Multilanguage Programming: An Automatic–Type–Mapping Approach*

Arturo J. Sánchez-Ruíz† **Ephraim P. Glinert**
Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180

Abstract

The "data type correspondence problem" arises when one tries to interconnect, within a single program, modules written in different languages and therefore within the framework of different type systems. We propose an interconnection model which hides details such as interlanguage data type mappings and low level communications, which are irrelevant or confusing to most programmers. Using this model, we develop an "automated" solution to the problem of finding, for a given type in a given programming language, the corresponding type in another programming language. We compare our approach with others which have been proposed in the literature, and discuss a prototype implementation in detail.

1 Introduction

Software reusability, whether of one's own programs or those written by others, is a main objective of modern software engineering. If we want to use existing things, we cannot impose strong restrictions on the way things are created. Ideally, we should have no say regarding the choice of programming language used to code a (sub)program; neither should we be able to dictate the machine on which a (sub)program executes. The term multilanguage programming (MLPG) refers to a situation where modules within a single program are written in different languages. MLPG and machine heterogeneity thus arise naturally when one wants to put reusability into practice.

The work described in this paper is part of a long–term project whose ultimate goal is to develop a powerful distributed, multilanguage environment for scientists which will support high performance numeric computation, scientific visualization, virtual realities and more. An immediate and pressing need is to provide a suite of tools to automate the interlanguage aspects of multilanguage programming. That the tools be automatic insofar as possible is important, because we want to liberate our target users—scientists who are not computer scientists—from chores which they perceive as irrelevant to their research.

All programmers have to build a mental map between the (abstract) objects natural to an application and the (concrete) type system offered by the language at hand. Our experience indicates that scientists who are not computer scientists commonly use what might be called a *data–structure–oriented methodology* to map mental objects directly into the data structures the language provides. This is as opposed to an *object–oriented methodology* such as computer scientists might prefer, in which the programmer first specifies the objects relevant to the computation and then finds appropriate representations for these objects in terms of the available data structures.

Therefore, although a type system that supports abstract data types and objects may be appropriate for computer scientists, it isn't very useful for other kinds of scientists. Computer scientists don't approach programming in the same way that other scientists do![‡] If we want out target users to prefer our environment, we must enable them to take full advantage of the available features while continuing to program in the paradigm(s) with which they are familiar.

2 Definitions and Scenarios

Denote a multilanguage program P by:

$$P = m_1 \bigoplus m_2 \bigoplus \cdots \bigoplus m_n$$

where each m_i is a module consisting of program units (procedures, functions, subroutines) all written in the same programming language L_i:

*The work reported here comprises part of the first author's dissertation (in progress). This research was supported, in part, by the National Science Foundation under contract CDA–8805910 and contract CDA–9015249. Authors' e–mail addresses: *sancheza@cs.rpi.edu*, *glinert@cs.rpi.edu*.

†On leave from Universidad Central de Venezuela, Caracas.

‡Cf. [1] for an interesting discussion of this topic in the context of parallel programming.

$$m_i = f_{i_1} + f_{i_2} + \cdots + f_{i_{k_i}}$$

We consider three possible scenarios as descriptions of the manner in which P was implemented:

- *Scenario 1*: Every m_i is built from scratch. We don't reuse existing modules.
- *Scenario 2*: We want to create some m_i from scratch, by reusing a set of existing modules $\{m_j^i\}$.
- *Scenario 3*: A module m_i has already been partially written and we want to take advantage of the existence of some modules $\{m_j^i\}$ in completing it.

The problem of interconnecting the set of modules $\{m_k\}$, that is to say, the problem of giving meaning to $\bigoplus$, is different for each scenario.

In Scenario 1, we can take advantage of the fact that nothing pre-exists to choose the data structures so that everything fits together nicely. For example, if Fortran and C are being used we can limit ourselves to working with arrays in both languages, without exploiting the advantages of other structures like records or lists. In this case, certain problems related to reusability clearly don't arise.

Scenario 2 poses a more challenging problem. If a certain existing module m_j is going to be used by the module m_i, the module m_i has to be able to "understand" the data types m_j manipulates. For example, consider the creation of a Fortran program by reusing a C function which manipulates linked lists. Here we must deal with two problems:

1. *The data type correspondence problem*: How to represent in a given language data types which are not defined in it.
2. *The data type checking problem*: How to ensure that the usage of an existing module agrees with its definition.

In Scenario 3 we have to be able to represent "foreign" types in the implementation language, as in Scenario 2, but with added restrictions imposed by the existing data structures in the already partially written m_i. For instance, we might have a Fortran program which manipulates matrices (represented as arrays) and that calls a Lisp routine which manipulates polynomials (represented as lists), where all of the information needed to build the polynomials is stored in the matrices.

With these three scenarios in mind, consider two languages L_1 and L_2 with type systems T_1 and T_2, respectively. Given a type t_i in T_1, there are two possibilities:

- There is a type t_j in T_2 such that t_i and t_j are isomorphic.
- There is no such t_j.

The data type correspondence problem can now be defined as follows:

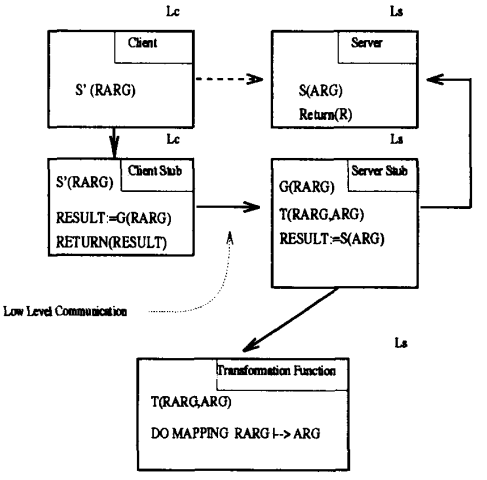

Figure 1: The MLPG Interconnection Model.

How do we represent t_i in L_2 if there is no isomorphic type associated with it?

The term isomorphic is used here to mean "essentially the same." The best way to define the isomorphic types for two given languages is by means of a *dictionary of types* which allows one to identify, for a given type t_i in language L_1, the corresponding type t_j in L_2. An example of such a dictionary, for Fortran and C would have `REAL` correspond to `float`, `INTEGER` to `int` and `DIMENSION` to array. It is unclear what single Fortran type translates to C structures, so an instance of the type correspondence problem is to find a way of representing C structures in Fortran.

The experience we have gained through working with scientists in several disciplines indicates, that Scenario 2 is an attractive one for the creation of new software by "gluing" together existing "parts" (a technique which has been called *post facto integration* [2]). With this scenario in mind, our support software will solve the data type correspondence problem by automatically computing the representatives for a type not available in a particular language. As a by-product, it will also solve the type checking problem.

How do other multilanguage systems approach the data type correspondence problem? Not surprisingly, as far as can be determined from the published literature, many of them appear to address it from the point of view of Scenario 1, employing type systems which are supersets of all the type systems present in the supported programming languages. Since everything is created from scratch, programmers have the prerogative of choosing their data types so that the type correspondence and checking problems aren't an issue. System support for the programmer is limited to declaration generation from the type specifica-

```
<type definition> ::=
  <basic type> | <structured type>
<basic type> ::=
  R | I | B | C | D
<structured type> ::=
  [ <type definition> ] |
  ( <type definition> ) |
  { <fields> }
<fields> ::=
  <identifier> : <type definition> |
  <identifier> : <type definition> ; <fields>
```

Figure 2: STS Syntax.

STS Type	Informal Meaning
R	Real
D	Double
I	Integer
B	Boolean
C	Character
$[\cdots]$	Array of $(\cdots)$
$(\cdots)$	List of $(\cdots)$
$\{f_1 : T_1; \cdots; f_n : T_n\}$	Record of $(f_1 : T_1; \cdots; f_n : T_n)$

Figure 3: STS Semantics.

tions. Clearly, however, Scenario 1 is the one we *shouldn't* have in mind if we want to promote reusability!

UTS, the type system of Hayes *et al.*'s MLP [3, 4], which is designed for Scenario 2, introduces the concept of type representative. This type is the one associated with objects which are not directly representable in a given language. So, if a record is needed in a Fortran program, say, the programmer needs to first declare the object as representative using the UTS notation, and then manipulate the object (from Fortran) by using a set of functions provided by UTS to create objects, select components of them, and modify them. Since the programmer has control over the type, the system has to worry about type checking to insure data integrity.

Purtilo *et al.*'s Nimble [5] extension to Polylith [6], and Wileden *et al.*'s SLI [7], are both aimed at Scenario 3. Early versions of Polylith treated the problem of finding a representative for a nonexisting type as an error. Although Nimble is a clear step forward, the user has to explicitly specify the mapping between actual and formal parameter, and the system still has to deal with type checking issues. The reported SLI prototype is unable to solve the type correspondence problem; the user has to provide so–called "underlying implementations" for the abstract data types used.

3 An Automatic Approach

Our approach to multilanguage programming consists of the following elements:

- An *interconnection model* to realize the interconnection between modules.
- A *type system* that abstracts useful properties present in type systems of "popular" languages such as C, Pascal, Fortran and Lisp.
- An *interconnection specification language* to describe how modules interconnect.
- A *translator* to produce interconnection model elements from specifications.

Consider a program unit S (the *server*) written in language L_s and with argument **ARG**. Suppose this program unit is needed by some other program unit C (the *client*) which is going to be written in language L_c. We are interested in the problem where the type of **ARG** is not directly representable in L_c and the client and server reside on different machines. In order to establish a connection between the client and its server, we propose the following elements and procedure:

- *An automatic type mapper* (ATM) which solves for the user the problem of finding in L_c a representative **RARG** for **ARG**.
- *A client stub* which acts as if it were the server (from the client's point of view). This means taking **RARG** and making the connection with the server (stub).
- *A server stub* which takes the representative **RARG** and transforms it back into **ARG**, calls the server, and returns the results to the client (stub).
- *A type transformation function* which transforms **RARG** into **ARG**.

All these elements are put together in Fig. 1, in which the phrase "low level communication" refers to the use of communication primitives to transmit typed information from one stub to another. These primitives either are provided by the operating system or the transport layer of the network, or they can be built from available primitives.

The programmer specifies the interconnection between two program units by using a specification language that provides expression for our "*simple type system*" (STS) whose syntax is shown in Fig. 2; an informal explanation of the semantics is given in Fig. 3. An example of an interconnection specification (to which we will return later) is given in Fig. 4. For the moment, it is sufficient for the reader to note the simple form of the specification.

The interconnection specification is the input to the *automatic type mapper* (ATM), whose function is to produce a *client template* (which the user completes to build the client), the *client stub*, the *server stub*, and the *transformation function*. Note that the last three of these elements

```
SERVER NAME:        EvalPoly
SERVER LANGUAGE:    C
CLIENT LANGUAGE:    FORTRAN
INPUT PARAMETERS:   p :  ({exp:R; coeff:R})
                    x :  R
OUTPUT RESULT:      D
```

Figure 4: Example Interconnection Specification.

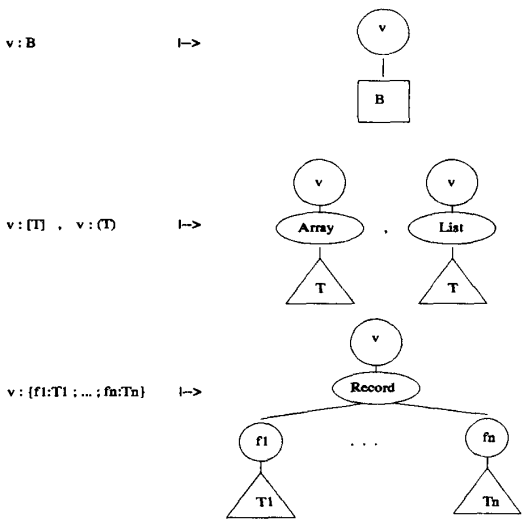

Figure 5: Syntax Tree Construction.

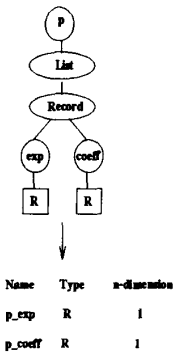

Figure 6: Syntax Tree for p in Fig. 5.

have already been discussed within the context of the interconnection model.

4 Implementation

To gain further insight into the practicality of the ideas discussed above, we applied them to the following simple problem:

- Clients are written in Fortran.
- Servers are written in C.
- Server functions return scalar values but may have structured arguments.
- The static structures used to represent dynamic structures are of fixed size.
- All modules reside on the same machine, so we can ignore low level communications.

We take advantage of the fact that Sun Microsystems `cc` and `f77` compilers produce code which can be linked together by the `ld` linker. Therefore the linker implements the communication platform mentioned above. The prototype we built has three main modules: a module to parse the specification, a module to compute the way each non-Fortran type is represented in Fortran, and a module to generate the code to establish the interconnection. We treat each of them in the following sections.

4.1 Parsing the Specification

This module receives as input the file containing the interconnection specification through the lexical analyzer (`yylex`) which was built by using the UNIX `lex`. It generates a list containing all input arguments and the corresponding syntax tree associated with its STS type expression, as follows (cf. Fig. 5):

- The syntax tree associated with a:

  `<basic type>`

  is a terminal node labeled with a character which identifies the type.
- The syntax tree associated with an expression of the form:

  `[<type definition>]`

  is a tree having as its root a node labeled with a character that identifies the type constructor *array* and with one child which is the syntax tree associated with the expression `<type definition>`. The same applies to an expression of the form:

  `(<type definition>)`

  for the type constructor *list*.
- Finally, the syntax tree of an expression of the form:

$$\{f_1 : T_1; \cdots; f_n : T_n\}$$

where each T_i is a <type definition>, is a tree having as its root a node labeled with a character which identifies the type constructor *record* and with n children which are the syntax trees associated with the expressions $T_1, \cdots, T_n$ (in that order).

4.2 Computing the Representatives

Given an argument with its STS syntax tree, this module finds a way of representing it assuming that only arrays and basic types are available in the client type system (which is in fact the case for Fortran). In this way, a list gets represented by an array of fixed size and a record gets represented by as many objects as there are fields in the record. Essentially, this is done by flattening the structure of the syntax tree, so in general the representative of an argument will be a set of n-dimensional arrays with $n \geq 0$ (where, by definition, a 0–dimensional array is a single variable). The types of these arrays are of course basic types. More precisely, if v is an object with representatives $rep(v)$ and r is a member of $rep(v)$, then r is an n–dimensional array named $v\_f_{i_1}\_\cdots\_f_{i_k}$, and:

- n is the number of *array* or *list* nodes in the path, associated with r, from the root to the corresponding terminal node.
- k is the number of *record* nodes in the same path.

This is implemented by traversing the tree in preorder (first visit the root, then traverse its children in preorder from left to right). An example of representative computation for the variable p in Fig. 4 can be seen in Fig. 6.

4.3 Code Generation

Our ATM has to generate four program units: a client template, the stubs and a transformation function. What follows is a description of how each of these is produced. We will then look at an example of what is actually generated from a specific interconnection specification.

Generating the Template. In the case we are considering, the program has to generate a Fortran template. Since the template deals with representatives, it can be easily produced from the list of arguments and their representatives, which is the input to this module. The template has placeholders to define the *maximum* array sizes. Also, the user has to define the actual array sizes.

Generating the Stubs. The client stub collects all the representatives and passes them to the server stub. It therefore need only contain declarations for all the representatives and an appropriate calling sequence to the server stub. The server stub is generated from the list of arguments with their corresponding syntax trees and representatives, from which declarations of representatives and arguments in the server's language can be deduced, as well as the sequence of calls to the transformation function and to the server. Generation of declarations for the representatives is carried out directly from their specification. Generation of the declarations for the arguments is done by traversing the syntax tree and translating every type constructor to the corresponding syntactic element (in C, for the case we are considering). Because the *list* type constructor is not quite a type in C, we decided to represent it as a linked list with two fields, one called *info* to contain the "information" and the other called *next* to hold the pointer to the next element. The rest of the server (sequence of calls to the transformation function and to the server) is generated easily from the information provided as input.

Generating the Transformation Function. To build the transformation function we need, for each argument, its syntax tree and its representatives. Since the representatives of an argument is a set of n–dimensional arrays, the transformation function has a well defined structure, namely that of a set of nested loops, where each loop contains the statements to build each argument component from the corresponding representative. Thus, we first create a data structure which is an image of the set of nested loops, and then we traverse this data structure to actually produce them. The loop structure is built by traversing each syntax tree (in preorder) and by translating each selector operator to its C counterpart.

5 An Example

Let `EvalPoly(p,x)` and `EvalDeriv(p,x)` be two C routines for evaluating a polynomial with real coefficients:

$$P(x) = \sum_{i=1}^{n} coeff_i * x^{exp_i}$$

and its first derivative, respectively, at point x (`EvalPoly.c` is shown in Fig. 7). Suppose that in the given routines P is represented as a linked list, where each node contains an ordered pair:

$$(exp_i, coeff_i)$$

Our goal is to employ these routines to implement the well known Newton–Raphson method for finding a root of P given an initial estimate of its location. Our multilanguage program will call the C routines from a Fortran driver.

The prototype ATM we implemented helps the user find out which data structures are needed to call the C functions. In order to do this, the user provides the ATM with two specification files, namely `EvalPoly.spec` and `EvalDeriv.spec`, which contain the specifications of `EvalPoly.c` and `EvalDeriv.c`, respectively (the two files are very similar in this case; `EvalPoly.spec` was previously shown in Fig. 4). From each `.spec` file, the ATM produces a template file (one of these, `EvalPoly_tmplt.f`, is shown in Fig. 8) as well as the corresponding stubs and auxiliary files needed to establish the interconnection (not

```
#include <stdio.h>
typedef struct node { float expo;
                      float coeff;
                      struct node *next;
                    } poly ;

double EvalPoly (p,x)   poly *p;   float x;
{
   #define power(A,B) (((B) == 0.0)?
                      (double) 1.0 : pow(A,B))

   double pow();   poly *head;   double sum;

   head = p;   sum = 0.0;
   while (head != NULL)
   {
      sum += ((double)head->coeff *
              power((double)x,
                    (double)head->expo));
      head = head->next;
   }
   return (sum);
}
```

Figure 7: File `EvalPoly.c`.

```
C   Declaration of representatives - where a "#"
C   appears, the corresponding actual dimension
C   bound should be inserted in its place
C
C   Representatives for p:
C
      REAL p_coeff_ (#1)
      REAL p_exp_ (#1)
C
C   Representatives for x:
C
      REAL x_
C
C   Declaration of returning value and function:
C
      DOUBLE PRECISION result_EvalPoly
      DOUBLE PRECISION EvalPoly
C
C   Declaration of dimension size for arrays:
C
      INTEGER dim_
C
C   Sequence of call from client to server:
C
      result_EvalPoly =
     +     EvalPoly (p_coeff_, p_exp_, x_, dim_)
```

Figure 8: File `EvalPoly_tmplt.f`.

shown here due to lack of space). Using the templates, the user creates the file **Newton.f** (also not shown due to lack of space) and then invokes the script:

```
glue Newton EvalPoly EvalDeriv
```

to produce the executable **Newton**. This script uses `f77`, `cc` and `ld` to generate the interconnection between the client and the servers with the aid of the stubs.

Although our example is a relatively simple one, it would be much more difficult for the user to implement in competing systems. This is because the user would have to either provide the serialization/deserialization routines or to tell the system how to build them. Our idea is to liberate her/him from these chores as much as possible.

6 Summary

In this paper we have proposed an "automatic" solution to the MLPG data type correspondence problem for a programming scenario in which program construction is conceived as the creation of a collage assembled from parts prepared by others ("Scenario 2"). Our approach is based on an interconnection model which hides details not relevant to the user. In particular, type checking, communication platform and code generation are totally transparent. This promotes software reusability, thereby helping to reduce both development time and cost. We plan to continue this research, with the ultimate goal of developing a suite of automatic tools to assist in the construction of distributed, multilanguage programs by scientists who are not computer scientists.

References

[1] C. M. Pancake and D. Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers? IEEE Computer, 23(12):13–23, December 1990.

[2] L. R. Power. Post–Facto Integration Technology: New Discipline for an Old Practice. In Proc. IEEE Int. Conf. on Systems Integration (ICSI'90), Morristown, pages 4–13, 1990.

[3] R. Hayes and R. D. Schlichting. Facilitating Mixed Language Programming in Distributed Systems. IEEE Trans. on Software Engineering, SE–13(12):1254–1264, December 1987.

[4] R. L. Hayes. UTS: A Type System for Facilitating Data Communication. PhD thesis, Dept. of Computer Science, University of Arizona, Tucson (Technical Report 89–16), 1989.

[5] J. M. Purtilo and J. M. Atlee. Improving Module Reuse by Interface Adaptation. In Proc. IEEE Int. Conf. on Computer Languages, New Orleans, pages 208–217, 1990.

[6] J. M. Purtilo. An Environment for Prototyping Distributed Applications. In Proc. 9th IEEE Int. Conf. on Distributed Computing Systems, Newport Beach, pages 588–594, 1989.

[7] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt and P. L. Tarr. Specification Level Interoperability. Comm. of the ACM, 34(5):72–87, May 1991.