Permutations etc.

Jacques Carette Amr Sabry

June 5, 2015

Abstract

...

1 Introduction

Main points:

- We want a language for writing and reasoning about equivalences à la HoTT. That would be a reversible language that comes with its own executable optimizer.
- Doing this for a λ-calculus based language requires finding an appropriate semantics for equivalences that gives a computational interpretation to univalence; this is still subject of research; our approach is to start with finite types and leave higher-order functions for now. More about this later (talk then about negative and fractional types as a possibility for extending the work to accommodate some form of higher-order functions). More motivation about our approach: starting with all functions makes it very difficult to identify equivalences (must use function extensionality); instead we build equivalences inductively. This is relatively easy for finite types but will get much more interesting when we go to negative and fractional types.
- Equivalences between finite types can be expressed in many ways; it is conjectured (Baez) that the canonical way is permutations on finite sets. However, it is important to note that we are not talking about just the set (or setoid) of permutations, but with the rig of permutations, with disjoint union as $+$ and tensor product as $*$.
- Even though operations (such as tensor product, and even reversal) of permutations are operationally quite complex, we can show that they originate (entirely) from simpler operations on natural numbers and on types.
- More abstractly these equivalences can be expressed using *symmetric rig categories*. The beauty of going to the categorial setting is that the principles for reasoning about permutations are essentially the coherence conditions for the categories. We quote:

  > What Mac Lane does can be described in logical terms in the following manner. On the one hand, he has an axiomatization, and, on the other hand, he has a model category where arrows are permutations; then he shows that his axiomatization is complete with respect to this model. It is no wonder that his coherence problem reduces to the completeness problem for the usual axiomatization of symmetric groups. (p.3 of `http://www.mi.sanu.ac.rs/~kosta/coh.pdf`)

- Putting the observations above together, we can develop a programming language with the following characteristics:
  - The set of types consists of the conventional finite types: empty, unit, sums, and products

  - The set of terms consists of a rich enough set of combinators that can denote every equivalence between the types
  - More interestingly, we have a higher-level of combinators that manipulate the first level of combinators to provide a sound and complete calculus for computing and reasoning about equivalences of equivalences.
  - The language has a simple, intuitive, and almost conventional operational semantics
  - Denotationally, the language can be interpreted in any *symmetric rig category*. One possibility is the canonical category of finite sets and permutations; another would be the Agda universe Set.

- In the setting describe above, we can *prove* a theorem that intuitively corresponds to the statement of *univalence* in our setting. The theorem states that the set of equivalences between equivalences is equivalent to identities of permutations.

- Pi0: we have an operational semantics that maps each combinator to a function; this is intuitive but bad for reasoning as it would require reasoning about extensional equivalence of functions; it is also bad because we completely lose the fact that we are starting with a reversible language; we have an alternative semantics that maps each combinator to a permutation. Semantically permutations (with a rich algebra) can be modeled using a *typed* semiring; we have several instances of these categories that show that a few simple properties of natural numbers lift to properties of Fin then vectors then permutations.

- Pi1: we have a nice semantics which maps each combinator to a permutation but we don't have a way yet to reason about which permutations are equivalent to each other? We would like to answer this question without going to extensional equality of functions. It turns out that this question has essentially been answered by category theorists and is encoded in the coherence conditions for monoidal categories (precisely symmetric rig categories). These conditions classify what is going on. We can turn these coherence conditions into a typed operational semantics for program transformations

- Pi2: we can start seeing which program transformations are equivalent. This requires a generalization of rig categories.

- Possible application: reversible circuits + optimizations

- Now how do you fit Thm 2 into that story and is it possible to do it in a way that makes the structure of Pi0 and Pi1 part of the same general pattern? At level 0, thm 2 says that, given a choice of enumeration, permutations are *initial* and *complete*.

2 Finite Types

Our first step is to recall a few properties of *finite types*. Syntactically, these types are constructed by the following grammar:

$$(\textit{Types}) \quad \tau \quad ::= \quad 0 \mid 1 \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2$$

The set of types τ includes the empty type 0, the unit type 1, and conventional sum and product types.

Semantically, each such type denotes a *finite set*. The only equivalence on the elements of these sets is the *identity* relation which we write as $=$. In other words if $x, y : A$ and $x = y$ then it must be the case that x and y are identical. For two finite sets A and B, two maps $f, g : A \to B$ are *extensionally equivalent*, $f \sim g$, if for all $x : A$ we have that $f(x) = g(x)$.

$$
\begin{array}{rrcll}
identl_+ : & 0+\tau & \leftrightarrow & \tau & : identr_+ \\
swap_+ : & \tau_1+\tau_2 & \leftrightarrow & \tau_2+\tau_1 & : swap_+ \\
assocl_+ : & \tau_1+(\tau_2+\tau_3) & \leftrightarrow & (\tau_1+\tau_2)+\tau_3 & : assocr_+ \\
identl_* : & 1*\tau & \leftrightarrow & \tau & : identr_* \\
swap_* : & \tau_1*\tau_2 & \leftrightarrow & \tau_2*\tau_1 & : swap_* \\
assocl_* : & \tau_1*(\tau_2*\tau_3) & \leftrightarrow & (\tau_1*\tau_2)*\tau_3 & : assocr_* \\
dist_0 : & 0*\tau & \leftrightarrow & 0 & : factor_0 \\
dist : & (\tau_1+\tau_2)*\tau_3 & \leftrightarrow & (\tau_1*\tau_3)+(\tau_2*\tau_3) & : factor
\end{array}
$$

$$
\frac{}{\vdash id : \tau \leftrightarrow \tau}
\qquad
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_2 \leftrightarrow \tau_3}{\vdash c_1 \fatsemi c_2 : \tau_1 \leftrightarrow \tau_3}
$$

$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1+\tau_3 \leftrightarrow \tau_2+\tau_4}
\qquad
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \otimes c_2 : \tau_1*\tau_3 \leftrightarrow \tau_2*\tau_4}
$$

Figure 1: Π-combinators [**?**]

3 Equivalences between Finite Types

In the general context of HoTT, there are several *equivalent* definitions of *equivalences* between types. The most natural definition for our purposes is the notion of "bi-invertible maps." A map $f : A \to B$ is *bi-invertible* if it has both a left inverse and a right inverse, i.e., if there exist maps $g, h : B \to A$ such that $g \circ f \sim \mathrm{id}_A$ and $f \circ h \sim \mathrm{id}_B$. A semantic equivalence between sets A and B is completely specified by a bi-invertible map $f : A \to B$. Our aim is to define a syntactic, *inductive*, characterization of this equivalence.

Intuitively, we expect two finite sets to be equivalent if there is a *permutation* between them and hence our aim is essentially to produce a syntactic characterization of permutations on finite types that is sound and complete with respect to the semantic equivalence on finite sets above. We will immediately present a syntactic definition of permutations on finite types and then proceed to prove that it exactly corresponds to semantic equivalence.

3.1 Π-Combinators

In previous work, we defined a reversible language Π whose only computations are witnesses to isomorphisms $\tau_1 \leftrightarrow \tau_2$ between finite types [**?**]. Syntactically, these computations consist of base combinators (on the left side of Fig. **??**) and compositions (on the right side of the same figure). Each line of the figure on the left introduces a pair of dual constants[1] that witness the type isomorphism in the middle. This set of isomorphisms is known to be complete [**?**, **?**] and the language is universal for hardware combinational circuits [**?**].[2]

The technical goal now is to prove that every Π-combinator corresponds to a semantic equivalence and vice-versa that every semantic equivalence can be expressed as a Π-combinator. Furthermore, going back and forth between these two spaces produces "equal" objects, e.g., mapping a Π-combinator to an equivalence and back produces an "equal" Π-combinator. To express the main theorem, we therefore need to specify what it means for two Π-combinators to be the "same" and similary what it means for two equivalences to be the "same". The formal statement we prove (to be dissected and explained in detail in the remainder of the section) is:

```
thm : ∀ {τ₁ τ₂ : U} → (≃S-Setoid ⟦ τ₁ ⟧ ⟦ τ₂ ⟧) ≃S (Π-Rewriting τ₁ τ₂)
```

For the main theorem we are given two types A and B and an *enumeration* Enum that establishes an equivalence between each type and Fin n. These equivalences establish three things: (i) that each type is indeed a finite type, (ii) that the two types have the same number of elements, and (ii) fix an ordering on the elements of each type.

[1] where $swap_+$ and $swap_*$ are self-dual.

[2] If recursive types and a trace operator are added, the language becomes Turing complete [**?**, **?**]. We will not be concerned with this extension in the main body of this paper but it will be briefly discussed in the conclusion.