Unification and Matching Modulo Type Isomorphism

Dan Dougherty[1] and Carlos C. Martínez[2]

[1] Worcester Polytechnic Institute,
Department of Computer Science,
Worcester, MA 01609 USA
dd@cs.wpi.edu
[2] Wesleyan University,
Department of Mathematics and Computer Science,
Middletown, CT 06459 USA
cmartinez@wesleyan.edu

Abstract. We present some initial results in an investigation of higher-order unification and matching in the presence of type isomorphism.

1 Introduction

Two simple types S and T are isomorphic if their interpretations are isomorphic in every model of the simply-typed λ-calculus, or equivalently, if there exist terms $f : S \to T$ and $g : T \to S$, such that $g \circ f$ and $f \circ g$ are each $\beta\eta$-convertible with the identity.

The study of type isomorphism is currently an active area of research, well-represented in Roberto di Cosmo's book [DC95]. There are connections with logic (cf Tarski's "high school algebra problem"), with category theory [FCB02],and with information retrieval in software libraries [Rit90,Rit91,RT91,ZW93].

Some of the most interesting work concerns polymorphic type disciplines but our focus here will be restricted to simple types. Indeed, in this preliminary report *we consider arrow-types only.* Much of complexity of type isomorphism per se is avoided in this setting, but the novel issues surrounding unification and matching still arise, as we will see. We also work only with pure terms (ie, with no constants).

It is natural to want be sensitive to type isomorphism when one is doing higher-order rewriting, in particular if we are interested in code querying or transformation. For example, suppose one wants to perform a code transformation with a certain function-pattern of type $(A \to B \to C)$. The use of standard higher-order matching allows us to ignore the *names* of the arguments in a code fragment potentially matching the pattern. But the *order* in which these parameters appear in the code is significant, since it determines the code's type. Since the types $(A \to B \to C)$ and $(B \to A \to C)$ are isomorphic, a code fragment of the latter type may very well be a candidate that we want to consider. So it seems that a more refined tool than standard higher-order matching would be useful.

Indeed, what we require is a richer notion of matching which accepts a match as long as the term being matched is *the same as the target term modulo a type isomorphism*. That is, type isomorphism induces a notion of equality on terms, more lenient than equality modulo $\beta\eta$, and it is this equality that we want to guide our matching. To our knowledge this relation on terms — which we call "term isomorphism" for want of a better phrase — has not been explored in the published literature.

Definition 1. Given $s : S$ and $t : T$, we say that s and t are *term isomorphic*, written $s \simeq t$ if there is a type isomorphism $p : S \to T$ with $ps = t$.

Here and below, the equality symbol "=" denotes convertibility modulo β and η.

It is also natural to consider higher-order *unification* modulo type isomorphism, for example in the context of higher-order logic programming, when λ-terms are ubiquitous as data structures, and type isomorphism induces a natural equivalence on this data.

In fact the problems we face and the solutions we propose arise equally in unification and in matching. So for simplicity in this abstract we focus on unification: our goal is to explore algorithms to solve the following problem.

Unification modulo Type Isomorphism

INPUT: Two terms s and t, of isomorphic types

OUTPUT: A substitution θ such that $\theta s \simeq \theta t$

2 An easy algorithm

We first note that the consideration of type isomorphism does not have any consequences as to decidability.

Dezani [DC76] defined the notion of *finite hereditary permutation*, which is a term of the following form:

$$\lambda z x_1 \ldots x_n . z(p_1 x_{\pi(1)}) \ldots (p_n x_{\pi(n)})$$

where π is a permutation of $[1..n]$ and each p_i is a finite hereditary permutation, and proved that a normalizable untyped term p is invertible iff it is a finite hereditary permutation.

Any finite hereditary permutation is typable. So, as observed in [BCL92], a term $p : A \to B$ is a type isomorphism iff it is a finite hereditary permutation.

It is not hard to see that at any type $A \to B$ there are only finitely many finite hereditary permutations. So we have the following

Proposition 2. *For each pair of types S and T there are finitely many type isomorphisms $p : S \to T$, and these can be effectively generated given S and T.*

In this way we can reduce any unification/matching problem modulo type isomorphism to finitely many similar standard problems.

Definition 3 (Naive algorithm). Given s and t as an instance of the problem of unification (or matching) modulo type isomorphism, the naive algorithm for the problem is:

> *For each type isomorphism p from S to T, generate the standard problem $s = pt$. If any of these problems is solvable, return that solution; otherwise return failure.*

Proposition 4. *The algorithm of Definition 3 is sound and complete relative to the soundness and completeness of the standard algorithms used.*

In particular, at any type where the classical higher-order matching is decidable, the problem of matching modulo type isomorphism is decidable. Also, unification of higher-order patterns [Mil91] modulo type isomorphism is decidable.

This is reassuring, but since there can be exponentially many finite hereditary permutations at a given pair of types, such a generate-and-test algorithm is clearly not practical. We want to "build-in" (in the sense of [Plo72]) type isomorphism to the matching algorithm.

3 Building in type isomorphism

Consider a set of standard transformations for pure higher-order unification, such as described in [GS89]: Imitation, Projection, Variable Elimination, and Decomposition. The idea here is to enhance this set of transformations so that in addition to gradually building up an answer substitution as a problem is solved, we also build up a finite hereditary permutation that serves as part of the witness to the problem's solution.

The main work in unification transformations is the "guessing" component, where substitutions are generated and propagated. These are represented in higher-order unification by the Imitation, Projection, and Variable Elimination transformations. There is another transformation — Decomposition — which breaks a problem into subproblems or recognizes that the current problem is not solvable and reports failure.

(Standard) Decomposition

$$\frac{E\ ;\ (\lambda\overline{x}.x_e\,\overrightarrow{t}\ \doteq\ \lambda\overline{yx}.x_d\,\overrightarrow{u})}{E\ ;\ (\lambda\overline{x}.t_1\ \doteq\ \lambda\overline{x}.u_1)\ ;\ \ldots;\ (\lambda\overline{x}.t_k\ \doteq\ \lambda\overline{x}.u_k)}\quad \text{if } e=d\text{, else fail}$$

Perhaps surprisingly, with a little work one can see that in defining transformations for higher-order unification modulo type isomorphism Imitation, Projection, and Variable Elimination need not be changed at all. The complexity of considering type isomorphism is completely reflected in the need for a refinement of the Decomposition transformation.

This is because the Decomposition transformation is the embodiment of a basic fact about *equality* between terms: by the Church-Rosser theorem $\lambda\overline{x}.x_e\,\overrightarrow{t}$ and $\lambda\overline{yx}.x_d\,\overrightarrow{u}$ are $\beta\eta$-equal if and only if $e=d$ and corresponding immediate subterms are equal. This is false when the equality in question is $\simeq$.

So we see that the essence of building in type isomorphisms to the unification algorithm is to have a good characterization of when a equation is valid (as opposed to unifiable) modulo type isomorphism. That is, we are led to seek an incremental analysis of term isomorphism.

4 Characterizing term isomorphism

4.1 Labelled trees

In order to analyze the combinatorics of term isomorphism it is convenient to abstract away from variable binding and work with ordinary labelled trees.

As a data structure for manipulating types and terms, trees are typically implemented as "ordered trees" in the sense that for each node x there is an ordering on the children of x. The notion of tree mapping below could be defined more simply just by saying that a tree mapping is required to respect the roots of the trees and the child-parent relationships, but not the ordering on the children of a node. The equivalent (more tedious) definition we give, in terms of explicit permutations on addresses, is necessary in order to subsequently define the key notion of *labelled tree mapping*.

We work with partial functions between trees in anticipation of needing to construct functions incrementally in unification algorithms.

Definition 5. Let $\mathcal{T}$ and $\mathcal{U}$ be tree domains. A *partial tree mapping* $\Phi:\mathcal{T}\to\mathcal{U}$ is a pair (Φ^a,Φ^p) such that

- Φ^a is a partial function from the addresses of $\mathcal{T}$ to the addresses of $\mathcal{U}$ whose domain is a subtree of $\mathcal{T}$,
- for each $\alpha\in\text{domain}(\Phi^a)$, $\Phi^p(\alpha)$ is a permutation of $[1..\mathit{arity}(\alpha)]$, and
- Φ^a maps the i-th child of α to the $\Phi^p(\alpha)(i)$-th child of $\Phi^a(\alpha)$. That is, $\Phi^a(\alpha\cdot i)=\Phi^a(\alpha)\cdot\Phi^p(\alpha)(i)$.

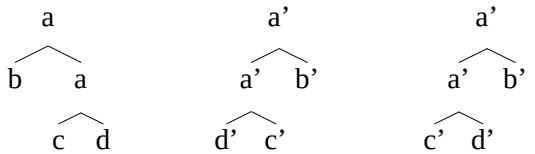
Definition 6. Let $\mathcal{L}$ be a set of *labels* together with an arity function $\mathit{arity}:\mathcal{L}\to\mathbb{N}$. A *labelled tree* is an tree domain $\mathcal{T}$ and a partial map $\mathit{label}:\mathcal{T}\to\mathcal{L}$ labelling some of the nodes of $\mathcal{T}$ such that if $\mathit{label}(\alpha)$ is defined then the number of children of α is $\mathit{arity}(\mathit{label}(\alpha))$. Furthermore, a labelled tree comes with an arity-consistent equivalence relation $\approx$ defined on its set of labels.

If $\mathcal{T}$ and $\mathcal{U}$ are labelled trees, a *partial labelled tree mapping* from $\mathcal{T}$ to $\mathcal{U}$ is a partial tree mapping $\Phi:\mathcal{T}\to\mathcal{U}$ satisfying

- if $\mathit{label}(\alpha)=\mathit{label}(\beta)$ then $\mathit{label}(\Phi^a(\alpha))=\mathit{label}(\Phi^a(\beta))$
- if $\mathit{label}(\alpha)\approx\mathit{label}(\beta)$ then $\Phi^p(\alpha)=\Phi^p(\beta)$.

The labelled trees $\mathcal{T}$ and $\mathcal{U}$ are *isomorphic* as labelled trees if there is a labelled tree homomorphism Φ from $\mathcal{T}$ to $\mathcal{U}$ with Φ^a a bijection between the nodes of $\mathcal{T}$ and $\mathcal{U}$.

Example 7.



The first and second labelled trees above are isomorphic; the first and third labelled trees are *not* isomorphic.

4.2 Labelled trees for terms

We first need some notation allowing us to track relationships among bound-variable occurrences in a Böhm tree.

Definition 8. Let $t : T$ be a pure term. We let $BT(t)$ denote the Böhm tree for t. In $BT(t)$ we make the following definitions.

If the node at address α has binder $\lambda x_1 \ldots x_n$ and head variable y then we say that y has an *occurrence* at address α and for each i we say that bound variable x_i is *introduced* at address α, at *index* i.

We assume that in $BT(t)$ no name is used for both a free and bound variable, and no bound variable name is introduced in more than one place.

Note the distinction between the introduction of a bound variable and an occurrence of a variable (in particular an introduction is not an occurrence). Note also that a variable may have any number of occurrences, but it is only introduced at one address and index.

Now, given a term t, we define $LT(t)$, the *labelled tree associated to* t, essentially by forgetting the binders in the Böhm tree for t (so that $LT(t)$ will fail to have a label at those tre addresses with free variables). The precise definition is Definition 9 below. By using bound-variable names from $BT(t)$ as our labels we are of course not determining the labels of $LT(t)$ uniquely. But since we have adopted a convention that Böhm trees always obey our strong variable conventions about not reusing variable names, the differences in labelled trees we obtain for a term are identical up to renaming of labels. Indeed it will often be convenient to be able to assume that the labels of a given pair of trees are disjoint.

Definition 9. The underlying tree of $LT(t)$ is the underlying tree of $BT(t)$. If there is a bound-variable occurrence at α in $BT(t)$ the label in $LT(t)$ at address α is that bound variable name.

The relation $\approx$ is the smallest equivalence relation satisfying: $x \approx y$ if

- x is introduced in $BT(t)$ at address αa, index i
- y is introduced $BT(t)$ at address $\alpha' a$, index i, and
- $label(\alpha) \approx label(\alpha')$

Theorem 10. *Let $t : T$ and $u : U$ be terms of isomorphic type. Then $t \simeq u$ if and only if $LT(t)$ and $LT(u)$ are isomorphic labelled trees.*

5 The transformations

Theorem 10 allows us to define transformations for higher-order unification under type isomorphism; as described earlier the key is defining a sound Decomposition transformation.

A *system* $E \rhd \Lambda \mid \sigma$ is given by a set E of equations, a partial mapping Λ on labels, and a substitution σ. Λ and σ denote the label mapping and answer substitutions computed "so far". Transformations are presented as inference rules for deriving systems. If e is an equation, the notation $E;e$ is shorthand for $E \cup e$.

In this short abstract we only present the transformations which are different from the standard ones.

Fail

$$\frac{E \;;\; (\lambda \overline{x}.x_e\, \overrightarrow{t} \doteq \lambda \overline{y}.y_d\, \overrightarrow{u}) \rhd \Lambda \mid \sigma}{\textsf{Fail}}$$

if the type of x_e is not isomorphic to the type of y_d.

We note that using the techniques of [ZGC03], failure can be detected in constant time, after a linear-time preprocessing step on the types of the original terms.

Decomposition

$$\frac{E\ ;\ (\lambda\overline{x}.x_e\,\overrightarrow{t}\ \doteq\ \lambda\overline{y}.y_d\,\overrightarrow{u})\ \rhd\ \Lambda\ \mid\ \sigma}{E\ ;\ (\lambda\overline{x}.t_1\ \doteq\ \lambda\overline{y}.u_{\pi(1)})\ ;\ \ldots;\ (\lambda\overline{x}.t_k\ \doteq\ \lambda\overline{y}.u_{\pi(k)})\ \rhd\ \Lambda_+\ \mid\ \sigma}\quad \text{if } \Lambda_+ \text{ is a valid label mapping}$$

where Λ^a_+ is $\Lambda^a \cup \{x_e \mapsto y_d\}$, and Λ^p_+ is $\Lambda^p \cup \{x_e \mapsto \pi\}$. The condition "$\Lambda_+$ is a valid label mapping" is : if $a \approx b$ and a and b are in the domain of Λ^a_+ then $\Lambda^p_+(a) = \Lambda^p_+(b)$

It is possible that no permutation π allows Λ to be extended by mapping x_e to y_d. In this case the procedure fails at this point. The decomposition can succeed either because Λ itself determines the decomposition π, or because some label $\approx$ with x_e or with y_d was already bound by Λ, or that no label was $\approx$ with either x_e or y_d and so we had freedom to extend Λ by $\{x_e \mapsto y_d\}$ and π.

Theorem 11. *Replacing the traditional Decomposition transformation by the Decomposition and Fail transformations above yields a sound and complete set of transformations for higher-order unification and matching modulo type isomorphism.*

In fact the result of a successful sequence of transformations also yields the finite hereditary permutation witnessing the term-isomorphism between the instantiated terms.

6 Ongoing work

This is a preliminary report, most of the interesting work remains to be done.

Of course we need to incorporate product types into our setting. We do not anticipate any conceptual challenges here, but the algorithmic complexity of working with the types is known to increase due to the fact that products allow more succinct representations of types.

The most important task before us is to derive efficient algorithms to guide the Decomposition transformation defined above. We need to explore the problem of determining when two labelled trees are isomorphic and in particular we require a top-down algorithm (an "online algorithm" in algorithms parlance) in order to be applicable to trees that are being generated during the unification process. Inspired by the results of Zibin, Gil and Considine in [ZGC03] we hope to find algorithms which will ultimately lead to unification and matching procedures which incur only a modest performance penalty for treating the more flexible notion of equality modulo type isomorphism.

It will be important to derive complexity results and to do empirical studies of the performance of our algorithms in cases known to be decidable, such as matching at low orders and unification of patterns.

References

[BCL92] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[DC76] Mariangiola Dezani-Ciancaglini. Characterization of normal forms possesing an inverse in the $\lambda_{\beta\eta}$-calculus. *Theoretical Computer Science*, 2:323–337, 1976.

[DC95] Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[FCB02] Marcelo Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In *Logic in Computer Science*, pages 147–156, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[GS89] J. H. Gallier and W. Snyder. Higher-order unification revisited: complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Plo72] Gordon Plotkin. Building in equational theories. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, volume 7, pages 73–90. Edinburgh University Press, 1972.

[Rit90] M. Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 603–617. Springer, Berlin, Heidelberg, 1990.

[Rit91] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[RT91] C. Runciman and I. Toyne. Retrieving re-usable software components by polymorphic type. *jfp*, 1(2):191–211, 1991.

[ZGC03] Yoav Zibin, Yossi Gil, and Jeffrey Considine. Efficient algorithms for isomorphisms of simple types. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL 2003)*, pages 160–171. ACM Press, 2003.

[ZW93] Amy Moormann Zaremsky and Jeannette M. Wing. Signature matching: a key to reuse. In *First ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 182–190, 1993. ISBN:0-89791-625-5.