Homomorphic type casting of algebraic data types through partial-α-equivalence

Iain Moncrief

moncrief@oregonstate.edu

January 2021

1 Grammar

1.1 λ-Calculus

expression ::=	variable	*identifier*
|	expression expression	*application*
|	λ variable . expression	*abstraction*
|	(expression)	*atom*

1.2 Types

type ::=	*	*abstract type*
|	typename	*defined type*
|	type $\mapsto$ type	*function type*
|	(type)	*atom*

2 Encoding Rules

2.1 Definitions

Expression type is the grammar rule to which an expression in normal form conforms to. This is denoted with $(*e)$ where e is a λ-expression.

Terms of an expression are the most basic components of an expression, defined in the grammar.

Expression is a composition of terms that can be referred to.

2.2 Axioms

Axiom 1. *For all λ-encodings of type A, any pair $(g, h),\ g, h \in A$ must not differ by a top-level-abstraction.*

$$((\lambda m.t_1) \wedge (\lambda m.\lambda n.t_2) \implies (\lambda m.t_1) \notin A \vee (\lambda m.t_2) \notin A)$$

Note: Still not sure how to do formal proofs with λ-calculus, so I needed to add this to make the theorems work. I still have yet to find a counter example to theorem 1. This axiom may be unnecessary.

2.3 Shared abstraction structure of type elements

In λ-calculi, data types and respective operations can be encoded as abstractions.

Theorem 1. *The encodings of a type must share the same abstraction argument structure and types of each argument.*

Proof. The lambda abstractions

$$(\lambda a.\lambda b.\lambda c.t_1), \quad (\lambda x.\lambda y.\lambda z.t_2)$$

may encode different values of the same type, except when t_1 or t_2 is an abstraction and the other is not. Suppose t_2 is an abstraction

$$t_2 := (\lambda w.t_2')$$

the right-hand-side encoding expands to

$$(\lambda a.\lambda b.\lambda c.t_1), \quad (\lambda x.\lambda y.\lambda z.\lambda w.t_2')$$

unless t_1 is also an abstraction, both encodings have unlike argument structures. Given the Rule of Typed λ-cacluli, these terms cannot encode elements of the same type. $\square$

2.4 Unlike abstraction body of type elements

Theorem 2. *The encodings of two different elements of the same type cannot share the same abstraction body.*

Proof. The type A where a_1 and a_2 are two different values of type A.

$$a_1 \neq a_2$$
$$a_1, a_2 \in A$$

From Theorem 1, all encodings of a type share the same abstraction argument structure, a_1 and a_2 will have the same abstraction arguments.

$$(\lambda \ldots .t) \in A$$

where t is a non-abstraction term. Let P represent the bound variable list of the λ-abstraction for each encoding of type A

$$P := \lambda a.\lambda b.\lambda c.\lambda \ldots$$
$$(P.t) \in A$$

With the abstraction argument pattern P, we can expand the encodings

$$a_1 = (P.t_1)$$
$$a_2 = (P.t_2)$$

where t_1 and t_2 are abstraction bodies, therefore

$$a_1 \neq a_2$$

Finally, we take a_1 and a_2 to have the same abstraction body

$$b := \text{an arbitrary abstraction body}$$
$$(P.t_1) \neq (P.t_2)$$
$$(P.b) = (P.b)$$
$$a_1 = a_2$$

forming ambiguity between both values of type A, as there are no differences in the encodings of a_1 and a_2.

□

2.5 Example

Two encodings of the same type must have equivalent parameters, and differing bodies.

`tru`	`fls`
$\lambda x.\lambda y.x$	$\lambda x.\lambda y.y$

`Bool`

$\lambda x.\lambda y.t$

where t is not an abstraction

3 Isomorphisms

These notion of structure within λ-terms can be generalized by introducing the concept of λ-isomorphisms. To preserve the meaning of an isomorphism from other areas of Mathematics, a λ-isomorphism should refer to a bijective λ-*homomorphism.*

Formally, a λ-homomorphism of λ-expressions G, H, is a function $f : G \to H$ that maps a term $t \in G$ to a term $f(t) \in H$, preserving the expression type of t. Therefore, we can write

$$\forall \tau \left((\tau \in G \iff f(\tau) \in H) \wedge (*\tau) = (*f(\tau)) \right)$$

implies that $f : G \to H$ is a homomorphism. For the case where τ is an abstraction $(*\tau) = \text{Abs}$, it would not be enough to say $\tau,\ f(\tau)$ are homomorphic as long as they are both abstractions. To accommodate this, the requirements must be adjusted for a homomorphism between two abstractions, such that the body of $\tau,\ f(\tau)$ share the same expression type.

The two terms

$$\begin{array}{cc} \lambda a.d & \lambda b.\lambda c.e \\ G & H \end{array}$$

are homomorphic because there exists a homomorphism $f : G \to H$. Therefore f can be shown to exist because it can map a term $t \in G$ to a term $f(t) \in H$. Since we have

$$a, d \in G,\ b, c, e \in H$$

we can show the function f, applied to it's input

$$\begin{aligned} f(a) &= c \\ f(d) &= e \end{aligned}$$

where on the right, is the image f. It is important to note that for the case of the terms a, c which are abstractions

$$f(a) = c \iff ((*a) = (*c) \wedge (*d) = (*e))$$

due to the rule that the expression type of the body of an abstraction must be preserved. In the general case

$$f(\tau) = \sigma \implies (*\tau) = (*\sigma)$$

and is obvious that

$$f(\tau) = \sigma \iff (*f(\tau)) = (*\sigma)$$

.

From the definition of a λ-homomorphism, a λ-isomorphism can be defined. An isomorphism usually is a bijective homomorphism, so it will be just that. Formally, a λ-isomorphism is a λ-homomorphism $f : G \to H$ that is one-to-one and onto (bijective).

The two terms

$$\begin{array}{cc} \lambda x.\lambda y.x(x\ y) & \lambda g.\lambda h.g(g\ h) \\ I & J \end{array}$$

are isomorphic because there is a bijective homomorphism between them. Two terms can be shown to be isomorphic if one can use α-renaming to either term to arrive at the other (α-equivalence). This is possible because renaming identifiers completely preserves all other structure (assuming there are no name collisions).

Continuing to inherit the properties of isomorphisms in mathematics, the expressions T_1, T_2 are said to be isomorphic with the notation

$$T_1 \cong T_2$$

While it is true that two terms are λ-isomorphic if and only if they are α-equivalent, the distinction is necessary as it allows the concept of λ-homomorphisms to be defined more naturally. Expressions that are λ-homomorphic can be considered as partially-α-equivalent.

4 Isomorphic type casting

Type casting is a common topic in computer science, referring to the action of converting a value from one type to another type. Without typing rules defined in addition to the basic rules of the λ-calculus, types and their *behaviors* can be encoded within abstractions along with a specification for their operation. The `if, then, else` rules can be encoded using the boolean encodings defined above, and the natural number can be represented through Church encodings ($\lambda f.\lambda x.f^n \circ x$).

Some languages such as Rust, Haskell, Swift, and C++, have type systems that will store additional information about a value namely, their type. These languages require that types be converted according to a specific procedure. Other languages such as C and BF-lang variations allow type casting without following a conversion rule.

This C program shows how values of different types can be represented through the same information.

```
int main(void) {
    typedef unsigned char byte;
    struct { byte a; byte b; byte c; byte d; } value;
    value.a = 11; value.b = 22; value.c = 33; value.d = 44;

    byte* bytes; bytes = (byte*)&value;

    printf("a = %i \n", value.a); printf("b = %i \n", value.b);
    printf("c = %i \n", value.c); printf("d = %i \n", value.d);

    for (int i = 0; i < 4; ++i) { printf("bytes[%i] = %i \n", i, bytes[i]); }
    return 0;
}
```

In reality, `value` is just a name for a section of 4 bytes. The operation rules are determined by the types of the *things* that reference it. Furthermore, the data caries no information about itself, only the encoding of the value for a certain type.

This is also the case in untyped λ-calculi. Values of a type are encoded through application and abstraction terms, and define the operation rules of the values. Since there is no other information relating a value to its type, values of different types may have the same encoding for a given value.

The most weakly typed λ-calculus should still implement a form of type casting validation. Meaning that the type of a value can be casted to another value if and only if the values of each type are isomorphic.

Formally, isomorphic type casting for a value $a \in A$ of type A and $b \in B$ of type B, requires $a \cong b$ in order for

$$a \xrightarrow{\circ} c,\ c = b,\ c \in B,\ c \xrightarrow{\circ} d,\ d \in A,\ a = d$$

where $\tau \xrightarrow{\circ} \sigma$ denotes the true/false value of the value τ can to be casted to the value σ. In other words, $\tau \xrightarrow{\circ} \sigma$ means "a value of τ may be casted to a value of σ".

5 Homomorphic type checking of algebraic data types

Algebraic data types (ADT) or *parameterized* data types are types that can generalize the behavior of a type. Many object oriented programming languages refer to this as *templates* or *generic types*.

The `List` data structure in Haskell is implemented as an algebraic type

```
data List a = Nil | Cons a (List a)
```

where `a` can refer to any type.

The same may be done using a weakly typed λ-calculus implementation

$$(\lambda f.f\ (elem : A)\ (next : \text{Nil}|\text{List}(A)))\ : \text{List}(A)$$

A full binary tree can be represented using the encoding

$$(\lambda f.f\ (left : A|\text{Tree}(A))\ (right : A|\text{Tree}(A)))\ : \text{Tree}(A)$$

where the bound variable f defines the available behavior of the type, and allows operations to be applied to either children of the node. The Haskell implementation would be

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Moreover, a tree traversal can be represented as a list of `fst` or `scd` pair (not necessarily the pair encodings for λ-calculus) accesses

$$p = (\lambda f.f\ (\texttt{fst})\ (\lambda g.g\ (\texttt{scd})\ (\lambda h.h\ (\texttt{fst})(\ldots))))$$

where p is an access sequence of a tree, to a specific node, taking the route of left, right, left (first, second, first) to get to the node. To ensure that only valid behaviors are applied to a tree, the bound f could be given a type restriction

$$(\lambda f : \text{Ops}(\text{Tree}(A))\ .f\ (left : A|\text{Tree}(A))\ (right : A|\text{Tree}(A)))\ : \text{Tree}(A)$$

where $\text{Ops}(T)$ is the union type of all operations for all values of type T. In Haskell, this could be implemented as

```
type HandSideSelector = a -> a -> a

scd :: HandSideSelector
scd _ x = x -- Also the false encoding

fst :: HandSideSelector
fst x _ = x -- Also the true encoding

data TreeAccess = List HandSideSelector

accessValue :: Tree a -> TreeAccess -> Tree a
accessValue (Leaf x) _ = x
accessValue tree (Cons _ Nil) = tree
accessValue (Node x y) (Cons choose tail) = accessValue (choose x y) tail
```

Homomorphisms can be used to validate if an encoding conforms to the structure of a certain type. Furthermore, homomorphic type checking does not require that the type *encapsulated* by an ADT need to the same, and only requires that two concrete types are instances of the same ADT.

Two λ-terms τ, σ that both encode trees of possibly different types, can be checked that they conform to the structure of the tree type by determining if their root terms are homomorphic. The terms

$$\underset{A}{(\lambda f.f\ (a : T_1)\ (b : T_1))} \qquad \underset{B}{(\lambda f.f\ (g : T_2)\ (h : T_2))}$$

encode a tree, but with differing types. Expression A has type $A\ : \text{Tree}(T_1)$, and B is of type $B : \text{Tree}(T_2)$. Therefore, $A \approxeq B$ are homomorphic.

Things that I have not yet figured out

Defining $\text{Ops}(T)$ to be the union for all operations for all values of the type T might be difficult. The type Q, where

$$n = (\lambda f.f\ (numerator : \text{Int})(denominator : \text{Int})\ : Q$$

$$(n\ \text{div}) \in \mathbb{Q}$$

and in Haskell

```
data RationalNumber = Div Integer Integer

rational :: RationalNumber -> Double
rational (Div a b) = num / den where
      num = fromIntegral a :: Double
      den = fromIntegral b :: Double
```

are data types for the rational numbers. By definition, any operation $f \in \mathrm{Ops}(T)$, for all values $v \in T$, f should be a valid operation for v. In the example of the type Q, the operation div is not valid for all $v \in Q$ where $v = (\lambda f.f\ (\mathit{numerator} : \mathrm{Int})\ 0)$.