

# Deciding Isomorphisms of Simple Types in Polynomial Time

Jeffrey Considine  
jconsidi@cs.bu.edu

April 2, 2000

## Abstract

The isomorphisms holding in all models of the simply typed lambda calculus with surjective and terminal objects are well studied - these models are exactly the Cartesian closed categories. Isomorphism of two simple types in such a model is decidable by reduction to a normal form and comparison under a finite number of permutations (Bruce, Di Cosmo, and Longo 1992). Unfortunately, these normal forms may be exponentially larger than the original types so this construction decides isomorphism in exponential time. We show how using space-sharing/hash-consing techniques and memoization can be used to decide isomorphism in practical polynomial time (low degree, small hidden constant).

Other researchers have investigated simple type isomorphism in relation to, among other potential applications, type-based retrieval of software modules from libraries and automatic generation of bridge code for multi-language systems. Our result makes such potential applications practically feasible.

## 1 Introduction

Isomorphisms in various models extending simply typed lambda calculus have received much attention recently. Applications such as use of types as search keys in program libraries give these theories practical applications [12]. We examine isomorphisms holding in all models of the typed lambda calculus with surjective pairing and terminal types.

The seven axioms we will consider were shown to be complete with respect to isomorphism in the Cartesian closed categories as far back as 1985 [5]. This result was also already known in Russia in 1983 [14]. It is also known that models of the typed lambda calculus with surjective pairing and terminal types are exactly the Cartesian closed categories [4].

In [12], it was proposed that type isomorphisms in these systems should be studied to facilitate search of function libraries. The popular example (originating in [13]) shows how numerous the names and prototypes of functions “folding” a list may be. In response to this dilemma, isomorphisms are suggested as a means of identifying compatible functions and generating appropriate bridge code.

A reduction system to allow type comparisons using normal forms was initially given in [13]. A slightly different system was given in [4] which “curried” types instead of “uncurrying” them. These construction show the decidability of this system but involve an exponential blowup of the types in question. We further decompose the reduction system of [4] while preserving its confluence and strong normalization properties. By specifying a reduction order, we are able to place a polynomial bound on the number of types created during reduction allowing us to use hash-consing and memoization techniques to decide isomorphism in polynomial time.

## 2 Previous Work

### 2.1 Previous Work on Isomorphisms of Simple Types

**Definition 2.1.1** ( $Th^0$ ) We define  $Th^0$  to be the theory of equality with no additional axioms. By the theory of equality, we mean the deductive closure of the axiom schema  $A = A$  (reflexivity) under the following inference rules:

1.  $\frac{A = B, B = C}{A = C}$  (transitivity)
2.  $\frac{A = B}{B = A}$  (symmetry)
3.  $\frac{A = B, C = D}{A \rightarrow C = B \rightarrow D}$  (congruence with respect to  $\rightarrow$ )
4.  $\frac{A = B, C = D}{A \times C = B \times D}$  (congruence with respect to  $\times$ )

We build on the work of [4] which formally defined the theory as follows.

**Definition 2.1.2** ( $Th_{\times \mathbf{T}}^1$ )  $Th_{\times \mathbf{T}}^1$  is the theory of equality plus the following axiom schemas, where  $\mathbf{T}$  is a constant symbol:

1.  $A \times B = B \times A$
2.  $A \times (B \times C) = (A \times B) \times C$
3.  $(A \times B) \rightarrow C = A \rightarrow (B \rightarrow C)$
4.  $A \rightarrow (B \times C) = (A \rightarrow B) \times (A \rightarrow C)$
5.  $A \times \mathbf{T} = A$
6.  $A \rightarrow \mathbf{T} = \mathbf{T}$
7.  $\mathbf{T} \rightarrow A = A$

The soundness and completeness of  $Th_{\times \mathbf{T}}^1$  are proven in [4]. To decide isomorphisms in  $Th_{\times \mathbf{T}}^1$ , the following reduction system is defined.

**Definition 2.1.3 (Type Reduction  $\mathcal{R}^1$ )** Let  $\mathcal{R}^1$  be the transitive and substitutive type-reduction relation generated by:

1.  $A \times (B \times C) > (A \times B) \times C$
2.  $(A \times B) \rightarrow C > A \rightarrow (B \rightarrow C)$
3.  $A \rightarrow (B \times C) > (A \rightarrow B) \times (A \rightarrow C)$
4.  $A \times \mathbf{T} > A$
5.  $\mathbf{T} \times A > A$
6.  $A \rightarrow \mathbf{T} > \mathbf{T}$
7.  $\mathbf{T} \rightarrow A > A$

**Lemma 2.1.4**  $\mathcal{R}^1$  is confluent and strongly normalizes.

**Proof:**

This is proven in [4].

**Definition 2.1.5 ( $\mathcal{R}^1$ Normal Forms)** Let  $A$  be a type. The unique normal form of  $A$  under  $\mathcal{R}^1$  is denoted  $nf(A)$ .

Additionally, [8] defines the following sub-theories of  $Th_{\times \mathbf{T}}^1$  to facilitate analysis of the reduction.

**Definition 2.1.6 ( $Th_{\times}^1$ )**  $Th_{\times}^1$  is the theory of equality plus the following axiom schemas:

1.  $A \times B = B \times A$
2.  $A \times (B \times C) = (A \times B) \times C$
3.  $(A \times B) \rightarrow C = A \rightarrow (B \rightarrow C)$
4.  $A \rightarrow (B \times C) = (A \rightarrow B) \times (A \rightarrow C)$

**Definition 2.1.7 ( $Th^1$ )**  $Th^1$  is the theory of equality plus the following axiom schemas:

1.  $A \rightarrow (B \rightarrow C) = B \rightarrow (A \rightarrow C)$

**Lemma 2.1.8** *Let  $A$  be a type. Then,  $nf(A)$  is either of the form  $\mathbf{T}$  or  $A_1 \times \dots \times A_n$  where  $A_i$  does not contain  $\times$  or  $\mathbf{T}$ .*

**Proof:**

This lemma is proven in [4].

**Lemma 2.1.9** *Let  $A$  and  $B$  be types. Then,  $Th_{\times \mathbf{T}}^1 \vdash A = B$  if and only if  $nf(A) = nf(B) = \mathbf{T}$  or  $nf(A) = A_1 \times \dots \times A_n$  and  $nf(B) = B_1 \times \dots \times B_m$ ,  $n = m$ , and there is a permutation  $\sigma$  such that  $Th^1 \vdash A_i = B_{\sigma(i)}$  for all  $i$ .*

**Proof:**

This lemma is proven in [4].

## 2.2 Previous Work on Related Theories

An example of a practical system making use of isomorphisms is the Mockingbird project [3]. Mockingbird uses a combination of heuristics and programmer hints to detect isomorphisms and generate bridge code. [2] describes an attempt at formalizing these isomorphisms,  $Th_{\times \mathbf{T}\mu}^1$ , which combine  $Th_{\times \mathbf{T}}^1$  with  $Eq_{\mu \rightarrow \times}$ , a system of recursive isomorphisms known to be decidable. Unfortunately,  $Th_{\times \mathbf{T}\mu}^1$  is inconsistent, an example proof of which is found in [11]. Defining a meaningful and consistent theory of isomorphisms including recursive types which is decidable, preferably efficiently decidable, is still an open problem.

The restriction of this problem to recursive types with associativity and commutativity is solved in quadratic time in [11]. This approach is based on building up an approximation of an equivalence relation using automata techniques. This solution does not deal with currying (Axiom 3 of Definition 2.1.2) or the distributive law (Axiom 4 of Definition 2.1.2).

## 3 Reductions

### 3.1 Overview

We reduce types to normal forms under a variation of  $\mathcal{R}^1$  that is amenable to simple analysis. Instead of using  $\mathcal{R}^1$  to reduce a type to a normal form, we break up  $\mathcal{R}^1$  into two separate type reduction systems,  $\mathcal{R}_{\mathbf{T}}^1$  and  $\mathcal{R}_{\times}^1$ . Individually,  $\mathcal{R}_{\mathbf{T}}^1$  and  $\mathcal{R}_{\times}^1$  are straightforward to analyze. In particular, it is possible to give a polynomial bound on the types constructed by  $\mathcal{R}_{\times}^1$  using space sharing techniques. Previously, the reduction rule corresponding to the distributive law (reduction rule 3 in  $\mathcal{R}^1$ ) allowed exponential growth.

Once types have been reduced to the normal form of  $\mathcal{R}^1$ , we further reduce them to a variation of the normal form maintaining a sort property. This reduction is also possible in polynomial time using memoization. Once in this sorted normal form, type isomorphism is simply equality.

Trivial proofs, such as the equivalence of a type and its normal form, will be left out. Missing proofs either can be sketched in one line or proven through induction over the structure of the types in question.

### 3.2 Reduction to $Th_{\times}^1$

**Definition 3.2.1 (Type Reduction  $\mathcal{R}_{\mathbf{T}}^1$ )** Let  $\mathcal{R}_{\mathbf{T}}^1$  be the transitive and substitutive type-reduction relation generated by:

1.  $A \times \mathbf{T} > A$
2.  $\mathbf{T} \times A > A$
3.  $A \rightarrow \mathbf{T} > \mathbf{T}$
4.  $\mathbf{T} \rightarrow A > A$

Note that the rules generating  $\mathcal{R}_{\mathbf{T}}^1$  are exactly the rules generating  $\mathcal{R}^1$  that refer to  $\mathbf{T}$ .

**Lemma 3.2.2**  $\mathcal{R}_{\mathbf{T}}^1$  is confluent and strongly normalizes.

**Proof Sketch:**

$\mathcal{R}_{\mathbf{T}}^1$  terminates since the size of the expression decreases with each reduction step. It is trivial to verify that  $\mathcal{R}_{\mathbf{T}}^1$  is locally confluent.<sup>1</sup> These properties are sufficient to prove the desired lemma [10].

**Definition 3.2.3 ( $\mathcal{R}_{\mathbf{T}}^1$  Normal Forms)** Let  $A$  be a type. The unique normal form of  $A$  under  $\mathcal{R}_{\mathbf{T}}^1$  is denoted  $nf_{\mathbf{T}}(A)$ .

**Lemma 3.2.4** Let  $A$  be a type. Then,  $Th_{\times\mathbf{T}}^1 \vdash A = nf_{\mathbf{T}}(A)$ .

**Definition 3.2.5 (Trivial Types)** Let  $A$  be a type. Then,  $A$  is a *trivial* type if and only if  $Th_{\times\mathbf{T}}^1 \vdash A = \mathbf{T}$ .

**Lemma 3.2.6** Let  $A$  be a type. Then,  $A$  is trivial if and only if  $nf_{\mathbf{T}}(A) = \mathbf{T}$ .

**Proof Sketch:**

If  $nf_{\mathbf{T}}(A) = \mathbf{T}$ , then  $Th_{\times\mathbf{T}}^1 \vdash A = \mathbf{T}$  by Lemma 3.2.4 and transitivity. Otherwise,  $Th_{\times\mathbf{T}}^1 \vdash A = \mathbf{T}$ , and all series of reductions applied to  $A$  in  $\mathcal{R}^1$  result in  $\mathbf{T}$ . If all possible reductions in  $\mathcal{R}_{\mathbf{T}}^1$  are applied first, the result is either  $\mathbf{T}$  or a term that does not contain  $\mathbf{T}$ . In the first case,  $nf_{\mathbf{T}}(A) = \mathbf{T}$ . In the second case, this term may not be reduced to  $\mathbf{T}$  in  $\mathcal{R}^1$  since no rules in  $\mathcal{R}^1$  introduce  $\mathbf{T}$ . Therefore,  $nf_{\mathbf{T}}(A) = \mathbf{T}$ .

**Lemma 3.2.7** Let  $A$  and  $B$  be non-trivial types. Then,  $Th_{\times\mathbf{T}}^1 \vdash A = B$  if and only if  $Th_{\times\mathbf{T}}^1 \vdash nf_{\mathbf{T}}(A) = nf_{\mathbf{T}}(B)$ .

**Proof:**

$nf_{\mathbf{T}}(A)$  and  $nf_{\mathbf{T}}(B)$  do not contain  $\mathbf{T}$  so this is proven in [4].

<sup>1</sup>A reduction system is locally confluent if every pair of two terms reduced from another in one step may be reduced to a common term.

### 3.3 Reduction to $Th^1$

**Definition 3.3.1 (Type Reduction  $\mathcal{R}_\times^1$ )** Let  $\mathcal{R}_\times^1$  be the transitive and substitutive type-reduction relation applied to types that do not contain  $\mathbf{T}$  generated by:

1.  $A \times (B \times C) > (A \times B) \times C$
2.  $(A \times B) \rightarrow C > A \rightarrow (B \rightarrow C)$
3.  $A \rightarrow (B \times C) > (A \rightarrow B) \times (A \rightarrow C)$

Note that the rules generating  $\mathcal{R}_\times^1$  are exactly the rules generating  $\mathcal{R}^1$  that do not refer to  $\mathbf{T}$ .

**Lemma 3.3.2**  $\mathcal{R}_\times^1$  is confluent and strongly normalizes.

**Proof Sketch:**

This follows from the proof that  $\mathcal{R}^1$  is confluent and strongly normalizes. Let  $A$  be a type that does not contain  $\mathbf{T}$ . Reducing  $A$  in  $\mathcal{R}^1$  does not introduce  $\mathbf{T}$  so the possible reductions are exactly those in  $\mathcal{R}_\times^1$ .

**Definition 3.3.3 ( $\mathcal{R}_\times^1$  Normal Forms)** Let  $A$  be a type not containing  $\mathbf{T}$ . The unique normal form of  $A$  under  $\mathcal{R}_\times^1$  is denoted  $nf_\times(A)$ .

**Lemma 3.3.4** Let  $A$  be a non-trivial type. Then,  $Th_\times^1 \vdash A = nf_\times(A)$ .

**Proof Sketch:**

Since each reduction step of  $\mathcal{R}_\times^1$  corresponds to a rule in  $Th_\times^1$ , this follows trivially.

**Lemma 3.3.5** Let  $A$  be a non-trivial type. Then,  $Th_{\times\mathbf{T}}^1 \vdash A = (nf_\times \circ nf_{\mathbf{T}})(A)$ .

**Lemma 3.3.6** Let  $A$  be a non-trivial type. Then,  $(nf_\times \circ nf_{\mathbf{T}})(A) = nf(A)$ .

**Proof Sketch:**

The output of  $(nf_\times \circ nf_{\mathbf{T}})$  does not contain  $\mathbf{T}$  (they were removed by  $nf_{\mathbf{T}}$ ) and is irreducible by the rules not referring to  $\mathbf{T}$  ( $nf_\times$  reduces according to these rules). Therefore,  $(nf_\times \circ nf_{\mathbf{T}})(A)$  is irreducible in  $\mathcal{R}^1$ . Since  $nf_{\mathbf{T}}$  and  $nf_\times$  reduce terms according to rules of  $\mathcal{R}^1$ ,  $(nf_\times \circ nf_{\mathbf{T}})(A)$  must be the normal form of  $A$  under  $\mathcal{R}^1$ .

### 3.4 Reduction to $Th^0$

**Definition 3.4.1 (Type Ordering)** We choose a fixed total order on the set of all types, denoted  $\preceq$ . We define  $\preceq$  more precisely in 4.5. For the moment, we do not specify the properties of  $\preceq$  other than it is a total ordering of the set of all types. We write  $A \prec B$  in the case  $A \preceq B \wedge A \neq B$ .

Note that this ordering does not consider isomorphic types equal unless they actually are identical.

**Definition 3.4.2 (Type Reduction  $\mathcal{R}_{\rightarrow}^1$ )** Let  $\mathcal{R}_{\rightarrow}^1$  be a transitive and substitutive type-reduction relation applied to types built from type variables and  $\rightarrow$ .  $\mathcal{R}_{\rightarrow}^1$  is generated by:

1.  $A \rightarrow (B \rightarrow C) > B \rightarrow (A \rightarrow C)$  if  $B \prec A$

**Lemma 3.4.3**  $\mathcal{R}_{\rightarrow}^1$  is confluent and strongly normalizes.

**Proof:**

This may be proven by structural induction. Let  $A$  be the type in question.  $A$  is composed of type variables and  $\rightarrow$ . The base case is  $A$  is a type variable.  $A$  is trivially irreducible so  $\mathcal{R}_{\rightarrow}^1$  is trivially strongly normalizing in this case. If  $A$  is not a type variable,  $A$  is of the form  $A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$  where  $A_i$  is a type described by the induction hypothesis and  $B$  is a type variable.

Reduction steps may either reduce the  $A_i$  terms or rearrange  $A_i$  terms (i.e. for some permutation  $\sigma$ ,  $A > A_{\sigma(1)} \rightarrow \dots \rightarrow A_{\sigma(n)} \rightarrow B$ ). By the induction hypothesis, reducing each  $A_i$  terminates after a finite number of reduction steps so only a finite number of reduction steps are necessary to normalize all  $A_i$  terms. The number of reductions without reducing an  $A_i$  term is also finite - each step swapping  $A_i$  terms brings  $A_{\sigma(1)}, \dots, A_{\sigma(n)}$  closer to being sorted. Therefore, reduction of  $A$  terminates.

Let  $C_i$  be the unique normal form of  $A_i$  under the induction hypothesis. The normal form of  $A$  is of the form  $C_{\sigma(1)} \rightarrow \dots \rightarrow C_{\sigma(n)} \rightarrow B$ . Note that for all  $i$  and  $j$ ,  $i \leq j$  implies  $C_{\sigma(i)} \preceq C_{\sigma(j)}$  and  $C_{\sigma(1)}, \dots, C_{\sigma(n)}$  is the unique sorted version of  $C_1, \dots, C_n$ . Therefore, the normal form of  $A$  is unique and  $\mathcal{R}_{\rightarrow}^1$  strongly normalizes.

**Definition 3.4.4 ( $\mathcal{R}_{\rightarrow}^1$  Normal Forms)** Let  $A$  be a type built from type variables and  $\rightarrow$ . The unique normal form of  $A$  under  $\mathcal{R}_{\rightarrow}^1$  is denoted  $nf_{\rightarrow}(A)$ .

**Lemma 3.4.5** *Let  $A$  be a type built from type variables and  $\rightarrow$ . Then,  $Th^1 \vdash A = nf_{\rightarrow}(A)$ .*

**Lemma 3.4.6** *Let  $A$  and  $B$  be types built from type variables and  $\rightarrow$ . Then,  $Th^1 \vdash A = B$  if and only if  $nf_{\rightarrow}(A) = nf_{\rightarrow}(B)$ .*

**Proof Sketch:**

This may be proven by induction over the length of the proof that  $Th^1 \vdash A = B$  - any step is an application of the swap reduction axiom of  $Th^1$ . Given two consecutive equations in the proof, one of these equations may be reduced to the other by the sorted swap rule of  $\mathcal{R}_{\rightarrow}^1$ . Since  $\mathcal{R}_{\rightarrow}^1$  is confluent and strongly normalizable, these two equations have the same normal form. Therefore, all equations in the proof have the same normal form.

## 4 Efficient Implementation

### 4.1 Overview

To avoid exponential expansion arising from reduction using the distributivity rule, we use a hash-consing representation and memoization. Hash-consing allows individual reductions to be performed with constant space utilization. Handles from hash-consing are also used to obtain a cheap ordering of types. Memoization allows a polynomial upper bound on the reductions since each distinct type is reduced at most once. Space sharing techniques were used in [12] to avoid exponential space usage, but were not used for a time speedup.

Historically, hash consing is a technique originally used in LISP to avoid duplication of lists. In LISP, list structures are only created by the cons operation. By modifying cons with the help of hashing techniques, no two invocations of cons would ever return distinct copies of the same data. An early example of this technique is presented in [9]. While limiting the ability to modify lists generated in such a manner, this technique allows greater space efficiency and constant time equality checking [1].

Memoization is a variation of dynamic programming preserving the natural top-down recursive approach while storing the results of each sub-problem solved [7].

An example implementation in SML/NJ is given in appendix A.

### 4.2 Efficient Reduction to $Th_{\times}^1$

**Lemma 4.2.1** *Let  $A$  be a type. There exists an algorithm outputting  $nf_{\mathbf{T}}(A)$  using  $O(|A|)$  time and  $O(|A|)$  space.*

**Proof Sketch:**

$nf_{\mathbf{T}}(A)$  is easily calculated recursively in a bottom up fashion.

**4.3 Efficient Reduction to  $Th^1$** 

**Definition 4.3.1 (Type Degree)** Let  $A$  be a type of the form  $A_1 \times \dots \times A_n$  where  $A_i$  is built from type variables and  $\rightarrow$ . The *degree* of  $A$ , denoted  $degree(A)$ , is  $n$ .

**Lemma 4.3.2** Let  $A$  be a type. Then,  $degree(nf(A)) \leq |A|$ .

**Lemma 4.3.3** Let  $A$  be a non-trivial type. There exists an algorithm outputting  $nf(A)$  in  $O(|A|^2)$  time and  $O(|A|^2)$  space using space sharing techniques.

**Proof Sketch:**

$nf(A)$  can be constructed in a bottom up fashion. Let  $A' = nf_{\mathbf{T}}(A)$ . If  $A' = L \rightarrow R$ , then  $O(degree(nf(L)) * degree(nf(R)))$  new types are constructed after  $nf(L)$  and  $nf(R)$  are constructed. Otherwise, if  $A' = L \times R$ , then  $O(degree(nf(R)))$  new types are constructed after  $nf(L)$  and  $nf(R)$  are constructed. In either case,  $O(|L| * |R|)$  types are constructed after  $nf(L)$  and  $nf(R)$  are constructed. In the entire construction of  $nf(A)$ ,  $O(|A|^2)$  types are constructed.

**4.4 Efficient Reduction to  $Th^0$** 

**Definition 4.4.1 (Type Right Height)** Let  $A$  be a type built from type variables and  $\rightarrow$ .  $A$  is of the form  $A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$  where  $B$  is a type variable. The *right height* of  $A$ , denoted  $right\_height(A)$ , is  $n$ .

**Definition 4.4.2 (Total Distinct Right Height)** Let  $A$  be a non-trivial type. The *total distinct right height* of  $A$ , denoted  $total\_distinct\_right\_height(A)$ , is the sum of all the right heights of the types forming  $A$  which occur in sub-expressions of  $A$  besides  $X \rightarrow A$ .

Consider an algorithm sorting  $A$  in a bottom up fashion. If  $A$  is of the form  $A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$ , the  $A_i$  types and  $B$  are sorted recursively and then the  $A_i$  types are sorted relative to each other. With a memoizing sort implementation, the total number of types involved as inputs to the various calls to sort is  $total\_distinct\_right\_height(A)$ .

**Lemma 4.4.3** Let  $A$  be a non-trivial type. Then,  $total\_distinct\_right\_height(A) \in O(|A|^2)$ .

**Proof Sketch:**

The proof of this lemma is similar to that of Lemma 4.3.3.

**Lemma 4.4.4** Let  $A$  be a non-trivial type. Let  $A' = nf(A)$ .  $A'$  is of the form  $A'_1 \times \dots \times A'_n$ . There exists an algorithm outputting  $nf_{\rightarrow}(A'_1) \times \dots \times nf_{\rightarrow}(A'_n)$  in  $O(|A|^2 \log |A|)$  time using  $O(|A|^2)$  space.

**Proof Sketch:**

$A'_i$  is of the form  $A'_{i,1} \rightarrow \dots \rightarrow A'_{i,n} \rightarrow B$ .  $nf_{\rightarrow}(A'_i)$  may be constructed by recursively normalizing each  $A'_{i,j}$  and sorting. The total length of all the lists to be sorted is  $O(|A|^2)$  by Lemma 4.4.3 so the total time sorting with memoization is  $O(|A|^2 \log |A|)$  and the total space is  $O(|A|^2)$ .

This is a tight bound for this particular algorithm - an example that has is asymptotically this slow is  $(A_1 \times \dots \times A_n) \rightarrow ((B_1 \rightarrow C_1) \times \dots \times (B_n \rightarrow C_n))$ .

**Lemma 4.4.5** Let  $A$  and  $B$  be types. Let  $n = \max(|A|, |B|)$ . Whether  $Th_{\times \mathbf{T}}^1 \vdash A = B$  is decidable in  $O(n^2 \log n)$  time and  $O(n^2)$  space.

**Proof Sketch:**

If  $A$  or  $B$  is trivial,  $Th_{\times \mathbf{T}}^1 \vdash A = B$  if and only if  $nf_{\mathbf{T}}(A) = nf_{\mathbf{T}}(B) = \mathbf{T}$ . This is decidable in  $O(n)$  time and  $O(n)$  space.

Let  $A'$  and  $B'$  be defined as in lemma 4.4.4.  $A'$  and  $B'$  are constructed in  $O(n^2)$  time and  $O(n^2)$  space. Sorting  $A'$  and  $B'$  may be done in  $O(n^2 \log n)$  time. This takes a total of  $O(n^2 \log n)$  time and  $O(n^2)$  space.

## 4.5 Costs of Hash-Consing and Memoization

Technically, this analysis ignores costs associated with hash-consing and memoization - overhead from these methods can add a logarithmic factor to operations that would normally take constant time. In practice, hashing techniques would mostly negate this, but stricter analysis shows that it does not change the asymptotic behavior. In the reductions to  $Th^1$ ,  $O(n^2)$  types are constructed so the additional cost of hash-consing is  $O(n^2 \log n)$  so it is no worse asymptotically than later sorting. The costs of memoization and hash-consing while sorting is  $O(n^2)$  types looked up at total cost  $O(n^2 \log n)$  so the cost is asymptotically the same as the actual sorting.

Another cost ignored is the cost of  $\preceq$ . In the reduction, the description of the total ordering was purposely left vague. This allows hash-consing handles to be used to order types in constant time under the common practical assumption that sufficiently large numbers can be stored in constant space.

## 5 Conclusions

We have presented an algorithm building on a sizable body of theoretical work and bringing it within practical limits. Deciding isomorphisms in a practical manner allows many applications currently dependent on heuristics for the discovery of isomorphisms to cheaply move to sounder theoretical footing.

## 6 Acknowledgments

I would like to acknowledge my advisor Assaf Kfoury who got me interested in type isomorphisms last fall and pointed out Josh Auerbach et al's paper. Assaf Kfoury also gave me the task of implementing hash-consing for recursive types [6], giving me the understanding of hash-consing necessary for this paper. Jens Palsberg was also helpful, both in mentioning his related work and in enlightening us about early work on this subject.

## A Example Implementation

This appendix gives source code listings for our implementation of this algorithm. Each file corresponds to one step in the reduction. An interesting property of the data types used in the first two reduction steps (those only based on the structure of the types involved) is that once a type is converted to the output type of that reduction, it is irreducible under that reduction system.

### A.1 Isomorphism.sml

```
(* Isomorphism.sml *)

(* This structure is a wrapper over the normalization functions of the various
   type schemas allowing type isomorphism to be decided. *)

structure Isomorphism =
  struct
    fun decide(a, b) =
      case (Schema1.normalize(a), Schema1.normalize(b)) of
        (NONE, NONE) =>
          true
      | (SOME(a'), SOME(b')) =>
          let
            val normalize =
```

```

                Schema3.normalize o Schema2.normalize
            in
                normalize(a') = normalize(b')
            end
        | _ =>
            false
    end
end

```

## A.2 Schema0.sml

(\* Schema0.sml \*)

(\* This structure models arbitrary types within the first order lambda calculus with terminal objects and surjective pairing. \*)

```

structure Schema0 =
  struct
    (* datatypes *)

    datatype Schema0 =
      Arrow0 of Schema0 * Schema0
    | Cross0 of Schema0 * Schema0
    | Terminal0
    | Variable0 of int
  end
end

```

## A.3 Schema1.sml

(\* Schema1.sml \*)

(\* This structure models arbitrary types in the first order lambda calculus with surjective pairing. \*)

(\* The reduce function returns either NONE when the input type tau is isomorphic to the terminal object or SOME(tau') where tau' is a type not containing the terminal object and tau is isomorphic to tau' \*)

```

structure Schema1 =
  struct
    (* datatypes *)

    datatype Schema1 =
      Arrow1 of Schema1 * Schema1
    | Cross1 of Schema1 * Schema1
    | Variable1 of int

    (* normalization function *)

    fun normalize(Schema0.Arrow0(x, y)) =
      (case (normalize(x), normalize(y)) of
        (NONE, NONE) =>

```

```

        NONE
      | (NONE, SOME(y')) =>
        SOME(y')
      | (SOME(x'), NONE) =>
        NONE
      | (SOME(x'), SOME(y')) =>
        SOME(Arrow1(x', y'))
    | normalize(Schema0.Cross0(x, y)) =
      (case (normalize(x), normalize(y)) of
        (NONE, NONE) =>
          NONE
        | (NONE, SOME(y')) =>
          SOME(y')
        | (SOME(x'), NONE) =>
          SOME(x')
        | (SOME(x'), SOME(y')) =>
          SOME(Cross1(x', y')))
    | normalize(Schema0.Terminal0) =
      NONE
    | normalize(Schema0.Variable0(n)) =
      SOME(Variable1(n))
  end

```

#### A.4 Schema2a.sml

```

(* Schema2a.sml *)

(* This structure models arbitrary types in the first order lambda
   calculus. *)

structure Schema2a =
  struct
    (* datatypes *)

    datatype Schema2aHandle =
      Handle2a of int

    datatype Schema2aData =
      Arrow2a of Schema2aHandle * Schema2aHandle
    | Variable2a of int;

    (* sorting *)
    structure HandleOrdKey : ORD_KEY =
      struct
        type ord_key =
          Schema2aHandle

        fun compare(Handle2a(x), Handle2a(y)) =
          Int.compare(x, y)
      end
  end

```

```

structure DataOrdKey : ORD_KEY =
  struct
    type ord_key =
      Schema2aData

    fun compare(Arrow2a(x1, x2), Arrow2a(y1, y2)) =
      (case HandleOrdKey.compare(x1, y1) of
        EQUAL =>
          HandleOrdKey.compare(x2, y2)
      | DEFAULT =>
        DEFAULT)
    | compare(Arrow2a(_), Variable2a(_)) =
      LESS
    | compare(Variable2a(_), Arrow2a(_)) =
      GREATER
    | compare(Variable2a(x), Variable2a(y)) =
      Int.compare(x, y)
  end

(* hashcons functions *)

local
  val next_handle =
    ref 0

  structure HandleMap =
    BinaryMapFn(HandleOrdKey)

  val handle_to_data_map =
    ref(HandleMap.empty : Schema2aData HandleMap.map)

  structure DataMap =
    BinaryMapFn(DataOrdKey)

  val data_to_handle_map =
    ref(DataMap.empty : Schema2aHandle DataMap.map)

  fun get_hashconsed(data) =
    let
      val current_handle =
        !next_handle

      val current_handle' =
        Handle2a(current_handle)
    in
      case DataMap.find(!data_to_handle_map, data) of
        NONE =>
          (data_to_handle_map := DataMap.insert(!data_to_handle_map,
                                                  data,
                                                  current_handle');
           handle_to_data_map := HandleMap.insert(!handle_to_data_map,

```

```

current_handle',
data);

        next_handle := current_handle + 1;
        current_handle')
    | SOME(found_handle) =>
        found_handle
    end
in
exception IllegalHandle2aException of HandleOrdKey.ord_key;

val compare =
    HandleOrdKey.compare

(* hashconsing wrappers *)

fun get_arrow(x, y) =
    get_hashconsed(Arrow2a(x, y))

fun get_data(h) =
    case HandleMap.find(!handle_to_data_map, h) of
        NONE =>
            raise IllegalHandle2aException(h)
        | SOME(data) =>
            data

    fun get_variable(x) =
        get_hashconsed(Variable2a(x))
    end
end
end

```

## A.5 Schema2.sml

(\* Schema2.sml \*)

(\* This structure models surjective pairs of arbitrary types in the first order lambda calculus. \*)

(\* The normalize function "moves up" crosses so that the result has no occurrences of crosses below the top level. \*)

```

structure Schema2 =
    struct
        (* datatypes *)

        datatype Schema2Handle =
            Handle2 of int

        datatype Schema2Data =
            Arrow2 of Schema2a.Schema2aHandle
            | Cross2 of Schema2Handle * Schema2a.Schema2aHandle
    end

```

```

(* sorting/mapping *)

structure HandleOrdKey : ORD_KEY =
  struct
    type ord_key =
      Schema2Handle

    fun compare(Handle2(x), Handle2(y)) =
      Int.compare(x, y)
  end

structure DataOrdKey : ORD_KEY =
  struct
    type ord_key =
      Schema2Data

    fun compare(Arrow2(x), Arrow2(y)) =
      Schema2a.compare(x, y)
      | compare(Arrow2(_), Cross2(_)) =
      LESS
      | compare(Cross2(_), Arrow2(_)) =
      GREATER
      | compare(Cross2(x1, x2), Cross2(y1, y2)) =
      (case HandleOrdKey.compare(x1, y1) of
        EQUAL =>
          Schema2a.compare(x2, y2)
        | DEFAULT =>
          DEFAULT)
  end

(* hashcons functions *)

local
  val next_handle =
    ref 0

  (* handle to data mapping *)

  structure HandleMap =
    BinaryMapFn(HandleOrdKey)

  val handle_to_data_map =
    ref(HandleMap.empty : Schema2Data HandleMap.map)

  (* data to handle mapping *)

  structure DataMap =
    BinaryMapFn(DataOrdKey)

  val data_to_handle_map =
    ref(DataMap.empty : Schema2Handle DataMap.map)
end

```

```

(* actual hashconsing *)

fun get_hashconsed(data) =
  let
    val current_handle =
      !next_handle

    val current_handle' =
      Handle2(current_handle)
  in
    case DataMap.find(!data_to_handle_map, data) of
      NONE =>
        (data_to_handle_map := DataMap.insert(!data_to_handle_map,
                                              data,
                                              current_handle');
         handle_to_data_map := HandleMap.insert(!handle_to_data_map,
                                              current_handle',
                                              data);

         next_handle := current_handle + 1;
         current_handle')
      | SOME(found_handle) =>
        found_handle
    end
  in
    exception IllegalHandle2Exception of HandleOrdKey.ord_key;

    (* hashconsing wrappers *)

    fun get_arrow(x) =
      get_hashconsed(Arrow2(x))

    fun get_cross(x, y) =
      get_hashconsed(Cross2(x, y))

    fun get_data(h) =
      case HandleMap.find(!handle_to_data_map, h) of
        NONE =>
          raise IllegalHandle2Exception(h)
        | SOME(data) =>
          data
      end

    (* reduction functions *)

    fun reduce_cross(x, y) =
      case (get_data(x), get_data(y)) of
        (Arrow2(_), Arrow2(y')) =>
          get_cross(x, y')
        | (Arrow2(_), Cross2(y1', y2')) =>
          get_cross(reduce_cross(x, y1'), y2')
      end
  end

```

```

    | (Cross2(_), Arrow2(y')) =>
        get_cross(x, y')
    | (Cross2(_), Cross2(y1', y2')) =>
        get_cross(reduce_cross(x, y1'), y2')

fun reduce_arrow(x, y) =
  case (get_data(x), get_data(y)) of
    (Arrow2(x'), Arrow2(y')) =>
        get_arrow(Schema2a.get_arrow(x', y'))
  | (Arrow2(x'), Cross2(y1', y2')) =>
        get_cross(reduce_arrow(x, y1'),
                  Schema2a.get_arrow(x', y2'))
  | (Cross2(x1', x2'), Arrow2(y')) =>
        reduce_arrow(x1',
                    get_arrow(Schema2a.get_arrow(x2', y')))
  | (Cross2(_), Cross2(y1', y2')) =>
        reduce_cross(reduce_arrow(x, y1'),
                    reduce_arrow(x, get_arrow(y2')))

(* normalization function *)

fun normalize(Schema1.Arrow1(x, y)) =
  reduce_arrow(normalize(x), normalize(y))
  | normalize(Schema1.Cross1(x, y)) =
  reduce_cross(normalize(x), normalize(y))
  | normalize(Schema1.Variable1(n)) =
  get_arrow(Schema2a.get_variable(n))
end

```

## A.6 Schema3.sml

```

(* Schema3.sml *)

(* This structure models sorted surjective pairs of sorted types in the first
   order lambda calculus. *)

(* The normalize function sorts the input types using commutativity of "cross"
   and "Swap". *)

structure Schema3 =
  struct
    local
      structure Schema2aHandleMap =
        BinaryMapFn(Schema2a.HandleOrdKey)

      (* memoization *)

      val memoized_2a_3a =
        ref(Schema2aHandleMap.empty : Schema2a.Schema2aHandle Schema2aHandleMap.map)

      fun memoize_2a_3a(x, y) =

```

```

(memoized_2a_3a := Schema2aHandleMap.insert(!memoized_2a_3a, x, y);
 y);

(* list extraction *)

fun extract_arrow_list(h) =
  case Schema2a.get_data(h) of
    Schema2a.Arrow2a(x, y) =>
      let
        val (x', y') =
          extract_arrow_list(y)
      in
        (x::x', y')
      end
    | Schema2a.Variable2a(n) =>
      (nil, Schema2a.get_variable(n))

fun extract_cross_list(h) =
  case Schema2.get_data(h) of
    Schema2.Arrow2(x_2a) =>
      x_2a::nil
    | Schema2.Cross2(x_2, y_2a) =>
      y_2a::extract_cross_list(x_2)

(* list folding *)

val fold_arrow_list =
  Array.foldr(fn(a, b) => Schema2a.get_arrow(a, b))

val foldi_cross_list =
  Array.foldri(fn(_, a, b) => Schema2.get_cross(b, a))

(* normalization *)

fun normalize_a(x) =
  case Schema2aHandleMap.find(!memoized_2a_3a, x) of
    NONE =>
      let
        val (input_list, output) =
          extract_arrow_list(x)

        val input_array =
          Array.fromList(input_list)
      in
        Array.modify(normalize_a)(input_array);

        ArrayQSort.sort(Schema2a.compare)(input_array);

        memoize_2a_3a(x, fold_arrow_list(output)(input_array))
      end
    | SOME(h) =>

```

```

                                h
in
  fun normalize(x) =
    let
      val arrow_2a_array =
        Array.fromList(extract_cross_list(x))
    in
      Array.modify(normalize_a)(arrow_2a_array);

      ArrayQSort.sort(Schema2a.compare)(arrow_2a_array);

    let
      val first =
        Schema2.get_arrow(Array.sub(arrow_2a_array, 0))
    in
      foldi_cross_list(first)(arrow_2a_array, 1, NONE)
    end
    end
  end
end

```

## References

- [1] John Allen. *Anatomy of LISP*. McGraw-Hill Book Company, New York, 1978.
- [2] Joshua Auerbach, Charles Barton, and Mukund Raghavachary. Type isomorphisms with recursive types. Technical Report RC 21247, IBM Research Division, Yorktown Heights, New York, August 1998.
- [3] Joshua Auerbach and Mark C. Chu-Carroll. The mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Technical Report RC 20178, IBM Research Division, Yorktown Heights, New York, February 1997.
- [4] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 1:1–20, 1991.
- [5] Kim B. Bruce and Giuseppe Longo. Provable isomorphisms and domain equations in models of typed languages. In *ACM Symposium on Theory of Computing*, May 1985.
- [6] Jeffrey Considine. Efficient hash-consing of recursive types. Technical Report 2000-06, Boston University, Boston, Massachusetts, January 2000.
- [7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.
- [8] Roberto Di Cosmo. *Isomorphisms of types: from lambda-calculus to information retrieval and language design*. Birkhauser, Boston, Massachusetts, 1995.
- [9] E. Goto. Monocopy and associative algorithms in an extended lisp. Technical report, University of Tokyo, Tokyo, Japan, May 1974.
- [10] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, London, 1996.
- [11] Jens Palsberg and Tian Zhao. Efficient and flexible matching of recursive types. manuscript, January 2000.

- [12] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1), 1989.
- [13] Mikael Rittri. Retrieving library identifiers by equational matching of types in 10th int. conf. on automated deduction. *Lecture Notes in Computer Science*, 449, July 1990.
- [14] Serjey V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.