Applicative Functors in Isabelle/HOL: Notes

Joshua Schneider

July 8, 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:

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

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. In certain cases stronger conditions are required. $\boxed{\text{To do.}}$

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 $\mathit{lhs} = \mathit{rhs}$, 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 $\mathit{pure}_f\ \_$ or $\_ \diamond_f \_$, a new, locally quantified variable is introduced. 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.

Example 2. Continuing with the set idiom from example 1, assume that the user wants to prove an instantiation of the associativity law for $\oplus$,

$$(X \oplus Y) \oplus Fa = X \oplus (Y \oplus Fa),$$

as part of a larger proof, where X, Y and F are fixed variables, and a is a constant. The system has been informed of *set* and $\oplus$. After applying the proof method, the new proof obligation reads

$$\bigwedge xyu.\ (x + y) + u = x + (y + u). \qquad \blacktriangle$$

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. All following steps depend on which additional properties of f have been provided.

If there are none, we normalize both sides of the equation. 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. As it turns out, we can compute this normal form for arbitrary terms. This is convenient because impure parts are handled implicitly. The details of the normalization algorithm are described in 2. The normalized equation is

$$\mathit{pure}\, g \diamond t_1 \diamond \cdots \diamond t_m = \mathit{pure}\, h \diamond s_1 \diamond \cdots \diamond s_n,$$

where g and h are new terms, and $\vec{t}$ and $\vec{s}$ are the impure subterms of the original equation. 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

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

The normal form has the interesting property that this is exactly the generalized base form of the original equation.

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

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 $\implies$ 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 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. Hinze [1] gave an explicit construction of this normal form for the monoidal variant of applicative functors. The normal form is useful for our purpose because the pure part reflects the term that was lifted. Its construction can be performed using only the applicative laws, so this is the most general approach regarding instances (but not regarding lifted equalities). In the following, we define lifting and normalization formally, based on a syntactic representation of idiomatic terms. Then we describe the implementation of the normalization procedure in Isabelle/ML.

2.1 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 a 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 2, 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. However, types are ignored for simplicity.

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. The set of untyped lambda terms is defined as

$$\mathcal{T} ::= \mathcal{V} \mid (\mathcal{T}\,\mathcal{T}) \mid \boldsymbol{\lambda}\mathcal{V}.\,\mathcal{T} \tag{2.1}$$

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. ▲

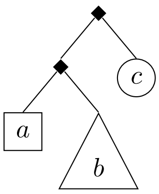

Figure 1: $(\mathsf{pure}\, a$ ‘ap‘ $b)$ ‘ap‘ $\mathsf{term}\, c$ as a tree.

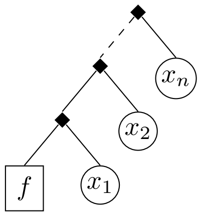

Figure 2: A term in normal form.

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

$$\mathcal{I} ::= \mathsf{term}\, \mathcal{T} \mid \mathsf{pure}\, \mathcal{T} \mid \mathcal{I} \text{ ‘}\mathsf{ap}\text{‘ } \mathcal{I}. \tag{2.2}$$

‘ap‘ associates to the left. An $\mathcal{I}$-congruence is an equivalence relation closed under ‘ap‘. The congruence $\simeq$ is induced by the rules

$$x \simeq \mathsf{pure}\, (\boldsymbol{\lambda} x.\, x) \text{ ‘}\mathsf{ap}\text{‘ } x \tag{2.3}$$

$$g \text{ ‘}\mathsf{ap}\text{‘ } (f \text{ ‘}\mathsf{ap}\text{‘ } x) \simeq \mathsf{pure}\, \mathbf{B} \text{ ‘}\mathsf{ap}\text{‘ } g \text{ ‘}\mathsf{ap}\text{‘ } f \text{ ‘}\mathsf{ap}\text{‘ } x \tag{2.4}$$

$$\mathsf{pure}\, f \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{pure}\, x \simeq \mathsf{pure}(f\ x) \tag{2.5}$$

$$f \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{pure}\, x \simeq \mathsf{pure}\, ((\boldsymbol{\lambda} x.\, \boldsymbol{\lambda} f.\, f\ x)\ x) \text{ ‘}\mathsf{ap}\text{‘ } f \tag{2.6}$$

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

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

$$\mathsf{pure}\, g \text{ ‘}\mathsf{ap}\text{‘ } (f \text{ ‘}\mathsf{ap}\text{‘ } x) \simeq \mathsf{pure}\, (\mathbf{B}\ g) \text{ ‘}\mathsf{ap}\text{‘ } f \text{ ‘}\mathsf{ap}\text{‘ } x \tag{2.7}$$

from (2.4), where g is instantied with $\mathsf{pure}\, g$, and a substitution along (2.5) 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 normal 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 (Normal form)**.** The set $\mathcal{N} \subset \mathcal{I}$ of idiomatic terms in normal form is defined inductively as

$$\mathsf{pure}\, x \in \mathcal{N}, \tag{2.8}$$

$$t \in \mathcal{N} \implies t \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{term}\, s \in \mathcal{N}. \tag{2.9}$$

▲

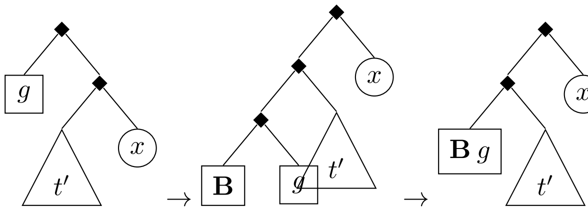

Figure 3: The "pure-rotate" step.

It is not entirely obvious how the normal form can be derived from equations (2.3)–(2.6). Rewriting blindly with these is prone to infinite recursion. Therefore we need a more controlled algorithm. 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 (2.3), we have $t \simeq \mathsf{pure}\,(\boldsymbol{\lambda} x.\, x)$ ‘ap‘ t, which is in normal form. But in the case of an $t = u$ ‘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 $u', v' \in \mathcal{N}$.

Now let us assume that u' is just $\mathsf{pure}\, g$. If v' is also a pure term, they can be combined along (2.5). 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 is separated at the outer-most level. This is the same equivalence as stated in (2.7). 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

$$\text{pure-nf}(\mathsf{pure}\, g \text{ ‘}\mathsf{ap}\text{‘ } (f \text{ ‘}\mathsf{ap}\text{‘ } x)) = \text{pure-nf}\,(\mathsf{pure}\,(\mathbf{B}g) \text{ ‘}\mathsf{ap}\text{‘ } f) \text{ ‘}\mathsf{ap}\text{‘ } x \tag{2.10}$$

$$\text{pure-nf}(\mathsf{pure}\, f \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{pure}\, x) = \mathsf{pure}\,(f x) \tag{2.11}$$

Lemma 1. *For all $g \in \mathcal{T}$ and $t \in \mathcal{N}$,* pure-nf($\mathsf{pure}\, g$ ‘ap‘ t) *is well-defined, and*[2] pure-nf($\mathsf{pure}\, g$ ‘ap‘ t) $\in \mathcal{N} \simeq \mathsf{pure}\, g$ ‘ap‘ t.

Proof. We prove all claims simultaneously by induction on $t \in \mathcal{N}$, 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

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

The right-hand side is an element of $\mathcal{N}$, and equivalence follows from (2.5).

Case 2. Assume $t = t'$ ‘ap‘ $\mathsf{term}\, x$ for some $t' \in \mathcal{N}$, $x \in \mathcal{T}$, and 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

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

is well-defined. $\boxed{\text{To do.}}$

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

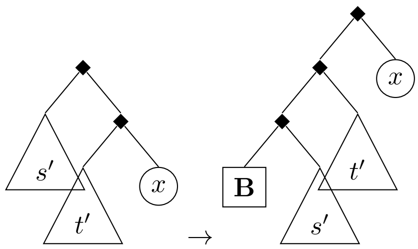

Figure 4: The “rotate” step.

$\square$

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

$$\text{nf-pure}(f \text{ ‘}\mathsf{ap}\text{‘ } \mathsf{pure}\, x) = \text{pure-nf}\,(\mathsf{pure}\,((\boldsymbol{\lambda} x.\, \boldsymbol{\lambda} f.\, f\, x)\, x) \text{ ‘}\mathsf{ap}\text{‘ } f) \tag{2.12}$$

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

Proof. Follows from Lemma 1 and (2.6). $\square$

Finally, we look at general u', v'. A term rotation separates a single opaque term, see figure 4. Before recursion, we must normalize the subterm $\mathsf{pure}\,\mathbf{B}$‘$\mathsf{ap}$‘$s'$. But we already know how to do this: by pure-nf. The base case is reached when v' is a single pure term, and that's the domain of nf-pure. The corresponding definition is nf-nf,

$$\text{nf-nf}(g \text{ ‘}\mathsf{ap}\text{‘ } (f \text{ ‘}\mathsf{ap}\text{‘ } x)) = \text{nf-nf}\,(\text{pure-nf}\,(\mathsf{pure}\,\mathbf{B} \text{ ‘}\mathsf{ap}\text{‘ } g) \text{ ‘}\mathsf{ap}\text{‘ } f) \text{ ‘}\mathsf{ap}\text{‘ } x \tag{2.13}$$

$$\text{nf-nf}(t) = \text{nf-pure}(t) \quad \text{(otherwise)} \tag{2.14}$$

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

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

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 $s \in \mathcal{N}$, Lemma 2 applies, and the claim follows.

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

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

We have pure-nf($\mathsf{pure}\,\mathbf{B}$ ‘ap‘ s) $\in \mathcal{N}$ from Lemma 1. Thus, we instantiate the induction hypothesis and the claim follows. $\boxed{\text{To do.}}$

Algorithm 1 Normalization of idiomatic terms.

$$\text{pure-nf}\, t = \begin{cases} \text{pure-nf}\,(\mathsf{pure}\,(\mathbf{B} g)\ \text{‘}\mathsf{ap}\text{‘}\ f)\ \text{‘}\mathsf{ap}\text{‘}\ x & \text{if } t = \mathsf{pure}\, g\ \text{‘}\mathsf{ap}\text{‘}\ (f\ \text{‘}\mathsf{ap}\text{‘}\ x), \\ \mathsf{pure}\,(f x) & \text{if } t = \mathsf{pure}\, f\ \text{‘}\mathsf{ap}\text{‘}\ \mathsf{pure}\, x, \\ t & \text{otherwise.} \end{cases}$$

$$\text{nf-pure}\, t = \begin{cases} \text{pure-nf}\,(\mathsf{pure}\,((\boldsymbol{\lambda} x.\ \boldsymbol{\lambda} f.\ f x)\ x)\ \text{‘}\mathsf{ap}\text{‘}\ f) & \text{if } t = f\ \text{‘}\mathsf{ap}\text{‘}\ \mathsf{pure}\, x, \\ t & \text{otherwise.} \end{cases}$$

$$\text{nf-nf}\, t = \begin{cases} \text{nf-nf}\,(\text{pure-nf}\,(\mathsf{pure}\,\mathbf{B}\ \text{‘}\mathsf{ap}\text{‘}\ g)\ \text{‘}\mathsf{ap}\text{‘}\ f)\ \text{‘}\mathsf{ap}\text{‘}\ x & \text{if } t = g\ \text{‘}\mathsf{ap}\text{‘}\ (f\ \text{‘}\mathsf{ap}\text{‘}\ x), \\ \text{nf-pure}\, t & \text{otherwise.} \end{cases}$$

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

□

Algorithm 1 shows 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 $t \in \mathcal{N} \simeq t$*.*

Proof. By induction on t, Lemma 3, and equation (2.3). □

Until now, we only have considered the syntactic structure of idiomatic terms together with the artificial relation $\simeq$, which is also based on syntax. In order to define the semantics of idiomatic terms, we assume that we operate in an equational theory Ω based on $\mathcal{T}$-terms, where $=_\Omega \supseteq =_{\alpha\beta\eta}$ is an equivalence relation.

Definition 4 (Idiomatic interpretation)**.** Let $\iota = \langle p, a \rangle$ with $p, a \in \mathcal{T}$. The interpretation $[\![t]\!]_\iota$ of the idiomatic term t w.r.t. ι is defined as

$$[\![\mathsf{term}\ t]\!]_\iota = t, \tag{2.15}$$

$$[\![\mathsf{pure}\ t]\!]_\iota = p\ t, \tag{2.16}$$

$$[\![s\ \text{‘}\mathsf{ap}\text{‘}\ t]\!]_\iota = (a\ [\![s]\!]_\iota)\ [\![t]\!]_\iota. \tag{2.17}$$

ι is an idiomatic structure (in Ω) iff

$$\forall q r.\ q \simeq r \implies [\![q]\!]_\iota =_\Omega [\![r]\!]_\iota. \tag{2.18}$$

▲

Definition 5.

$$\iota_{\text{id}} = \langle \boldsymbol{\lambda} x.\ x, \boldsymbol{\lambda} f.\ \boldsymbol{\lambda} x.\ f\ x \rangle \tag{2.19}$$

▲

Lemma 5. *For all $t \in \mathcal{I}$, there is a unique term $t' \in \mathcal{N}$ such that $t \simeq t'$.*

Proof. Existence of the normal form is a corollary of Lemma 4. In order to show that it is unique, we construct a relation $R \subseteq \mathcal{I} \times \mathcal{I}$, such that

- for all idiomatic terms s and t, $s \simeq t \implies (s, t) \in R$, and
- for all terms in normal form n and n', $(n, n') \in R \implies n =_{\alpha\beta\eta} n'$.

R is defined in two steps. The first deals with the sequence of opaque terms,

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

Let $(s, t) \in R'$ if and only if $\mathrm{opaq}(s) = \mathrm{opaq}(t)$ pointwise. Now assume $(s, t) \in R'$. We can modify both terms such that all opaque terms are replaced by new pure variables, say $\mathsf{pure}\, v_1, \ldots, \mathsf{pure}\, v_n$. The mapping is the same for both terms, i.e., the variable is determined by the position in opaq(_). Let s', t' be these modified terms. Then $(s, t) \in R$ if and only if $[\![s']\!]_{\iota_{\mathrm{id}}} =_{\alpha\beta\eta} [\![t']\!]_{\iota_{\mathrm{id}}}$.

It remains to show that R satisfies both properties. $\boxed{\text{To do.}}$ □

Lemma 6. $\boxed{\textit{To do.}}$ *Every encoding of an applicative functor in Ω is an idiomatic structure. Especially ι_{id} is an idiomatic structure.*

Definition 6 (Lifted terms)**.** $q \in \mathcal{I}$ is a lifting of $t \in \mathcal{T}$ if $[\![q]\!]_{\iota_{\mathrm{id}}} =_\Omega t$. ▲

Lemma 7. *Let q be a lifting of t. The normal form q' of q can be written as*

$$q' = (\cdots((\mathsf{pure}\, t' \text{ `}\mathsf{ap}\text{` } \mathsf{term}\, a_1) \text{ `}\mathsf{ap}\text{` } \mathsf{term}\, a_2) \cdots) \text{ `}\mathsf{ap}\text{` } \mathsf{term}\, a_n.$$

Then $t'\, \vec{a} =_\Omega t$.

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

References

[1] Ralf Hinze. "Lifting Operators and Laws". 2010. URL: `http://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf` (visited on 2015-06-06).

[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.