{pure}\,x$ for some $x \in \mathcal{T}$. The second equation applies, so we have

$$\text{nf-nf}(s \text{ `}\mathsf{ap}\text{`} t) = \text{nf-pure}(s \text{ `}\mathsf{ap}\text{`} \mathsf{pure}\,x).$$

Since $s \in \mathcal{C}$, the claim follows directly from Lemma 2.

Case 2. Assume $t = t' \text{`}\mathsf{ap}\text{`}\mathsf{term}\,x$ for some $t' \in \mathcal{C}$, $x \in \mathcal{T}$, and that the hypothesis holds for t' and all $s \in \mathcal{C}$. Only the first equation applies,

$$\text{nf-nf}(s \text{ `}\mathsf{ap}\text{`} t) = \text{nf-nf}(\text{pure-nf}(\mathsf{pure}\,\mathbf{B} \text{ `}\mathsf{ap}\text{`} s) \text{ `}\mathsf{ap}\text{`} t') \text{ `}\mathsf{ap}\text{`} \mathsf{term}\,x.$$

We have $\text{pure-nf}(\mathsf{pure}\,\mathbf{B} \text{ `}\mathsf{ap}\text{`} s) \in \mathcal{C} \simeq \mathsf{pure}\,\mathbf{B} \text{ `}\mathsf{ap}\text{`} s$ from Lemma 1. Thus we can instantiate the induction hypothesis, and the transformed term is indeed in normal form. Furthermore,

$$\begin{aligned}\text{nf-nf}(s \text{ `}\mathsf{ap}\text{`} t) &\overset{\text{(IH)}}{\simeq} \text{pure-nf}(\mathsf{pure}\,\mathbf{B} \text{ `}\mathsf{ap}\text{`} s) \text{ `}\mathsf{ap}\text{`} t' \text{ `}\mathsf{ap}\text{`} \mathsf{term}\,x \\ &\simeq \mathsf{pure}\,\mathbf{B} \text{ `}\mathsf{ap}\text{`} s \text{ `}\mathsf{ap}\text{`} t' \text{ `}\mathsf{ap}\text{`} \mathsf{term}\,x \\ &\overset{(3.4)}{\simeq} s \text{ `}\mathsf{ap}\text{`} (t' \text{ `}\mathsf{ap}\text{`} \mathsf{term}\,x) = s \text{ `}\mathsf{ap}\text{`} t.\end{aligned}$$

□

Algorithm 1 summarizes all pieces of the normal form transformation. 'normalize' is the entry point and performs the main recursion mentioned in the beginning. We haven't proved the desired property for 'normalize' yet, but this is just a straightforward induction.

Lemma 4. *For all* $t \in \mathcal{I}$, $\text{normalize}\,t$ *is well-defined, and* $\text{normalize}\,t \in \mathcal{C} \simeq t$.

Proof. By induction on t, Lemma 3, and equation (3.3). □

At this point, we know that it is always possible to obtain a certain canonical form. This is not sufficient to setup the complete proving process, though. We need to learn a bit more about the structure of idiomatic terms and how it relates to lifting.

Definition 4 (Opaque subterms)**.** The sequence of opaque subterms of an idiomatic term is defined by the recursive function

$$\text{opaq}(\mathsf{pure}\,x) = [], \quad \text{opaq}(\mathsf{term}\,x) = [x], \quad \text{opaq}(s \text{ `}\mathsf{ap}\text{`} t) = \text{opaq}\,s @ \text{opaq}\,t.$$

@ denotes concatenation of lists. ▲

Algorithm 1 Normalization of idiomatic terms.

$$\begin{aligned}
\text{normalize}(\mathsf{pure}\, x) &= \mathsf{pure}\, x \\
\text{normalize}(\mathsf{term}\, x) &= \mathsf{pure}\, (\boldsymbol{\lambda} x.\ x)\ \text{`}\mathsf{ap}\text{`}\ \mathsf{term}\, x \\
\text{normalize}(x\ \text{`}\mathsf{ap}\text{`}\ y) &= \text{nf-nf}(\text{normalize}\, x\ \text{`}\mathsf{ap}\text{`}\ \text{normalize}\, y)
\end{aligned}$$

$$\begin{aligned}
\text{nf-nf}(g\ \text{`}\mathsf{ap}\text{`}\ (f\ \text{`}\mathsf{ap}\text{`}\ x)) &= \text{nf-nf}\ (\text{pure-nf}\ (\mathsf{pure}\, \mathbf{B}\ \text{`}\mathsf{ap}\text{`}\ g)\ \text{`}\mathsf{ap}\text{`}\ f)\ \text{`}\mathsf{ap}\text{`}\ x \\
\text{nf-nf}(t) &= \text{nf-pure}(t) \quad \text{(otherwise)}
\end{aligned}$$

$$\begin{aligned}
\text{pure-nf}(\mathsf{pure}\, g\ \text{`}\mathsf{ap}\text{`}\ (f\ \text{`}\mathsf{ap}\text{`}\ x)) &= \text{pure-nf}\ (\mathsf{pure}\ (\mathbf{B} g)\ \text{`}\mathsf{ap}\text{`}\ f)\ \text{`}\mathsf{ap}\text{`}\ x \\
\text{pure-nf}(\mathsf{pure}\, f\ \text{`}\mathsf{ap}\text{`}\ \mathsf{pure}\, x) &= \mathsf{pure}\, (f x)
\end{aligned}$$

$$\text{nf-pure}(f\ \text{`}\mathsf{ap}\text{`}\ \mathsf{pure}\, x) = \text{pure-nf}\ (\mathsf{pure}\ ((\boldsymbol{\lambda} x.\ \boldsymbol{\lambda} f.\ f x)\ x)\ \text{`}\mathsf{ap}\text{`}\ f)$$

Definition 5 (Unlifting)**.** Let t be some idiomatic term, and $n = |\operatorname{opaq} t|$. Let $v_{i \in \{1..n\}}$ be new variable symbols that do not occur anywhere in t. The "unlifted" lambda term corresponding to t is defined as

$$\downarrow t = \boldsymbol{\lambda} v_1.\ \cdots \boldsymbol{\lambda} v_n.\ \text{vary}_1\, t,$$

where

$$\text{vary}_i(\mathsf{pure}\, x) = x, \tag{3.16}$$
$$\text{vary}_i(\mathsf{term}\, x) = v_i, \tag{3.17}$$
$$\text{vary}_i(s\ \text{`}\mathsf{ap}\text{`}\ t) = (\text{vary}_i\, s)\ (\text{vary}_{i+|\operatorname{opaq} s|}\, t). \tag{3.18}$$

▲

Example 4. The definition of $\downarrow$ may need some explanation. Consider the idiomatic term

$$t \equiv \mathsf{pure}\, f\ \text{`}\mathsf{ap}\text{`}\ x\ \text{`}\mathsf{ap}\text{`}\ (\mathsf{pure}\, g\ \text{`}\mathsf{ap}\text{`}\ y\ \text{`}\mathsf{ap}\text{`}\ z).$$

Its unlifted term is

$$\downarrow t = \boldsymbol{\lambda} a.\ \boldsymbol{\lambda} b.\ \boldsymbol{\lambda} c.\ f\ a\ (g\ b\ c).$$

The applicative structure is the same, but all opaque terms have been substituted for new bound variables, which are assigned from left to right. We do not define lifting formally here, but it should be clear that this is some sort of inverse operation, given that variables appear only once. ▲

The interesting properties about these two concepts is that they are preserved by the equivalence relation $\simeq$, and can be directly read from the canonical form. Furthermore, we can leverage them to show the uniqueness of the normal form.

Lemma 5. *For equivalent terms $s \simeq t$, the sequences of opaque terms are equivalent w.r.t $=_{\alpha\beta\eta}$, and $\downarrow s =_{\alpha\beta\eta} \downarrow t$.*

Proof. (Sketch.) By induction on the relation $s \simeq t$, where we show $\text{vary}_i\, s =_{\alpha\beta\eta} \text{vary}_i\, t$ instead. The index i is arbitrary. The part regarding opaque terms is

shown easily for each case: We note that in (3.3)–(3.6), the opaque terms are identical for both sides. By the induction hypothesis, this is also true for the necessary closure rules for symmetry, transitivity, and substitution. It is obvious that (3.7) also preserves opaque terms. Regarding unlifted terms, we have

Case 1. (3.3)

$$\mathrm{vary}_i(\mathsf{pure}\,(\boldsymbol{\lambda}x.\,x)\;\text{`}\mathsf{ap}\text{`}\;x) = (\boldsymbol{\lambda}x.\,x)\;(\mathrm{vary}_i\,x) =_{\alpha\beta\eta} \mathrm{vary}_i\,x.$$

Case 2. (3.4) Let $j = i + |\operatorname{opaq} g|$ and $k = j + |\operatorname{opaq} f|$.

$$\begin{aligned}\mathrm{vary}_i(\mathsf{pure}\,\mathbf{B}\;\text{`}\mathsf{ap}\text{`}\;g\;\text{`}\mathsf{ap}\text{`}\;f\;\text{`}\mathsf{ap}\text{`}\;x) &= \mathbf{B}\;(\mathrm{vary}_i\,g)\;(\mathrm{vary}_j\,f)\;(\mathrm{vary}_k\,g)\\ &=_{\alpha\beta\eta} (\mathrm{vary}_i\,g)\;((\mathrm{vary}_j\,f)\;(\mathrm{vary}_k\,g))\\ &= \mathrm{vary}_i(g\;\text{`}\mathsf{ap}\text{`}\;(f\;\text{`}\mathsf{ap}\text{`}\;x)).\end{aligned}$$

Case 3. (3.5) and (3.6) are similar.

Case 4. (3.7) By induction.

Case 5. Symmetry, transitivity, and substitution: These follow from the induction hypothesis and the corresponding properties of $=_{\alpha\beta\eta}$.

□

Lemma 6. *Let* $\mathsf{pure}\,f$ *be the single pure term in* $t \in \mathcal{C}$*. Then* $f =_{\alpha\beta\eta} \downarrow t$*.*

Proof. By induction on $\mathcal{C}$. The base case is trivial. For the step case, we need to prove that

$$f' =_{\alpha\beta\eta} \Big\downarrow(g\;\text{`}\mathsf{ap}\text{`}\;\mathsf{term}\,x) = \boldsymbol{\lambda}v_1.\;\ldots\boldsymbol{\lambda}v_n.\,\boldsymbol{\lambda}v_{n+1}.\,(\mathrm{vary}_1\,g)\;v_{n+1},$$

where $\mathsf{pure}\,f'$ is the single pure term in g, $n = |\operatorname{opaq} g|$, and v_i are new variables. The right-hand side can be eta-reduced to

$$\boldsymbol{\lambda}v_1.\;\ldots\boldsymbol{\lambda}v_n.\,(\mathrm{vary}_1\,g) = \downarrow g\;.$$

From the induction hypothesis we get $f' =_{\alpha\beta\eta} \downarrow g$, which concludes the proof. □

This lemma is why we need eta-equivalence and not just use $=_{\alpha\beta}$. In fact, we can find a counterexample of equivalent canonical forms that do not agree in the pure function if $\rightarrow_\eta$ is not available:

$$\mathsf{pure}\,(\boldsymbol{\lambda}x.\,x)\;\text{`}\mathsf{ap}\text{`}\;(\mathsf{pure}\,f\;\text{`}\mathsf{ap}\text{`}\;x) \simeq \mathsf{pure}\,f\;\text{`}\mathsf{ap}\text{`}\;x \simeq \mathsf{pure}\,(\boldsymbol{\lambda}x.\,f\;x)\;\text{`}\mathsf{ap}\text{`}\;x.$$

The middle term is derived from (3.3), the latter from (3.4).

Corollary 1. *The normal form is structurally unique. Formally, if* $s, t \in \mathcal{C}$ *and* $s \simeq t$*, then* $s =_{\alpha\beta\eta} t$*.*

Now we have all tools ready to complete the picture. The following theorem shows that (limited) lifting is possible with just the applicative laws. Its proof hints towards the implementation in Isabelle, which is of course based on the normal form: Under the condition that the opaque terms are equivalent, normalizing two idiomatic terms reduces the problem to the "unlifted" terms.

Function	Pattern		Substitution	Name
normalize	$\mathsf{term}\, x$	$\simeq$	$\mathsf{pure}\,(\boldsymbol{\lambda} x.\ x)$ ‘ap‘ $\mathsf{term}\, x$	
	$?x$	$=$	$\mathit{pure}\,(\lambda x.\ x) \diamond ?x$	`I_intro`
nf-nf	g ‘ap‘ $(f$ ‘ap‘ $x)$	$\simeq$	$\mathsf{pure}\,\mathbf{B}$ ‘ap‘ g ‘ap‘ f ‘ap‘ x	
	$?g \diamond (?f \diamond ?x)$	$=$	$\mathit{pure}\,(\lambda g f x.\ g(f x)) \diamond ?g \diamond ?f \diamond ?x$	`B_intro`
pure-nf	$\mathsf{pure}\, g$ ‘ap‘ $(f$ ‘ap‘ $x)$	$\simeq$	$\mathsf{pure}\,(\mathbf{B}\ g)$ ‘ap‘ f ‘ap‘ x	
	$\mathit{pure}\ ?g \diamond (?f \diamond ?x)$	$=$	$\mathit{pure}\,(\lambda f x.\ ?g(f x)) \diamond ?f \diamond ?x$	`B_pure`
pure-nf	$\mathsf{pure}\, f$ ‘ap‘ $\mathsf{pure}\, x$	$\simeq$	$\mathsf{pure}\,(f\ x)$	
	$\mathit{pure}\ ?f \diamond \mathit{pure}\ ?x$	$=$	$\mathit{pure}\,(?f ?x)$	`merge`
nf-pure	f ‘ap‘ $\mathsf{pure}\, x$	$\simeq$	$\mathsf{pure}\,((\boldsymbol{\lambda} x.\ \boldsymbol{\lambda} f.\ f\ x)\ x)$ ‘ap‘ f	
	$?f \diamond \mathit{pure}\ ?x$	$=$	$\mathit{pure}\,(\lambda f.\ f\ ?x) \diamond ?f$	`swap`

Table 1: Fixed transformations of Algorithm 1, with corresponding rewrite rules. Identity cases are omitted.

Theorem 1. *Let* $s, t \in \mathcal{I}$ *with* $\operatorname{opaq} s =_{\alpha\beta\eta} \operatorname{opaq} t$. *If* $\downarrow s =_{\alpha\beta\eta} \downarrow t$ *(the base equation), then* $s \simeq t$.

Proof. From Lemma 4 we obtain normal forms $s' \simeq s$ and $t' \simeq t$. With the base equation and Lemma 5 we get $\big\downarrow s' =_{\alpha\beta\eta} \big\downarrow t'$. By Lemma 6 and the condition on the opaque subterms, it follows that $s' =_{\alpha\beta\eta} t'$ and further $s \simeq t$. □

3.2 Implementation

We introduced an algorithm for computing the normal form of an idiomatic term, and argued for its central role in lifting. In order to be useful in the context of theorem proving, just providing the normal form is not sufficient. We need to establish a formal proof of the equivalence. In Isabelle, this means constructing a theorem $t = t'$, where t' is the normal form of t. We observe that in Algorithm 1 each equation can be recast in the following way: The input term is first transformed in a fixed way that changes the outermost constitution. Then, the whole term or subterms thereof are substituted by other functions—or recursively—, possibly multiple times. For instance, in the first equation (3.14) for nf-nf, the term g ‘ap‘ $(f$ ‘ap‘ $x)$ (where g, f, x should be understood as placeholders for concrete terms) is rearranged to $\mathsf{pure}\,\mathbf{B}$ ‘ap‘ g ‘ap‘ f ‘ap‘ x. The subterm $\mathsf{pure}\,\mathbf{B}$ ‘ap‘ g is passed to pure-nf and replaced by the result, say g'. nf-nf acts on g' ‘ap‘ f, yielding f', such that the final term is f' ‘ap‘ x.

Table 1 shows all initial transformations. It is no coincidence that these are the well-known equivalences (3.3)–(3.6) and (3.8), which were used in the correctness proofs. Each transformation has been augmented with the corresponding HOL equation. The mapping from syntactic idiomatic terms to HOL terms is obvious; *pure* and ($\diamond$) are the concrete constants of the idiom under consideration. The variables standing for arbitrary opaque terms become new schematic variables, representing unknowns which may be instantiated. Rule `B_pure` is easily proven by rewriting with `B_intro` and `merge`. The other rules are the applicative laws and thus are available from the registration infrastructure, see also To do.. Applying a transformation to a term means instantiating the rule such that the left hand side is equal to that term. Isabelle provides a low-level reasoning framework for combining *conversions* [21], which we use

here. A conversion is a function which takes a term t and returns a theorem $t = u$ for some u.[5] The ML function `Conv.rewr_conv` turns a proven equation into such a conversion function. When the resulting conversion is applied, it attempts to instantiate the schematic variables in the equation. Note that conversions can fail; here if there is no match. Basic conversion combinators are the `then_conv` operator, which chains two conversions sequentially (and fails if any of these does), and the `else_conv` operator, which attempts to apply the first conversion, but uses the second if the first fails. Because rewrite conversions act like pattern matching guards, we combine them with `else_conv` to recreate the case switching of the algorithm. Combinators like `Conv.arg1_conv` facilitate the rewriting of subterms—this one takes a conversions and modifies it such that it applies to the left operand of a binary operator. With these tools at hand, we can construct the normal form conversion. The recursive pieces can be expressed just as recursive auxiliary functions. However, we lose a bit of generality by using `Conv.arg1_conv` to access the left operand of ($\diamond$), because it requires that ($\diamond$) is not a beta-redex. This could be solved by creating a specialized combinator, which does pattern matching instead of simple term deconstruction.

Figure 5 shows the ML code. It is parameterized by the applicative functor `af`, which contains all the necessary rules. In the helper function `normalize`, the different cases are selected by predicates like `can (dest_comb ctxt af) t`, which is true only if term `t` fits the pattern $\_ \diamond_{\texttt{af}} \_$. We do this here because it is the only recursive conversion which is not guarded by a rewrite rule before recurring, so we want to protect against rampant rewriting. Finally, `rename_rewr_conv` and `rename_rr_conv` (not shown) adjust the names of bound variables in the argument rule to the term to convert. While these names are only hints in Isabelle and obviously subject to alpha conversion, they influence the display of the proof state. To elaborate, consider a HOL term $p \diamond (q \diamond r)$. Without the adjustment, the normal form would be $\mathit{pure}\,(\lambda x x_a x_b.\, x(x_a x_b)) \diamond p \diamond q \diamond r$. The name component x stems from the lambda abstraction in rule `B_intro`. If the term is part of an equation, the variables of the reduced base equation will also be called x, x_a, and x_b. This creates unnecessary confusion, and to help the user, we try to preserve the variable names of the input term.

4 Lifting with Combinators

4.1 Motivation

The normalization approach to solving lifted equations works only if the opaque terms on both sides coincide. This is not true for all equations of interest. Let's revisit the set version of addition of natural numbers, $\oplus$ from Example 1. This operator is also commutative, so it should be possible to prove

$$X \oplus Y = Y \oplus X.$$

After unfolding and normalization, we get

$$\mathit{pure}\,(\lambda x y.\, x + y) \diamond X \diamond Y = \mathit{pure}\,(\lambda y x.\, y + x) \diamond Y \diamond X. \tag{4.1}$$

[5] Isabelle's conversions use the Pure equality $\equiv$.

```
fun normalform_conv ctxt af =
  let
    val rules = facts_of_afun af;

    val leaf_conv = rename_rewr_conv (fn t => [("x", term_to_vname t)])
      (#I_intro rules);
    val merge_conv = Conv.rewr_conv (#merge rules);
    val swap_conv = Conv.rewr_conv (#swap rules);
    val rotate_conv = rename_rr_conv "x" (#B_intro rules);
    val pure_rotate_conv = rename_rr_conv "x" (#B_pure rules);

    fun normalize_pure_nf ct = ((pure_rotate_conv then_conv
      Conv.arg1_conv normalize_pure_nf) else_conv merge_conv) ct;
    val normalize_nf_pure = swap_conv then_conv normalize_pure_nf;
    fun normalize_nf_nf ct = ((rotate_conv then_conv
        Conv.arg1_conv (Conv.arg1_conv normalize_pure_nf then_conv
          normalize_nf_nf)) else_conv
      normalize_nf_pure) ct;

    fun normalize ct =
      let val t = Thm.term_of ct
      in if can (dest_comb ctxt af) t
        then (Conv.arg1_conv normalize then_conv
            Conv.arg_conv normalize then_conv normalize_nf_nf) ct
        else if can (dest_pure ctxt af) t
          then Conv.all_conv ct
          else leaf_conv ct
      end;
  in normalize end;
```

Figure 5: ML implementation of normalization.

Symbol	Reduction
B	$\mathbf{B}xyz = x(yz)$
I	$\mathbf{I}x = x$
C	$\mathbf{C}xyz = xzy$
K	$\mathbf{K}xy = x$
W	$\mathbf{W}xy = xyy$
S	$\mathbf{S}xyz = xz(yz)$
H	$\mathbf{H}xyz = xy(zy)$

Table 2: Useful combinators.

Clearly, this can't be solved with a standard congruence rule, because we would have to to prove that X is equal to Y. Since we are concerned with transferring properties from a base domain, we don't want to assume anything about those opaque subterms.

Hinze showed that such equations can be solved if certain *combinators* can be lifted. Informally, combinators are functions which rearrange their arguments in a specific manner. We have already used two combinators, **I** and **B**. Lifting their defining equations (see Table 2) gives us the identity and composition laws, respectively. If the lifted combinator performs the same rearrangement with arbitrary functorial values, one can translate that particular term structure between the two layers. In this case, we simply say that the combinator *exists*. To continue with (4.1), we could attempt to change the order of Y and X on the right-hand side. Note that these appear as arguments to a pure function. The **C** combinator, also known as 'flip' in functional programming, does what we want: $\mathbf{C}fxy = fyx$. The lifted equation is

$$\mathit{pure}\,\mathbf{C} \diamond f \diamond x \diamond y = f \diamond y \diamond x, \tag{4.2}$$

and it is indeed true for set idiom! From this we get

$$\mathit{pure}\,(+) \diamond X \diamond Y = \mathit{pure}\,(\mathbf{C}(+)) \diamond X \diamond Y. \tag{4.3}$$

The right-hand side is no longer the normal form of $Y \oplus X$, but still a canonical form (which is why we distinguish these two). But now the argument lists on both sides coincide. We reduce to

$$\lambda xy.\, x + y = \lambda xy.\, y + x,$$

which is extensionally equivalent to the base equation $x + y = y + x$. The availability of equation (4.2) is a quite powerful condition, because it will allow us to permute opaque terms freely. If permutations exist such that both sides of an equation in canonical form align regarding their opaque terms, reduction by congruence is possible again. This will again lead to the expected base equation. However, the combinator **C** does not exist for all applicative functors. For example, the order of values in a state monad may be significant.

Combinators appeared originally in the context of logic [22]. They were studied because it is possible to write logical formulas without variables using only applications of suitable combinators, as opposed to the usual lambda calculus. Table 2 lists all combinators which are used throughout this text, together with their defining equations. There are certain sets of combinators which are

sufficient to express all lambda terms, $\{\mathbf{S}, \mathbf{K}\}$ being one of them. In other sets, only a limited part of terms is representable. Hinze's Lifting Lemma shows that all terms and thus all equations can be lifted while preserving the variable structure if $\mathbf{S}$ and $\mathbf{K}$ exist. He also notes that other combinator set are useful, because there are idioms where more than $\{\mathbf{B}, \mathbf{I}\}$, but not all combinators exist. Generally speaking, additional combinators enlarge the set of equations which can be lifted.

The original proof of the Lifting Lemma [5, pp. 11–14] uses induction on the structure of idiomatic terms; it is not entirely obvious how it can be generalized to other combinators sets, as it depends on the availability of $\mathbf{K}$ to lift tuple projections. In this section we present an implementation of this generalized lifting, whose underlying concept works with arbitrary combinators. However, it depends on an abstraction algorithm and the structure of representable terms, which are difficult to derive automatically. Therefore we will restrict ourselves to certain sets ("bases") with fixed algorithms, while understanding that the scope can be extended if needed.

4.2 Generic Lifting

We start with the relationship of combinators and lambda terms. The equations in Table 2 can be expressed as abstractions $\mathbf{I} = \lambda x.\ x$ etc. If we substitute occurrences of combinators in a term (signified by $=_\delta$), new abstractions are introduced, which may be beta-reduced afterwards:

$$\mathbf{WB} =_\delta (\lambda f x.\ f x x)(\lambda g f x.\ g(f x)) =_\beta \lambda x y.\ x(x y).$$

The question arises when and how this process can be reversed, meaning that all abstractions are replaced by suitable combinators. In Curry et. al. [22, Section 6A], terms with variables, but no abstractions are considered. A syntactical operation is defined, denoted $[x]t$, where t is such a term and x is a variable. The desired property is that x does not occur in $[x]t$, and $([x]t)x =_{\delta\beta} t$. Due to its notation, the operation is known as *bracket abstraction*. There is an obvious correspondence with lambda abstractions $\lambda x.\ t$. Bracket abstraction however stands for a concrete applicative term, whereas a lambda is an object of the syntax itself. Replacing lambdas $\lambda x.\ t$ by brackets $[x]t$ performs the shift to a combinator representation. Curry et. al. give several possible definitions for bracket abstraction. They note that these follow a scheme they refer to as an algorithm—a sequence of rules, where each rule is a partial definition. The rules may invoke abstraction recursively. In particular, the following rules are used:

$$\begin{aligned}
[x]x &= \mathbf{I}, && && (i)\\
[x]t &= \mathbf{K}t && \text{if } x \text{ not free in } t, && (k)\\
[x]tx &= t && \text{if } x \text{ not free in } t, && (\eta)\\
[x]st &= \mathbf{B}s([x]t) && \text{if } x \text{ not free in } s, && (b)\\
[x]st &= \mathbf{C}([x]s)t && \text{if } x \text{ not free in } t, && (c)\\
[x]st &= \mathbf{S}([x]s)([x]t). && && (s)
\end{aligned}$$

The algorithm which consists of rules (i), (k) and (s), in that order, is written succinctly as (iks). The algorithm attempts to use the rules in their

left-to-right order, applying the first one whose restrictions are satisfied by the term at hand. Each abstraction algorithm A introduces a certain set of basic combinators, which we refer to as $C(A)$. It is sound only if certain postulates about those combinators, which are again the equations in Table 2, are assumed.

Example 5. Using the (iks) algorithm, one gets

$$[x]xxy \overset{(s)}{=} \mathbf{S}([x]xx)([x]y) \overset{(s),(k)}{=} \mathbf{S}(\mathbf{S}([x]x)([x]x))(\mathbf{K}y) \overset{(i)}{=} \mathbf{S}(\mathbf{SII})(\mathbf{K}y).$$

Attempting to use the $(ik\eta bc)$ algorithm with the same abstraction quickly comes to a stop:

$$[x]xxy \overset{(c)}{=} \mathbf{C}([x]xx)y,$$

which is undefined. ▲

As we can see, not all algorithms are total. Therefore, there is a trade-off between the combinators required and the terms for which abstraction is possible. Bunder [23] presents an analysis of the situation for certain algorithms and combinator sets, based on rigorous definitions for term translation and definability. We will come back to this later, when we discuss how to order the variables in an idiomatic term such that abstraction is defined. For now, the concept of bracket abstraction with the example of rules (i)–(s) is sufficient.

Next, we attempt to transfer these concepts to idiomatic terms. On the one hand, this is quite intuitive since the latter are also formed by an application operator, and pure terms can be identified with constants. But we do not have any "idiomatic abstractions". Hinze actually defines these in terms of abstract combinators and an extensionality property of the idiom. For our purpose it is sufficient to work directly with bracket abstraction, and we assume that all combinators are lifted, i.e. expressible as a pure term. To clarify the following discussion, we adjust our $\mathcal{I}$ formalism and replace opaque terms $\mathsf{term}\, x$ with variables.

Definition 6. The set of generic idiomatic terms $\mathcal{I}'$ is defined by

$$\mathcal{I}' ::= \mathsf{var}\, \mathcal{V} \mid \mathsf{pure}\, \mathcal{T} \mid \mathcal{I}' \text{ `}\mathsf{ap}\text{`} \mathcal{I}'. \tag{4.4}$$

We reuse the congruence $\simeq$ from Definition 2 for generic terms. The set of variables $\mathrm{var}(t)$ of t is defined as the set of all arguments to var occuring in t. The sequence of variables $\overrightarrow{\mathrm{var}}(t)$ is defined similarly to opaq. Unlifting (see Definition 5) is also transferred, but uses the variable x in subterms $\mathsf{var}\, x$ instead of inventing new ones. ▲

Using this definition, it is clear what the rules for idiomatic abstraction are:

$$\begin{aligned}
[x]'(\mathsf{var}\, x) &= \mathsf{pure}\, \mathbf{I}, && && (i')\\
[x]'t &= \mathsf{pure}\, \mathbf{K} \text{ `}\mathsf{ap}\text{`} t && \text{if } x \notin \mathrm{var}(t), && (k')\\
[x]'(t \text{ `}\mathsf{ap}\text{`} \mathsf{var}\, x) &= t && \text{if } x \notin \mathrm{var}(t), && (\eta')\\
[x]'(s \text{ `}\mathsf{ap}\text{`} t) &= \mathsf{pure}\, \mathbf{B} \text{ `}\mathsf{ap}\text{`} s \text{ `}\mathsf{ap}\text{`} [x]'t && \text{if } x \notin \mathrm{var}(s), && (b')\\
[x]'(s \text{ `}\mathsf{ap}\text{`} t) &= \mathsf{pure}\, \mathbf{C} \text{ `}\mathsf{ap}\text{`} [x]'s \text{ `}\mathsf{ap}\text{`} t && \text{if } x \notin \mathrm{var}(t), && (c')\\
[x]'(s \text{ `}\mathsf{ap}\text{`} t) &= \mathsf{pure}\, \mathbf{S} \text{ `}\mathsf{ap}\text{`} [x]'s \text{ `}\mathsf{ap}\text{`} [x]'t. && && (s')
\end{aligned}$$

In general, the algorithm A' on idiomatic terms is obtained from algorithm A on regular terms by lifting its rules in this fashion, preserving order.

Before we show the connection to the canonical form, there is one thing which remains to be considered. The interchange law allows us to move a variable out of the left subterm of an application, given that the right subterm is pure. This is not captured by rules (b') and (i'), which are the only ones from above which are valid in all idioms. We define a combinator $\mathbf{T}xy = yx$ and the rules

$$[x]st = \mathbf{T}t([x]s) \qquad \text{if } t \text{ contains no variables,} \qquad (t)$$
$$[x]'(s \text{ `ap` } t) = \mathsf{pure}\,\mathbf{T} \text{ `ap` } t \text{ `ap` } [x]'s \qquad \text{if } \mathrm{var}(t) = \emptyset. \qquad (t')$$

Soundness of rule (t') can be shown to be equivalent to the interchange law. It is important to understand that $\mathbf{T}$ does not have to exist in the idiom; these rules do not fit exactly in the pattern of the other rules. The $\mathbf{T}$ combinator is also necessary to formulate the most generic rule for the $\mathbf{W}$ combinator. Without the interchange law, it could only be used for terms t `ap` $\mathsf{var}\,x$ `ap` $\mathsf{var}\,x$, i.e. those where the same variable is applied twice in direct succession. In an idiom, there may be arbitrary pure terms "inbetween" the variables. We use the (w) rule when the variable occurs in both operands of an application, just like the (s) rule.

$$[x]st = \mathbf{W}(\mathbf{B}(\mathbf{T}[x]t)(\mathbf{BB}[x]s)) \qquad \text{if } [x]t \text{ contains no variables.} \qquad (w)$$

(w') is derived similarly to the other rules.

As with ordinary terms, we demand a soundness property for idiomatic bracket abstraction, namely that $[x]'t' \text{ `ap` } \mathsf{var}\,x \simeq_C t'$ holds true. The definitions for the additional combinators C get lifted to $\mathsf{pure}\,\mathbf{I} \text{ `ap` } x \simeq_C x$ and so on, consistently extending our congruence relation $\simeq$ to $\simeq_C$.

Lemma 7. *Let $t' \in \mathcal{I}'$ be a generic idiomatic term, and $x \in \mathcal{V}$ a variable. For an abstraction algorithm A' consisting of a subset of rules (i')–(t'), we have $\lfloor [x]'t' = [x] \lfloor t'$ and $[x]'t' \text{ `ap` } \mathsf{var}\,x \simeq_{C(A')} t'$, assuming that $[x]'t'$ is defined. Also, bracket abstraction does not add variables:* $\mathrm{var}([x]'t') \subseteq \mathrm{var}(t')$.

Proof. This statement uses that fact that the rules in A' are very similar to those of A. In particular, rule (r') is applied first when evaluating $[x]'t'$ iff rule (r) is applied first to $[x] \lfloor t'$. The remainder of the proof is a simple induction. We show the case involving rule (c') as an example, the other cases are similar. Thus $t = s \text{ `ap` } u$ with $x \notin \mathrm{var}(u)$. The induction hypothesis is

$$\lfloor [x]'s = [x] \downarrow s \quad \text{and} \quad [x]'s \text{ `ap` } \mathsf{var}\,x \simeq_{C(A')} s \quad \text{and} \quad \mathrm{var}([x]'s) \subseteq \mathrm{var}(s).$$

Then we have

$$\lfloor [x]'t \overset{(c')}{=} \lfloor (\mathsf{pure}\,\mathbf{C} \text{ `ap` } [x]'s \text{ `ap` } u) = \mathbf{C}(\lfloor [x]'s)(\downarrow u)$$
$$\overset{\text{(IH)}}{=} \mathbf{C}([x] \downarrow s)(\downarrow u) \overset{(c)}{=} [x](\downarrow s \downarrow u) = [x] \downarrow t$$

and

$$[x]'t' \text{ `ap` } \mathsf{var}\,x = \mathsf{pure}\,\mathbf{C} \text{ `ap` } [x]'s \text{ `ap` } u \text{ `ap` } \mathsf{var}\,x \simeq_{C(A')} [x]'s \text{ `ap` } \mathsf{var}\,x \text{ `ap` } u$$
$$\overset{\text{(IH)}}{\simeq}_{C(A')} s \text{ `ap` } u = t.$$

□

Base	Abstraction	Example idioms
BI	(ibt)	state, list
BIC	$(ibtc)$	set
BIK	$(kibt)$	
BIW	$(ibtw)$	either
BCK	$(kibtc)$	
BKW	$(kibtw)$	
BICW	$(ibtcs)$	maybe
BCKW	$(kibtcs)$	stream, $\alpha \to$

Table 3: Substructures of BCKW.

Now we can state the key obversation: The successful abstraction of all variables in an idiomatic term leaves a single pure term, per the homomorphism law. Moreover, that term is equivalent to the result of applying the same abstraction algorithm to the "unlifted term". In principle, this works with arbitrary rules, as long as the statements of Lemma 7 hold true.

Theorem 2. *In the following, bracket abstraction uses algorithms A and A' with a subset of the rules (i)–(t) and (i')–(t'), respectively. Let $t' \in \mathcal{I}'$ be a generic idiomatic term, and $x_1, \ldots, x_n$ a permutation of the variables* $\mathrm{var}(t')$*, or a superset thereof. If $f = [x_1] \cdots [x_n] \downarrow t'$ is defined for A, then*

a) $[x_1]' \cdots [x_n]' t'$ consists only of applications of pure terms, and

b) the unique canonical form of $[x_1]' \cdots [x_n]' t'$ is $\mathsf{pure}\, f$;

c) $\mathsf{pure}\, f$ ‘ap‘ $\mathsf{var}\, x_1$ ‘ap‘ $\cdots$ ‘ap‘ $\mathsf{var}\, x_n$ is a canonical form of t';

d) replacing all combinators from $C(A)$ in f with their definitions yields $f' =_{\beta\eta} \lambda x_1 \cdots x_n.\ \downarrow t'$.

Proof. a) is due to $\downarrow([x_1]' \cdots [x_n]' t') = f$ (induction and Lemma 7).

b) It is not difficult to see that a pure-only term p has a unique canonical form, which is equal to $\mathsf{pure} \downarrow p$.

c) We have $\mathsf{pure}\, f$ ‘ap‘ $\mathsf{var}\, x_1$ ‘ap‘ $\cdots$ ‘ap‘ $\mathsf{var}\, x_n \simeq_{C(A)} t'$ by induction, making repeated use of Lemma 7.

d) We show $[x_1] \cdots [x_n] \downarrow t' =_{\delta\beta} \lambda x_1 \cdots x_n.\ \downarrow t'$. Note that $([x_i]p)x_i =_{\delta\beta} p$ for all terms p, and hence $[x_i]p =_\eta \lambda x_i.\ ([x_i]p)x_i =_{\delta\beta} \lambda x_i.\ p$.

□

Remember that we are interested in equations, which obviously consist of two idiomatic terms. We get to the base equation only if the same variable sequence is used for both terms, and the assumptions of Theorem 2 are satisfied. To complete the *generic lifting* approach, we need a procedure for determining the abstraction order. Since this procedure has to depend on the abstraction algorithm, we fix the combinator bases first. Hinze focuses on $\mathbf{SK} = \mathbf{BCKW}$ and $\mathbf{BICS} = \mathbf{BICW}$, noting that **BIC** is also relevant. The set $\{\mathbf{B}, \mathbf{I}, \mathbf{C}, \mathbf{K}, \mathbf{W}\}$ and its subsets seem to be a good starting point to cover relevant cases. The additional combinators play an intuitive role: **C** reorders variables, **W** duplicates

them, and **K** permits abstraction over additional variables. Table 3 lists all distinct subsets containing **B** and **I**, together with the abstraction algorithms we propose. We routinely ignore **T** when listing the combinators. This highlights the connection with combinatorial logic, where some of those bases have been studied.

The algorithms have been chosen in order to simplify the implementation, by using the rules conditionally depending on the available combinators: If **K** exists, start with k. Then, for all bases, perform ibt. If **C** (or **W**) exists, add c (or w), respectively. However, if both do, use rule s instead. Below follows a detailed description of how the variable sequence can be found in each base, and we justify the abstraction algorithms, meaning that the preconditions of Theorem 2 are satisfied. The function $\mathrm{abs}_C(s,t)$ will denote the chosen sequence for terms s, t in the context of base C.

BI

This is the minimal base which is available for all idioms. We already know from Section 3.1 that there is only one canonical form with respect to $\simeq$. Therefore, there is exactly one permissible sequence:

$$\mathrm{abs}_{\mathbf{BI}}(s,t) = \overrightarrow{\mathrm{var}}(s) = \overrightarrow{\mathrm{var}}(t).$$

Equations where the two sequences differ are rejected. If any other sequence could be used, we would get a different canonical form per Theorem 2, thus contradicting the previous result on normal forms. Focusing on a single abstraction step $[x_i]t_i$, x_i must occur once in t_i, and it is the right-most variable. If t_i is an application, there are two cases: x_i occurs in the right subterm, and rule (b) is used successfully. Otherwise, there can be no variable in the right subterm, so (t) applies. This confirms that algorithm (ibt) is indeed acceptable for this base.

BIC

Bunder shows [23] that **BIC**-definable lambda terms are those where each bound variable occurs excatly once, irrespective of their order. His definition of a C-definable term t (with combinator base C) implies that there exists an abstraction algorithm such that $[x]s$ is defined if $t = \lambda x.\, s$. In particular, $(i\eta bc)$ is a valid algorithm. From this it follows that we can choose the order in which we abstract, as long as the corresponding variable occurs exactly one. Note that our special **T** combinator is not considered there. But as it can be simulated by the more powerful **CI**, adding it to the combinator base does not change anything.

In this base, we can work with all equations where the variable sequence of one term can be reordered to the sequence of the second. The order used for the abstraction is irrelevant, but it will be reflected in the quantifier order of the base equation. A simple choice is $\mathrm{abs}_{\mathbf{BIC}}(s,t) = \overrightarrow{\mathrm{var}}(s)$. However note that we do not use the (η) rule, but add the (t) rule. (η) could be a considered as an optimization, since $\lambda x.\, yx =_\beta \mathbf{B}y\mathbf{I}$, so (b) and (i) suffice. Conversely, (t) is a special case of (c) with (i), and adding it to the algorithm next to (c) just results in a slightly different combinator representation. We use this particular

algorithm in order to simplify the implementation, such that as much code as possible can be shared between the bases.

BCK and BICW

These cases were also analyzed by Bunder. For **BCK**, the definable terms are those where each bound variable occurs at most once, again ignoring the order. In terms of bracket abstraction, $[x]y$ is then also defined if x is not free in y. This allows us to extend the sequence with other variables. We make use of this to deal with variables which only occur on one side of an equation—it does not hurt to abstract over those too. Hence, $\mathrm{abs}_{\mathbf{BCK}}(s,t)$ can be any arrangement of the set $\mathrm{var}(s) \cup \mathrm{var}(t)$. The implementation uses a total order on the set of variables. On the contrary, the definable terms of **BICW** have at least one occurence of each bound variable; $[x]y$ can therefore be used if x occurs multiple times in y, and x will not be free in $[x]y$. Comparing this with **BIC**, we loosen the restriction that variables must not be repeated. $\mathrm{abs}_{\mathbf{BICW}}(s,t)$ must be an arrangement of $\mathrm{var}(s) = \mathrm{var}(t)$. We can compute this by sorting the sequences again, and then trimming duplicates which now are next to each other.

Bunder proposes the $(i\eta kbc)$ algorithm for **BCK**, and $(i\eta bcs)$ for **BICW**; our rules are $(kibtc)$ and $(ibtcs)$, respectively. The comments on (η) and (t) from above apply here as well. The only other difference is the position of (k) in the list. But (k) on one side and (i) and (η) on the sider work with disjoint sets of terms, so this is not an issue.

BCKW

This base, which is logically equivalent to **SK**, has some useful properties: $[x]y$ can always be defined, making it possible to use any abstraction sequence, and in turn handle all equations. In conjunction with Theorem 2, we want to abstract all free variables, though. Therefore we determine the sequence as with **BCK**, but the term restriction is rescinded. As for the abstraction algorithm, we use a variation of $(ik\eta bcs)$ from [22].

BIK, BIW and BKW

These bases do not appear to be significantly covered in the literature. Since there is at least one useful example of an idiom with **BIW** combinators, we still shall implement and discuss them. By adding the **K** combinator to **BI**, additional variables may be abstracted, but the other restrictions (order, single occurrence) remain. Accordingly, we demand that $\mathrm{abs}_{\mathbf{BIK}}(s,t)$ has $\overrightarrow{\mathrm{var}}(s)$ and $\overrightarrow{\mathrm{var}}(t)$ as subsequences. For definiteness, we can limit it to the shortest sequence where variables only in s appear before those only in t, whenever there is an ambiguity: $\mathrm{abs}_{\mathbf{BIK}}(s,t) = \mathrm{abs}'_{\mathbf{BIK}}(\overrightarrow{\mathrm{var}}(s), \overrightarrow{\mathrm{var}}(t))$,

$$\mathrm{abs}'_{\mathbf{BIK}}(a,b) = \begin{cases} x \cdot \mathrm{abs}'_{\mathbf{BIK}}(a',b') & \text{if } a = x \cdot a',\ b = x \cdot b'; \\ x \cdot \mathrm{abs}'_{\mathbf{BIK}}(a',b) & \text{if } a = x \cdot a', \\ x \cdot \mathrm{abs}'_{\mathbf{BIK}}(a,b') & \text{if } b = x \cdot b', \\ \langle\rangle & \text{if } a = b = \langle\rangle. \end{cases}$$

Trying rule (k) in the beginning of the abstraction algorithm obviously takes care of unused variables.

For the other bases, we simply take the abstraction algorithms as granted. Based on that we analyze the set of defined bracket abstractions. Since **W** duplicates variables, one might conjecture that a single variable may now occur repeatedly. No other variable may be interspersed in the lexical order, though, because none of the available combinators reorder their arguments. The following lemma proves that this intuition is correct. In order to express it on the level of lambda terms, we overload $\overrightarrow{\mathrm{var}}(x)$ in the obvious way. We use the notation x^n with sequence x to stand for n concatenated copies of x.

Lemma 8. *Let $x \in \mathcal{V}$ and $t \in \mathcal{T}$. With algorithm (ibtw), $[x]t$ is defined iff there is a natural number $n \geq 1$ and a variable sequence v such that $\overrightarrow{\mathrm{var}}(t) = v \mathbin{@} \langle x \rangle^n$, where x does not appear in v. The same statement holds true for algorithm (kibtw), except that n may be zero as well.*

Proof. The first part of the proof is concerned with the direction where $[x]t$ is assumed to be defined. We perform induction over the steps of the abstraction algorithm. In the evaluation of $[x]s$ for a subterm s of t, there are the following cases: If (k) is used, then x must not be free in s, thus $n = 0$ and $v = \overrightarrow{\mathrm{var}}(s)$. For (i), we have $n = 1$ and $v = \langle\rangle$. For (b), there must be terms u and w with $s = uw$, and x is not free in u. From the induction hypothesis we get adequate n' and v' for $[x]w$. Then n' and $\overrightarrow{\mathrm{var}}(u) \mathbin{@} v'$ satisfy the conditions for $[x]s$. Rules (t) and (w) are analogous: For (t), the side condition implies that $\overrightarrow{\mathrm{var}}(w) = \langle\rangle$; for (w) we have $\overrightarrow{\mathrm{var}}(w) = \langle x \rangle^k$ for some $k \geq 1$, thus $\overrightarrow{\mathrm{var}}(u) \mathbin{@} \overrightarrow{\mathrm{var}}(w) = v' \mathbin{@} \langle x \rangle^{n'+k}$, where n' and v' are from $[x]u$.

Now we show the other direction. Assume that suitable n and v exist. If $n = 0$, rule (k) is used, because x cannot be free in t. Otherwise we assume $n \geq 1$ during an induction on the structure of t. If t is a variable, it must be x, so rule (i) applies. If it is an application $s = uw$ instead, there are three cases: $x \notin \mathrm{var}(u)$, hence $v = \overrightarrow{\mathrm{var}}(u) \mathbin{@} v'$ for some v', and $[x]w$ is defined by the induction hypothesis so we can use rule (b). If $\mathrm{var}(w) = \emptyset$, $[x]u$ is defined and rule (t) applies. Otherwise, $\overrightarrow{\mathrm{var}}(w) = \langle x \rangle^k$ with $k \leq n$ must hold, so (w) can be used. □

It follows that the abstraction sequences for **BIW** and **BKW** should be chosen like those of **BI** and **BIK**, respectively; but for each variable, a single span of repetitions in $\overrightarrow{\mathrm{var}}(s)$ and $\overrightarrow{\mathrm{var}}(t)$ is allowed and treated like a single instance.

4.3 Implementation

The implementation of generic unlifting essentially has to compute bracket abstractions, i.e., recursively select the correct rule for a term. Similar to what we did with the normal form conversion, the goal is to prove a theorem representing

$$t \simeq ([x_1]' \cdots [x_n]' t) \text{ `ap` } x_1 \text{ `ap` } \cdots \text{ `ap` } x_n.$$

However, we also add a heuristic that tries to determine how variables have been instantiated in the supplied equation. First, arbitrary opaque terms take the place of variables. We can work directly with these, but it becomes less clear which subterms are the same, meaning that they should be abstracted simultaneously. Consider a base equation with two, universally quantified variables x

and y. If the lifted version can be proven, it will also have two variables. Now, the user might want to prove a proposition which is equivalent to the lifted equation, except that both variables have been replaced by the (same) term z. This modified statement is obviously true, so we might expect that the lifting tool can handle it. Remember that we prohibited multiple occurrences in certain bases, e.g. **BI**, when we defined bracket abstraction. What would happend if we normalize the equation? Normalization treats each instance of a opaque subterm separately, thus we will get something of the form $\mathit{pure}\, f \diamond z \diamond z = \mathit{pure}\, g \diamond z \diamond z$. By magic, the functions f and g have been equipped with two arguments! In the base **BI**, generic unlifting is not more powerful than normal form conversion, but we expect it not to be weaker either. We solve the problem by handling each opaque subterm as if it were a different variable. For other bases, we need to be more clever, or we would lose the advantage that generic unflifting was supposed to give us. For example, idempotency of an operator $\circ$, $x \circ x = x$ can be lifted if **W** exists; when proving a lifted instance like $f(x) \hat{\circ} f(x) = f(x)$, the two copies of $f(x)$ on the left-hand side must be identified. Therefore, the base of the idiom influences the assignment of "virtual" variables to opaque terms. In general, it is best if we identify as many terms as possible: The resulting base equation is certainly provable if it were so without grouping some pair of variables. However, we will see that the situation can sometimes be ambiguous.

```
1  datatype apterm =
     Pure of term | Var of int * term | Ap of apterm * apterm;
```

Figure 6: The datatype to keep track of idiomatic terms.

We use a ML datatype to make the term structure explicit. `apterm` (see Figure 6) has a similar structure as $\mathcal{I}$. Opaque terms `Var` are tagged with an integer, which is unique for each position in an equation. Sets of these positions play the role of variables in the previous section. The function `eliminate` takes an `apterm` t and such a set x, computing the pair $([x]'t, \phi)$, where ϕ is the theorem $t \simeq [x]'t$ `` `ap` `` x. One important building block for this function is `rewr_subst_ap`, whose code is shown in Figure 7. It combines three rewrite rules. Two of them, say $x = x'$ and $y = y'$, get merged to a congruence rule of the idiomatic application operator, $x \diamond y = x' \diamond y'$, by the standard `Drule.binop_cong_rule`. However, the application operator's types need to be instantiated before. The resulting equation is then chained with the third rule. Similar to our use of conversion combinators in Section 3.2, we employ `rewr_subst_ap` to handle the recursive nature of most abstraction rules. As an example refer to Figure 8. It takes as arguments two pairs of an idiomatic term and a theorem, the results of abstracting the two subterms of an application. Here, the key transformation is again provided by `B_intro` from Table 1. We must also update the `apterm` structure. New combinator terms are introduced, which also need proper type instantiation. Because these combinator are part of the equation we have computed, it seems to be the easiest method to extract the term from that equation. The integer supplied to `extract_comb` indicates how deeply nested to the left the combinator is.

The function `consolidate` determines the common abstraction sequence for two idiomatic terms. It begins with creating a left-to-right list of the variables

```
fun rewr_subst_ap ctxt af rewr thm1 thm2 =
  let
    val funT = thm1 |> Thm.lhs_of |> Thm.typ_of_cterm;
    val ap_inst = Thm.cterm_of ctxt (ap ctxt af funT);
    val rule1 = Drule.binop_cong_rule ap_inst thm1 thm2;
    val rule2 = Conv.rewr_conv rewr (Thm.rhs_of rule1);
  in Thm.transitive rule1 rule2 end;
```

Figure 7: The `rewr_subst_ap` function.

```
fun extract_comb n thm = Pure (thm |> Thm.rhs_of |>
  funpow n Thm.dest_arg1 |> Thm.term_of);
fun comb2_step def (tt1, thm1) (tt2, thm2) =
  let val thm = rewr_subst_ap def thm1 thm2;
  in (Ap (Ap (extract_comb 3 thm, tt1), tt2), thm) end;
val B_step = comb2_step (#B_intro rules);
```

Figure 8: Implementation of rule (b').

for each term. Then an additional tag is added such that two list elements have the same tag iff we consider the instantiated terms equal. In the current implementation, the equality predicate is plain alpha convertibility. The next steps depend on the combinator base:

a) If **C** exists, each list is sorted according to the tag. This groups equal terms together.

b) If **W** exists, adjacent copies of the same term (again as indicated by the tag) are merged in each list. The positions of the involved variables are collected into a list. Otherwise, each position is put in an singleton list, so we have a uniform type after this step.

c) If **K** exists, both lists are merged in a peculiar way: If the heads of them have the same tag, we merge their positions to obtain a single element and add it to the result, then proceeding with the tails. Otherwise, we try to align one head with the other list and vice versa. A head is aligned if its tag occurs in the list after some prefix. If alignment is possible in one direction, but not the other, the prefix and the aligned element become part of the result. Otherwise, the two heads are emitted one after the other. A potentially remaining single tail is kept as-is. Figure 9 displays an example.

   If **K** does not exist, we check that both lists have the same length and contain the same sequence of tags, or an error condition is raised. Again, the position lists within both lists are zipped.

We should investigate whether this algorithm is consistent with the theoretical development in Section 4.2, and also with the criteria regarding variable instantiation.

- With **BI**, there is not much flexibility—we can only abstract over a single position at a time. Aborting due to a sequence mismatch is basically the

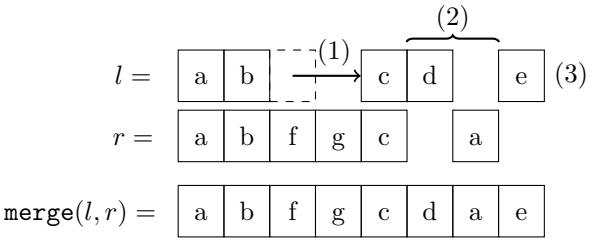

Figure 9: Example of the merge operation for **K** in `consolidate`. (1) Variable c can be aligned with the remainder of list r, which is not possible with f and list l. (2) Neither d nor the copy of a can be aligned, so we intersperse them. The previous occurrence of a is not considered, because the algorithm is greedy and works from left to right. (3) A single tail is preserved.

single difference to normal form conversion. However, there we will just get a likely unprovable result, so this behavior is justified.

- **BIC**: As before; sorting produces a suitable sequence if one exists.
- **BIW**: We have shown that simultaneous abstraction is only possible for coherent spans of a variable. Merging all adjacent copies of a variable is the best we can hope for.
- **BICW**: Sorting puts equal terms next to each other, where they will be merged. Thus, all possible terms are identified, making this the most general solution. If the sanity check at the end fails, then the set of variables for both sides of the equation must differ, and the precondition for this base is violated.
- **BCKW** differs from the previous case in the last step. As expected, we now accept every pair of terms. It should not happen that we will get a duplicate variable tag, in other words, if a tag appears in both lists, there must be a single element in the output. This is easily shown by induction. Especially if alignment is used, the "skipped" prefix can only contain variables which do not occur in both terms, because the lists are sorted.
- **BCK**: The lack of **W** implies that all positions have to be abstracted individually. However, it is interesting to see what the last step does with respect to consolidating both sides of the equation. Again by induction we see that as many pairings as possible between the two sides of the equation are created. The difference to **BCKW** is that the same term may occur repeatedly in the lists. In case of different repetition counts, the algorithm is biased to the left. This is a potentially destructive choice, but the user can always resort to using new variables in the desired manner, and then do the instantiation in a second step.
- **BIK** and **BKW** are not perfectly solvable either. For example, consider the lists (tags only) $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$. If we try to consolidate, then $\langle 1, 2, 1\rangle$ or $\langle 2, 1, 2\rangle$ are the only possible sequences. It is not clear which is the correct one. The algorithm bails out and selects $\langle 1, 2, 2, 1\rangle$ (which is probably

```
lemma set_plus_assoc: (X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z)
proof
  show X ⊕ Y ⊕ Z ⊆ X ⊕ (Y ⊕ Z) proof
    fix a assume a ∈ X ⊕ Y ⊕ Z
    then obtain x y z
      where elems: x ∈ X  y ∈ Y  z ∈ Z
      and sum: a = x + y + z
      unfolding set_plus_def ap_set_def by blast
    from sum have a = x + (y + z) using add.assoc by simp
    with elems show a ∈ X ⊕ (Y ⊕ Z)
      unfolding set_plus_def ap_set_def by blast
  qed
next
  show X ⊕ (Y ⊕ Z) ⊆ X ⊕ Y ⊕ Z proof
    symmetric proof omitted ...
  qed
qed
```

Figure 10: A semi-manual proof of the associative property of addition, lifted to sets.

useless). Likewise, discretion on behalf of the user is required. One could try more sophisticated algorithms like minimizing the edit distance. Due to a lack of example idioms for these bases, we decided against this extra effort.

5 Usage Examples

5.1 User Interface

This section demonstrates the user interface of our package by revisiting Example 1. We assume that $\oplus$ and $\diamond$ for sets have been defined according to equations (1.5) and (1.4), and "set_plus_def" and "ap_set_def" refer to these equations. It is possible to prove the associative property with the standard facilities of Isabelle/HOL, of course. Figure 10 shows a canonical Isar proof. Its structure follows natural reasoning about sets: We prove set equality by showing mutual inclusion in both directions, each of which is proven by the corresponding implication. The operators in the term $a \in X \oplus Y \oplus Z$ are unfolded to obtain nested set comprehensions. The fully automatic proof method *blast* is fortunately able to deduce the relation to elements of the individual sets X, Y, Z. Also note how the fact "add.assoc" is used in the middle—it states the base equation.

The proof scheme can be adjusted for other properties. It is, however, suitable only for the set idiom. Inductive datatypes would likely need induction (or case splits for non-recursive types), coinductive datatypes use coinduction, and so on.

Now we want to automate the full proof with our package. First, it needs to be informed about the set idiom. We provide a command to declare applicative functors:

```
applicative set (C) for
```

pure: $\lambda x.\ \{x\}$
ap: $\diamond_{set}$
unfolding ap_set_def **by** fast+

Its general syntax is

applicative *name* (*combinator*, ...)
for
pure: $pure_f$
ap: $\diamond_f$
proof

The idiom will be made available under the *name*. It can be used to refer to the idiom manually in proofs. The name is followed by an optional list of a subset of the symbols C, K, and W. These declare which combinators are lifted, as explained in Section 4. The set idiom only lifts the **C** combinator. The functions *pure* and $\diamond$ imply the type scheme, see also Section 2.3. Finally, the idiom laws need to be proven. The system presents the user with corresponding goals, which are solved by the proof. For sets, the goals are

1. $\bigwedge x.\ \{\lambda x.\ x\} \diamond x = x$
2. $\bigwedge gfx.\ \{\lambda gfx.\ g(fx)\} \diamond g \diamond f \diamond x = g \diamond (f \diamond x)$
3. $\bigwedge fx.\ \{f\} \diamond \{x\} = \{fx\}$
4. $\bigwedge fx.\ f \diamond \{x\} = \{\lambda f.\ fx\} \diamond f$
5. $\bigwedge fxy.\ \{\lambda fxy.\ fyx\} \diamond f \diamond x \diamond y = f \diamond y \diamond x$

In this example, the proof obligations are easily discharged by unfolding and the automatic *fast* prover. After the command has been issued, the functor can be used in subsequent commands. Its data is stored in the theory context and thus can be imported along other theory content. This allows the construction of a reusable idiom library.

Next, the definition of $\oplus$ needs to be registered. Otherwise, the proof method is not able to interpret a term like $X \oplus Y$ as a composite idiomatic expression. Lifted constants can be registered with the attribute *applicative_unfold*, which can be applied to facts *lhs* = *rhs*, where *rhs* is an idiomatic expression. The equation must be suitable for rewriting.

Finally, we are able to compress the proof to a single invocation of the new proof method *applicative_lifting*. The base equation is provided as a fact and is used automatically by the supporting Isar framework:

lemma set_plus_assoc: $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$
using add.assoc **by** applicative_lifting

The proof method can also be used when the variables in the lifted equation have been instantiated with other terms.

References

[1] John Harrison. "A Short Survey of Automated Reasoning". In: *Algebraic Biology*. Ed. by Hirokazu Anai, Katsuhisa Horimoto, and Temur Kutsia. Vol. 4545. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 334–349.

[2] Timothy Bourke et al. "Challenges and Experiences in Managing Large-Scale Proofs". In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring et al. Vol. 7362. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 32–48.

[3] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002.

[4] Brian Huffman and Ondřej Kunčar. "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL". In: *Certified Programs and Proofs*. Ed. by Georges Gonthier and Michael Norrish. Vol. 8307. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 131–146.

[5] Ralf Hinze. "Lifting Operators and Laws". 2010. URL: `http://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf` (visited on June 6, 2015).

[6] Conor McBride and Ross Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.01 (2008), pp. 1–13.

[7] Philip Wadler. "Monads for Functional Programming". In: *Advanced Functional Programming*. Ed. by Johan Jeuring and Erik Meijer. Vol. 925. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 24–52.

[8] Philip Wadler. "How to Replace Failure by a List of Successes: A Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages". In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Vol. 201. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1985, pp. 113–128.

[9] Ralf Hinze. "Functional Pearl: Streams and Unique Fixed Points". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. ACM, 2008, pp. 189–200.

[10] Ralf Hinze. "The Bird Tree". In: *Journal of Functional Programming* 19 (05 2009), pp. 491–508.

[11] Lawrence C. Paulson. "Isabelle: The Next 700 Theorem Provers". In: *Logic and Computer Science*. Ed. by Piergiorgio Odifreddi. Vol. 31. APIC Studies in Data Processing. Academic Press, 1990, pp. 361–386.

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

[13] Makarius Wenzel. *The Isabelle/Isar Implementation*. May 25, 2015. URL: `http://isabelle.in.tum.de/dist/Isabelle2015/doc/implementation.pdf`.

[14] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. May 25, 2015. URL: `http://isabelle.in.tum.de/dist/Isabelle2015/doc/isar-ref.pdf`.

[15] Markus Wenzel. “Isar – A Generic Interpretative Approach to Readable Formal Proof Documents”. In: *Theorem Proving in Higher Order Logics*. Ed. by Yves Bertot et al. Vol. 1690. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 167–183.

[16] Markus M. Wenzel. “Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents”. Dissertation. Technische Universität München, 2002.

[17] Martin Wildmoser and Tobias Nipkow. “Certifying Machine Code Safety: Shallow Versus Deep Embedding”. In: *Theorem Proving in Higher Order Logics*. Ed. by Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan. Vol. 3223. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 305–320.

[18] Amine Chaieb and Tobias Nipkow. “Verifying and Reflecting Quantifier Elimination for Presburger Arithmetic”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Geoff Sutcliffe and Andrei Voronkov. Vol. 3835. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 367–380.

[19] Andreas Schropp and Andrei Popescu. “Nonfree Datatypes in Isabelle/HOL”. In: *Certified Programs and Proofs*. Ed. by Georges Gonthier and Michael Norrish. Vol. 8307. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 114–130.

[20] Andreas Schropp. “Instantiating Deeply Embedded Many-sorted Theories into HOL Types in Isabelle”. Master Thesis. Technische Universität München, 2012.

[21] Lawrence Paulson. “A Higher-Order Implementation of Rewriting”. In: *Science of Computer Programming* 3.2 (1983), pp. 119–149.

[22] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*. Vol. 1. North-Holland Publishing Company, 1968.

[23] Martin W. Bunder. “Lambda Terms Definable as Combinators”. In: *Theoretical Computer Science* 169.1 (1996), pp. 3–21.