Applicative Functors in Isabelle/HOL: Notes

Joshua Schneider

October 12, 2015

1 Project Overview

1.1 Introduction

Our primary goal is to implement an Isabelle/HOL proof method which reduces lifted equations to their base form. Here, lifting refers to a transition from operations on base types to related operations on some structure. Hinze [1] studied the conditions under which lifting preserves the validity of equations. He noticed that lifting can be defined in an intuitive fashion if the target structure is an applicative functor [2]: a unary type constructor f with associated constants[1]

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

The operator $\diamond_f$ is left-associative. We omit the subscripts if the functor is clear from the context. Moreover, the following laws must be satisfied:

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

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 under the identity functor interpretation, or identity "idiom" as coined in [2]. By choosing a different applicative functor, we obtain a different interpretation of the same term structure. In fact, this is how we define the lifting of t to an idiom. We also permit variables, which remain as such in the lifted term, but range over the structure instead. A term consisting only of pure and $\diamond$ applications and free variables is called an idiomatic expression.

Example 1. Another applicative functor can be constructed from sets. For each type α there is a corresponding type $\alpha\,\mathit{set}$ of sets with elements in α; pure denotes the singleton set constructor $x \mapsto \{x\}$; $F \diamond X$ takes a set of functions F

[1] Types are given in Isabelle notation.

and a set of arguments X with compatible type, applying each function to each argument:

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

We can lift addition on natural numbers to the set idiom by defining the operator

$$(\oplus) :: \mathit{nat\,set} \Rightarrow \mathit{nat\,set} \Rightarrow \mathit{nat\,set},$$
$$X \oplus Y = \mathit{pure}\,(+) \diamond X \diamond Y = \{x + y \mid x \in X,\, y \in Y\}.$$

The associative property of addition

$$\forall xyz.\, (x + y) + z = x + (y + z)$$

can be translated to sets of natural numbers

$$\forall XYZ.\, (X \oplus Y) \oplus Z = X \oplus (Y \oplus Z),$$

where it holds as well, as one can check with a slightly laborious proof. Note that the two sides of the latter equation are the lifted counterparts of the former, respectively. ▲

As we have seen, lifting can be generalized to equations. There is actually a more fundamental relationship between the two equations from above example—the lifted form can be proven for all applicative functors, not just *set*, using only the base property and the applicative functor laws. We want to automate this step with a proof method.

Not all equations can be lifted in all idioms, though. 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.

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

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

Here the effect of $\{\}$ is that it cancels out everything else if it occurs somewhere in an idiomatic expression. This makes it impossible to lift any equation with a variable occuring only on one side to *set*. ▲

1.2 User Interface

Since Isabelle's core logic does not allow parameterization of type constructors, we need a custom mechanism for registering applicative functors with the system. In order to apply the proof method, the user must provide beforehand

a) corresponding *pure* and $\diamond$ instances, and

b) a proof of the applicative functor laws, optionally with extended properties.

Lifted constants may be registered with an attribute, which can be applied to facts $lhs = rhs$, where rhs is an idiomatic expression. These must be suitable for rewriting.

The complete set of subgoal forms to support has not been determined yet. $\boxed{\text{To do.}}$ As a minimal requirement, after unfolding lifted constants, HOL equations of idiomatic expressions shall be handled. Only the outermost functor f is considered per invocation. The conceptual variables of the lifted expressions may be instantiated with arbitrary terms. However, the method actually proves the fully universally quantified form—for every subterm not matching $pure_f\ \_$ or $\_ \diamond_f \_$, a new, locally quantified variable is introduced. We call these subterms *opaque*. The method attempts to transform the first subgoal to the base form of the equation, in other words, its identity functor interpretation. Variable names shall be preserved, if possible. Finally, it is desirable to have some kind of debugging facility for tracing intermediate steps. This may be especially useful if the proof fails because additional laws are required.

Example 3. Continuing with the set idiom from Example 1, assume that the user wants to prove an instantiation of the associativity law for $\oplus$ as part of a larger proof. *set* and $\oplus$ have been declared to the enclosing theory. The current Isabelle goal state is

$$1. \quad (X \oplus Y) \oplus f Z = X \oplus (Y \oplus f Z).$$

The variables X, Y and Z have been fixed in the proof context, and f is some constant, all of appropriate type. This illustrates how we allow a larger variety of propositions, making it easier to apply the method without too much preparation. After applying the proof method, the new proof obligation reads

$$1. \quad \bigwedge xyu.\ (x + y) + u = x + (y + u),$$

which is easily discharged. The corresponding Isabelle/Isar fragment could be

fix $X\ Y\ Z$
...
have "$(X \oplus Y) \oplus f Z = X \oplus (Y \oplus f Z)$" **by** *af_lifting algebra*

af_lifting is our new proof method, and the standard *algebra* method completes the subproof. ▲

1.3 Proof Strategy

The proof method starts with testing the first subgoal for the expected structure. If the test succeeds, the applicative functor f is known, such that the relevant theorems can be accessed subsequently. 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 simple equation of two idiomatic expressions.

The following step depends on which additional properties of f have been provided. All approaches have in common that both expressions are replaced by others in *canonical form*:

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

If either $m \neq n$ or $t_i \neq s_i$ for some i (as terms modulo $\alpha\beta\eta$-conversion), the proof method fails. Otherwise, we apply appropriate congruence rules until the subgoal is reduced to $g = h$. 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 [1, p. 7] asserts the existence of a certain normal form for idiomatic expressions where each variable occurs only once. This normal form has the desired structure. As it turns out, we can compute it for arbitrary terms. This is convenient because opaque parts are handled implicitly. However, the result might be too general and unprovable, so we will use the normal form only if no other properties are available. The details of the normalization algorithm are described in Section 3. There we will also show that the transformed equation is exactly the generalized base form of the original equation.

Hinze then explored under what circumstances a larger variety of equations can be lifted. He found that sufficient conditions can be expressed in terms of combinators as known from combinatory logic. We take this idea and adapt it our framework in Section 4. In contrast to the normal form approach, we gain flexibility regarding the opaque terms in the canonical form. This means that the algorithm must determine the sequence $\vec{t} = \vec{s}$ of opaque terms prior to the transformations. The set of available combinators further limits the admissible sequences. Because this relationship cannot be generalized easily, we restrict ourselves to certain combinator sets.

1.4 Choice of Embedding

In Isabelle, it is not possible to construct an abstract framework for applicative functors in such a way that it is inhabited by all instances. 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. This is a *deep embedding* [3] of the logic. Then it would be possible to derive the Normal Form Lemma as a regular inference rule, for example. However, we want to prove propositions about arbitrary HOL objects, not just their encodings in the embedded logic. Some machinery is necessary, which performs the encoding and transfers results.

- finite number of types involved per term $\Longrightarrow$ could use sum types
- number of types in sum is linear in size of terms

- would introduce a large number of projections/abstractions

$\boxed{\text{To do.}}$

A different approach, which we will take, is a *shallow embedding.* The "formulæ" (here, idiomatic terms) are expressed directly in HOL. Due to aforementioned restrictions, meta-theorectical results must be provided in specialized form for each case. We make use of the powerful ML interface of Isabelle to program the proof construction. The correctness of the proofs is still verified by the system, of course.

2 Background

2.1 Proving with Isabelle

Isabelle was originally designed as a framework for interactive theorem proving, without being restricted to a specific logical system [4]. 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 [5]. It implements the higher-order logic which was used in the HOL system [6], another proving environment. 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. We will further discuss the choice of HOL for the extension presented here in Section $\boxed{\text{To do.}}$.

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: $(\alpha, \beta)\mathit{fun}$. *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 $(\alpha, (\beta, \gamma)\mathit{fun})\mathit{fun}$. 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 4. We already introduced the type *nat*. Number literals can be used directly. Common arithmetic operators are available, like[2] $\mathit{plus} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$. We can also use infix operators:

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

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 $\alpha \Rightarrow \mathit{bool}$. Then $\{x.\ P x\}$ is the set of those values $x :: \alpha$ such that $P x$ is true, and $\{f x \mid x.\ P x\}$ 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

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

states that addition of natural numbers is commutative. Note that $=$ is just another operator of polymorphic type $'a \Rightarrow' a \Rightarrow \mathit{bool}$. Internally, quantifiers are represented as constants applied to lambda abstractions, which handle the variable binding. ▲

In order to achieve the goal of supporting different object-logics, Isabelle contains an immediate layer, the meta-logic Pure. It is an "intuitionistic fragment of higher-order logic" [7, p. 27]. In Pure itself there is only the type *prop* of propositions. A term of this type combined with a proof relative to some context forms a *theorem*. For now, the context is an abstract entity which may contain local hypotheses. Assumptions can also be recorded explicitly in a proposition, using meta-implication $\Longrightarrow$. Note that this is technically different from HOL's implication $\longrightarrow$, though there are theorems which allow conversion between the two. The meta-quantifier $\bigwedge$ denotes universal quantification (with a similar relationship to $\forall$); it is used to restrict the scope of variables in assumptions. Meta-equality $\equiv$ is the third main operator of Pure. The generic rewriting and simplification tools work with such equations. Again, conversion with $=$ is possible. Object-logics embed their own notion of a proposition into Pure via a truth judgement. In HOL, this is the constant $\mathit{Trueprop} :: \mathit{bool} \Rightarrow \mathit{prop}$, which turns an object-level formula into a proposition stating that said formula is indeed true. It is usually left implicit. See [8, Chapter 2] for further details about Pure.

Pure has two main uses within the Isabelle framework: representation and manipulation of deduction rules, and goal states. For the former, consider the traditional introduction rule for conjunction,

$$\frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \cup \Gamma_2 \vdash P \wedge Q},$$

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

which we want to turn into a theorem. Deduction in Isabelle handles the contexts Γ_i automatically, and therefore do not have to be stated in the corresponding proposition. The dependency of the conclusion $P \wedge Q$ on the hypotheses P and Q translates to repeated meta-implication. This gives us the proposition

$$\bigwedge PQ.\, P \Longrightarrow Q \Longrightarrow P \wedge Q$$

(the *Trueprop* markers have been omitted, and P, Q range over *bool*). There is one quirk, however. The outermost meta-quantified variables (and all type variables) are turned into schematic variables, which are free variables distinguished by the prefix *?*. Thus, the introduction rule as it is supplied by HOL appears as

$$\texttt{conjI}: \quad ?P \Longrightarrow ?Q \Longrightarrow ?P \wedge ?Q.$$

Schematic (type) variables are eligible for instantiation during *resolution*, which is Isabelle's primary tool for proof construction. Resolution combines two rules (theorems), identifying an assumption of the second with the conclusion of the first by higher-order unification. Additionally, the first rule is brought into the context of that assumption.

A goal state is a theorem that represents a partially completed proof of some proposition, the goal. While the proof is incomplete, the remaining subgoals are tracked as assumptions. Proof steps transform the goal state. The proof is finished once only the goal remains.

In contemporary use of Isabelle, user input to the system is expressed in the Isar language [9, 10, 7]. 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. Commands constitute the so-called outer syntax of Isar. Terms and types occuring within them are parsed separately, according to the inner syntax. They are usually embedded in quotes '`"`...`"`' to disambiguate them. In certain cases, a command may put the theory state into proof mode. After a proof is finished, the associated goal becomes a *fact*.

Some commonly used specifications are:

- The **definition** command introduces new constants by means of defining equations.
- $\boxed{\text{To do.}}$ **datatype**, **primrec**
- Facts can be given names for further use in proofs. The canonical command for this is **lemma** and its variants **theorem** and **corollary**.
- $\boxed{\text{To do.}}$ **locale**?

Finally, there two syntactical categories which are repeatedly used in commands: *Proof methods* denote (possibly parameterized) operations on the goal state. *Attributes* invoke further processing steps on facts, either transforming them or causing additional declarations.

3 Normal Form Conversion

McBride and Paterson [2] 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 [1, 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 1.1, 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 3, 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

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

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

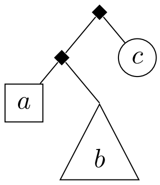

Figure 1: $(\mathsf{pure}\, a$ ‘ap‘ $b)$ ‘ap‘ $\mathsf{term}\, c$ as a tree.

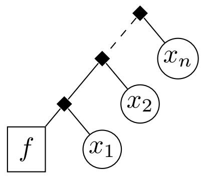

Figure 2: A term in canonical form.

$=_{\alpha\beta\eta}$ 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:

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

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

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

▲

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

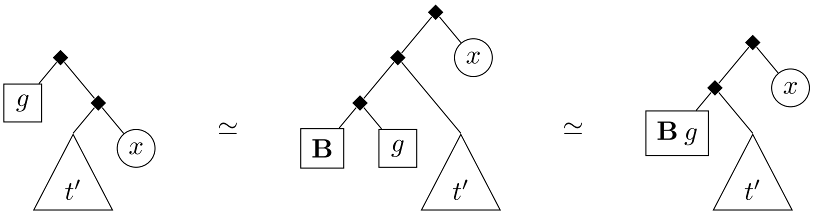

Figure 3: The “pure-rotate” step.

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

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

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

Lemma 1. *For all $g \in \mathcal{T}$ and $t \in \mathcal{C}$,* pure-nf($\mathsf{pure}\, g$ ‘ap‘ t) *is well-defined, and*[3] 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 $t = t'$‘ap‘$\mathsf{term}\, x$ 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

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

[3] $a \in S \simeq b$ abbreviates “$a \in S$ and $a \simeq b$”.

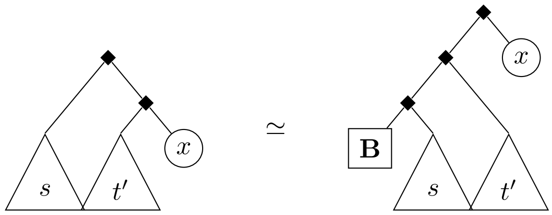

Figure 4: The "rotate" step.

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

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

Going back to u' ‘ap‘ 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 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 $s, t \in \mathcal{C}$,* nf-nf$(s$ ‘ap‘ $t)$ *is well-defined, and* nf-nf$(s$ ‘ap‘ $t) \in \mathcal{C} \simeq s$ ‘ap‘ 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.

Algorithm 1 Normalization of idiomatic terms.

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

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

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

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

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 $t \in \mathcal{I}$,* normalize t *is well-defined, and* 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. ▲

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

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

where

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

▲

Example 5. 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 $\mathrm{vary}_i\, s =_{\alpha\beta\eta} \mathrm{vary}_i\, t$ 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 $=_{\alpha\beta\eta}$.

$\square$

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, $n = |\,\mathrm{opaq}\, g|$, 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. $\square$

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.

Theorem 1. *Let* $s, t \in \mathcal{I}$ *with* $\mathrm{opaq}\, s =_{\alpha\beta\eta} \mathrm{opaq}\, t$. *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 $s \simeq t$. $\square$

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

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.

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. Now let us identify idiomatic terms $\mathcal{I}$ with HOL terms. While the implementation of the normal form conversion does not create an internal representation in the style of $\mathcal{I}$, this discussion links the abstract properties described earlier to the ML algorithm. Assume that the applicative functor is fixed, i.e., there is a type scheme $f[{'a}]$ and corresponding terms pure, $(\diamond)$. A type scheme is simply a normal HOL type 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 Pure.[4] The lifted type of α is then obtained by substitution of ${'a}$ by α in $f[{'a}]$. The mapping $[\![\_]\!]$ from idiomatic terms to HOL terms can now be stated as

$$[\![\mathsf{term}\, x]\!] = x, \quad [\![\mathsf{pure}\, x]\!] = \mathit{pure}\, x, \quad [\![f \text{ `}\mathsf{ap}\text{` } x]\!] = [\![f]\!] \diamond [\![x]\!];$$

with a side condition for well-typedness. For the inverse mapping, we demand that pure- and ‘ap‘-nodes are created eagerly. Otherwise, it would suffice to map every term x to $\mathsf{term}\, x$, for example. Another problem is identifying terms of the form $\mathit{pure}\,\_$ 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 $\mathit{pure}\, v$ (or $v_1 \diamond v_2$) with variables v (v_1, v_2) as the pattern.

Figure 1 has been augmented with the corresponding HOL equations. Rule `B_pure` is easily proven by rewriting with `B_intro` and `merge`. The other rules

[4] We allow additional type variables in functor signatures, which therefore represent families of functors.

are the applicative laws and thus are available from the registration infrastructure. Applying a transformation to a term means instantiating the rule such that the left hand side is equal to that term. This is done by the ML function `Conv.rewr_conv`, which uses matching internally.

$\boxed{\text{To do.}}$ current implementation with conversion combinators relies on term structure

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

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

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: $\mathbf{C} f x y = f y x$. 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

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

Table 2: Useful combinators.

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 $\mathbf{C}$ 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 [11]. 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 [1, 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. [11, 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 6. 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 [12] 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. 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

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

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 $t \text{ `}\mathsf{ap}\text{` } \mathsf{var}\,x \text{ `}\mathsf{ap}\text{` } \mathsf{var}\,x$, 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.

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

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

As with ordinary terms, we demand a soundness property for idiomatic bracket abstraction, namely that $[x]'t' \text{ `}\mathsf{ap}\text{` } \mathsf{var}\,x \simeq_C t'$ holds true. The definitions for the additional combinators C get lifted to $\mathsf{pure}\,\mathbf{I} \text{ `}\mathsf{ap}\text{` } x \simeq_C x$ 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 $\downarrow [x]'t' = [x] \downarrow t'$ and $[x]'t' \text{ `}\mathsf{ap}\text{` } \mathsf{var}\,x \simeq_{C(A')} t'$, 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 $[x]\bigdownarrow t'$. 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 $x \notin \text{var}(u)$. The induction hypothesis is

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

Then we have

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

and

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

□

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 $\text{var}(t')$. If $f = [x_1] \cdots [x_n] \big\downarrow t'$ is defined for A and no variable in $\text{var}(t')$ is free in f, 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) $\mathsf{pure}\, f \text{ `ap` } \mathsf{var}\, x_1 \text{ `ap` } \cdots \text{ `ap` } \mathsf{var}\, x_n$ is a canonical form of t';

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

Proof. a) is due to $\big\downarrow([x_1]' \cdots [x_n]'t') = f$ (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 $\mathsf{pure}\, f \text{ `ap` } \mathsf{var}\, x_1 \text{ `ap` } \cdots \text{ `ap` } \mathsf{var}\, x_n \simeq_{C(A)} t'$ by induction, making repeated use of Lemma 7.

d) $\boxed{\text{To do.}}$

□

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

Table 3: Substructures of BCKW.

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 **SK** = **BCKW** and **BICS** = **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. 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.) The rules have been chosen in the following way: 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.

$\boxed{\text{To do.}}$

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

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

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