RC 21247 (08/04/98)
Computer Science

IBM Research Report

Type Isomorphisms with Recursive Types

Joshua Auerbach, Charles Barton, Mukund Raghavachari

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

In this note, we define a theory of type isomorphisms $Th^1_{\times \boldsymbol{T}_\mu}$ and ask whether it is decidable. The name $Th^1_{\times \boldsymbol{T}_\mu}$ follows the nomenclature of theories of isomorphism in [6]. $Th^1_{\times \boldsymbol{T}_\mu}$ is derived by composing the axioms of $Th^1_{\times \boldsymbol{T}}$ [8, 5, 6] with the axioms of a theory of equality for recursive types $EQ\mu_{\rightarrow\times}$. Individually, the decidability of $Th^1_{\times \boldsymbol{T}}$ [5, 6] and $EQ\mu_{\rightarrow\times}$ [1, 7, 4][1] have been proven. However, axioms in $Th^1_{\times \boldsymbol{T}_\mu}$ interact in new ways that do not permit the individual decidability results to be composed. We do not believe $Th^1_{\times \boldsymbol{T}_\mu}$ has been investigated, though it is mentioned as future work in [6]. In the final section of this note we suggest why the problem is important. In the event that a solution to this problem is identified quickly, we also suggest some extensions that should increase the difficulty.

1 $Th^1_{\times \boldsymbol{T}}$

Consider a type system with the following terms.

$$\tau \equiv \gamma \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2$$

The symbol γ ranges over a finite set of constant types (eg *Int*, *Real*). A distinguished type $\boldsymbol{T} \in \gamma$ is the unit type or terminal type. The $\rightarrow$ denotes function types and $\times$ denotes products.

Now consider type *isomorphisms*. A formal treatment of type isomorphisms can be found in [6]. Informally, two types τ and σ are isomorphic if terms of type τ can be converted to terms of type σ and vice versa, so that the compositions of the conversion from τ to σ with the conversion from σ to τ (in both orders) are the identity mappings. We do not consider semantics in this exposition, and so we just take isomorphism to be defined by the axioms in Figure 1. We discuss the significance of type isomorphisms to our own work in Section 4.

The axioms of isomorphism shown in Figure 1 were first employed by Rittri [8], who demonstrated the utility of using types, under rules of isomorphism, as search keys in retrieving components from a library. Rittri noted that these isomorphisms were those that hold in all Cartesian Closed Categories. This system, called $Th^1_{\times \boldsymbol{T}}$, was also investigated by Bruce, Di Cosmo, and Longo [5], who proved it complete

[1]The decidability of equality for recursive types has been known for a long time, so these recent citations actually deal with the harder problem subtyping of recursive types. Our presentation of $EQ\mu_{\rightarrow\times}$ is drawn from [4].

$$\Gamma \vdash \tau \times \sigma = \sigma \times \tau \quad (\textsc{Com}\times)$$

$$\Gamma \vdash \tau \times (\sigma \times \delta) = (\tau \times \sigma) \times \delta \quad (\textsc{Assoc}\times)$$

$$\Gamma \vdash (\tau \times \sigma) \to \delta = \tau \to \sigma \to \delta \quad (\textsc{Curry})$$

$$\Gamma \vdash \tau \to (\sigma \times \delta) = (\tau \to \sigma) \times (\tau \to \delta) \quad (\textsc{Distrib}\!\to\times)$$

$$\Gamma \vdash \tau \times \boldsymbol{T} = \tau \quad (\textsc{Ident}\times)$$

$$\Gamma \vdash \tau \to \boldsymbol{T} = \boldsymbol{T} \quad (\textsc{Unit})$$

$$\Gamma \vdash \boldsymbol{T} \to \tau = \tau \quad (\textsc{Ident}\!\to)$$

$$\Gamma \vdash \tau = \tau \quad (\textsc{Ref})$$

$$\frac{\Gamma \vdash \tau = \delta \quad \Gamma \vdash \delta = \sigma}{\Gamma \vdash \tau = \sigma} \quad (\textsc{Trans})$$

$$\frac{\Gamma \vdash \tau = \sigma}{\Gamma \vdash \sigma = \tau} \quad (\textsc{Sym})$$

$$\frac{\Gamma \vdash \tau_1 = \sigma_1 \quad \Gamma \vdash \tau_2 = \sigma_2}{\Gamma \vdash \tau_1 \to \tau_2 = \sigma_1 \to \sigma_2} \quad (\textsc{Cong}\!\to)$$

$$\frac{\Gamma \vdash \tau_1 = \sigma_1 \quad \Gamma \vdash \tau_2 = \sigma_2}{\Gamma \vdash \tau_1 \times \tau_2 = \sigma_1 \times \sigma_2} \quad (\textsc{Cong}\times)$$

Figure 1: $Th^1_{\times\boldsymbol{T}}$

for models of the simply typed λ-calculus with pairs and unit. Di Cosmo [6] went on to investigate more complex systems involving second-order types, including one equivalent to the core ML type system. Zaremski and Wing [10] investigated similar rules, along with others, in various combinations, emphasizing flexibility in the choice of rules rather than the completeness of any particular set. Their work was also motivated by the library search problem.

In [5, 6] $Th^1_{\times\boldsymbol{T}}$ is proven decidable as a corollary of the proof of completeness. Although the complete set of axioms is not normalizing when regarded as a set of directional rewriting rules, a critical step in the proof does, in fact, hinge on defining type normal forms. Equivalence is ultimately decided by a finite set of comparisons of normal forms under permutations.

$$\Gamma \vdash \mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha] \quad (\textsc{Unfold/Fold})$$

$$\Gamma, \tau = \sigma, \Gamma' \vdash \tau = \sigma \quad (\textsc{Hyp})$$

$$\Gamma \vdash \tau = \tau \quad (\textsc{Ref})$$

$$\frac{\Gamma \vdash \tau = \delta \quad \Gamma \vdash \delta = \sigma}{\Gamma \vdash \tau = \sigma} \quad (\textsc{Trans})$$

$$\frac{\Gamma \vdash \tau = \sigma}{\Gamma \vdash \sigma = \tau} \quad (\textsc{Sym})$$

$$\frac{\Gamma, \tau_1 \times \tau_2 = \sigma_1 \times \sigma_2 \vdash \tau_1 = \sigma_1 \quad \Gamma, \tau_1 \times \tau_2 = \sigma_1 \times \sigma_2 \vdash \tau_2 = \sigma_2}{\Gamma \vdash \tau_1 \times \tau_2 = \sigma_1 \times \sigma_2} \quad (\textsc{Cross/Fix})$$

$$\frac{\Gamma, \tau_1 \rightarrow \tau_2 = \sigma_1 \rightarrow \sigma_2 \vdash \tau_1 = \sigma_1 \quad \Gamma, \tau_1 \rightarrow \tau_2 = \sigma_1 \rightarrow \sigma_2 \vdash \tau_2 = \sigma_2}{\Gamma \vdash \tau_1 \rightarrow \tau_2 = \sigma_1 \rightarrow \sigma_2} \quad (\textsc{Arrow/Fix})$$

Figure 2: $EQ\mu_{\rightarrow\times}$

Notably absent are recursive types.

2 $EQ\mu_{\rightarrow\times}$

Consider a type system with the following terms.

$$\tau \equiv \gamma \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha.(\tau_1 \rightarrow \tau_2) \mid \mu\alpha.(\tau_1 \times \tau_2)$$

This is the type system of the previous section with μ forms added. The symbol α ranges over type variables. The symbol μ binds its variable in a recursive type definition (α appears free in τ_1 and/or τ_2).[2] We may still have $\boldsymbol{T} \in \gamma$ but $\boldsymbol{T}$ has no special role in the axioms of $EQ\mu_{\rightarrow\times}$.

Axiomatizations exist not only for equality of recursive types but also for subtyping of recursive types [1, 7, 4] and these axiomatizations have been proven decidable. The recent axiomatization by Brandt and Henglein [4], adapted for equality in a type system including $\times$, is shown in Figure 2. These axioms have a simple coinductive

[2] Following the practice in [4], we used $\mu\alpha.(\tau_1 \rightarrow \tau_2) \mid \mu\alpha.(\tau_1 \times \tau_2)$ rather than the more general $\mu\alpha.\tau$ in order to eliminate problematic types like $\mu\alpha.\alpha$. In other formulations, these types are allowed syntactically and dealt with by means of a special *contractivity* condition.

form that makes them easier to work with than the older formulations. But in all the cited work, decidability comes from reasoning that a recursive type and its unfoldings (foldings) are the same regular tree. From properties of regular trees, it is then possible to justify that type equivalence be decided from finite truncations of those trees.

Notably absent are any axioms of isomorphism.

3 $Th^1_{\times}\boldsymbol{T}_{\mu}$

We now define $Th^1_{\times}\boldsymbol{T}_{\mu}$ as the theory whose whose forms are those of $EQ\mu_{\rightarrow\times}$ and whose axioms are those of $Th^1_{\times}\boldsymbol{T}$ and $EQ\mu_{\rightarrow\times}$ combined.

Note that rules REF, TRANS and SYM are common to both sets. The rules CONG$\times$ and CONG$\rightarrow$ in $Th^1_{\times}\boldsymbol{T}$ are redundant in $Th^1_{\times}\boldsymbol{T}_{\mu}$ in that they can be proven from rules HYP, ARROW/FIX, and CROSS/FIX. This should not be a problem (one can think of them as theorems if desired).

We now wish to ask: *Is the composed system consistent and decidable*?

To suggest where the difficulties may lie, let's consider an example. The following type equation can be proven in $Th^1_{\times}\boldsymbol{T}_{\mu}$.

$$\begin{aligned}&\mu\alpha.(Int \rightarrow (Bool \times (Real \rightarrow (Bool \times \alpha)))) = \\ &\quad (Int \rightarrow Bool) \times (Int \rightarrow (\mu\beta.(Real \rightarrow (Bool \times (Int \rightarrow (\beta \times Bool))))))\end{aligned}$$

In proving the above, we had to apply rule UNFOLD/FOLD (from $EQ\mu_{\rightarrow\times}$) followed by rule DISTRIB$\rightarrow\times$ (from $Th^1_{\times}\boldsymbol{T}$) in order to establish even as much as the following.

$$\begin{aligned}&\mu\alpha.(Int \rightarrow (Bool \times (Real \rightarrow (Bool \times \alpha)))) = \\ &\quad (Int \rightarrow Bool) \times (Int \rightarrow Real \rightarrow (Bool \times \mu\alpha(Int \rightarrow (Bool \times (Real \rightarrow (Bool \times \alpha))))))\end{aligned}$$

Having gotten that far, we still need the full power of $EQ\mu_{\rightarrow\times}$ augmented with rule COM$\times$ to finish the job, because no amount of manual unfolding of the two types will produce equal terms (see [1] for simpler examples of the same phenomenon).

Proofs like this one in $Th^1_{\times}\boldsymbol{T}_{\mu}$ allow the axioms to interact freely regardless of which theory they came from. With rule UNFOLD/FOLD included, the type normal forms used to prove $Th^1_{\times}\boldsymbol{T}$ decidable cannot be derived. With the rules of

$Th^1_{\times\boldsymbol{T}}$ included, it is no longer true that the types isomorphic to a given type are the same regular tree, so the justification that allowed finite truncations to be used in the decidability of $EQ\mu_{\rightarrow\times}$ is undermined. Thus, the decidablity of $Th^1_{\times\boldsymbol{T}_\mu}$ doesn't follow from the separate results.

4 Motivations

Much of the previous work on type isomorphisms [8, 5, 6, 10] is motivated by the retrieval problem: component signatures, under type isomorphisms, are used to find relevant components in libraries. When the retrieved signature differs from the requested one, the programmer must write the invoking code accordingly or else must change the retrieved code. This is a minor burden when the retrieved components are written in the language that the programmer is using, especially when the type isomorphisms are relatively simple.

Our own concern is with the composition of components from multiple programming languages, particularly popular imperative languages like C, C++ and Java. These languages make heavy use of non-abstract types. Efficient programming often requires that arrays and the public fields of objects be exposed. Information is often expressed structurally via patterns of inter-object references that require recursive type definitions. The type structure chosen for a given problem will tend to differ by language as well as by programming style. It therefore requires richer type isomorphisms to express useful similarities between components.

Thus, in the multilanguage case, the task of translating between isomorphic types can easily become more time-consuming than the task of finding the right component in the first place. The resulting "bridge code" may require knowledge of two languages, and it will have to be manually maintained. For this reason, we want to generate the bridge code automatically. We believe this can be done if the rules of isomorphism have a straightforward operational interpretation (as do the axioms of [4]). However, the first task is to formulate the rules themselves.

The PolySPIN project [3] has already moved from retrieval to actual generation of bridge code, using an isomorphism framework similar to Zaremski and Wing. This represents significant progress. But, PolySPIN carries forward the assumption that interesting types are abstract types like objects, whose type properties are completely captured in the signatures of methods. They do relatively little with non-abstract types such as records, linked lists, arrays, etc.

In the Mockingbird project [2],[3] abstract types are dealt with as in [3], by matching method signatures and generating bridge code accordingly. However, our treatment of method parameters more fully exploits isomorphisms between non-abstract types, and the generated code covers these isomorphisms as well. Mockingbird supports C++, Java, and CORBA IDL, and does both local and remote composition. Using *ad hoc* rules and some hints from the programmer, Mockingbird finds the following Java and C signatures to be equivalent and generates the bridge code. The resulting composition of components executes correctly, both locally, and when distributed across machines.

```
/* Java */
public interface LineFitter {
  Line fit(Vector points);
}
/* C */
typedef void lineFitter(
   float * x_coords, /* really an array */
   float * y_coords, /* really an array */
   int length,       /* of x_coords and y_coords */
   double * x1_output,
   double * y1_output,
   double * x2_output,
   double * y2_output);
```

We are not claiming that structure-based isomorphisms cover all possible issues concerning non-abstract types. We recognize that such types can represent implicit abstractions (eg., pairs of *Real* represent points). Wing and Ockerbloom [9] show how to define value conversions that preserve implicit abstractions (eg., conversion between Cartesian and polar representations of points), but these have to be manually written. Automated structure-based isomorphisms *complement* manual abstraction-preserving conversions. When told that some parts of a value represent an abstraction, Mockingbird uses library- or programmer-supplied conversions for those parts. Mockingbird then overcomes any remaining structural differences. As in any signature-based approach, it is up to the programmer to combine components in semantically appropriate ways. Mockingbird's role is to avoid tedious and error prone work.

[3]See also http://www.research.ibm.com/internettech/mocking.html.

Although Mockingbird is able to solve real problems, its rules of equivalence are not exactly those discussed in the earlier sections and don't correspond to any formal theory. In order to put the algorithms underlying Mockingbird on a truly sound basis, we need to work with a system of type isomorphism that, at a minimum, includes recursive types.

Our interest in these matters will not end with a solution to just the problem of Section 3. To express a "list of pairs of real numbers" we need a sum (+) operator as in $RealPairList = \mu\alpha.(\boldsymbol{T} + (Real \times Real \times \alpha))$. To solve larger problems in Java and C++ we will require subtypes and dynamic types. If it turns out to be easier than we thought to prove results about $Th^1_{\times \boldsymbol{T}_\mu}$, we would then like to know about extensions involving sums and subtypes.

References

[1] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993.

[2] J. Auerbach and M. C. Chu-Carroll. The Mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Technical Report RC20718, IBM T. J. Watson Research Center, 1997.

[3] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Fourth Symposium on the Foundations of Software Engineering (FOSE)*, October 1996.

[4] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, April 1997.

[5] K. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[6] R. Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design.* Birkhauser, 1995.

[7] D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 419–428. ACM, January 1993.

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

[9] J. M. Wing and J. Ockerbloom. Respectful type converters. Technical Report CMU-CS-98-130, Carnegie Mellon University, 1998. Submitted to IEEE Transactions on Software Engineering, May 1998.

[10] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering Methodology (TOSEM)*, April 1995.