A Type-Based Implementation of a Language With Distributed Scope

Dominic Duggan[1] *and Piotr Przybylski*[2]

1 Department of Computer and Engineering Science, Case Western Reserve University, Cleveland, Ohio 44106, USA. E-mail: dominic@alpha.ces.cwru.edu.

2 Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

The ML programming language has several features that make it desirable as a language for distributed application programming. It supports first-class closures, which are useful for distributed scope and mobile agents. Type inference removes much of the tedium of lower-level programming, without compromising reliability. Finally ML provides a powerful type system, including type polymorphism. A dialect of ML intended for distributed application programming is introduced. A distinguishing characteristic of this implementation is the use of run-time type information, motivated by several constructs in the language. This is intended to motivate the use of run-time types in implementations of polymorphic languages intended for distributed programming.

1 Introduction

Our starting point for the current work was to consider ML [60] as a language for writing distributed applications, in a similar manner to languages such as Tcl [71], Python [86], Obliq [14], Telescript [91] and Java [33]. There are several reasons motivating taking this approach. In particular ML combines higher-order features (particularly first-class closures) with a well-developed and well-understood type system based on implicit typing and type inference. The former fact means that the facilities for remote invocation and distributed scope pioneered by Obliq are realizable without any change to the underlying language semantics. The latter fact makes the language particularly suitable for application programming, since much of the tedium of programming is removed by a sophisticated type-checker, without compromising run-time reliability. Finally ML is by now a reasonably familiar programming language, so there is already a pool of experienced ML users to draw upon.

In this paper we describe an implementation of a distributed dialect of ML that addresses these issues. In Section 2 we describe some of the modifications we have made to the ML semantics as part of this work. In Section 3 we consider a type-based implementation of this ML dialect. Finally Section 4 provides our conclusions.

2 Language Changes

We call our particular dialect of ML (with some apologies for the pun) ML3000. ML3000 includes some changes to the usual Standard ML semantics in order to support distributed programming, which we now summarize. The main point about these changes is the motivation they provide for our type-based implementation.

2.1 Dynamic Typing

Although static typing is clearly useful for reliable and efficient code, dynamic typing is also necessary for distributed programming. The need arises for several reasons, chiefly the fact that two programs may be compiled completely separately and only come in contact via network communication at run-time. Some use of type information must be used at that point to check the well-typedness of the communication. Following the experience with Modula-3 Network Objects [10], we take the position that the well-typedness of the communication should be determined using purely local information available to the two programs, and without the use of any kind of global protocols or "type servers." This approach appears crucial for scalability.

Dynamics constitute a mechanism for adding dynamic typing to statically typed programming languages [2]. Some evidence for the usefulness of dynamics in distributed programming is provided by Krumvieda's implementation of Distributed ML [49]. The approach of dynamics essentially involves the addition of an operation, dynamic, for bundling a value with its type, and also the addition of a typecase construct, for examining a type bundled in a dynamic. However there is a problem with using dynamics in polymorphic languages, and that is its interaction with parameterized types. Consider for example defining a print server that can print lists (bundled as dynamics):

```
fun print (x: dynamic) = typecase x of
    (α list) (ys) => for each y in ys, do | print(dynamic(y)) |
```

The problem is in how to recurse over the elements of the data structure. In the approach suggested by Leroy et al [53][54], the elements of the data structure are rebundled as dynamics, as shown above.

Part of the distribution support in our language is a new construct for computing with dynamics which allows pickled data structures such as lists and trees to be unbundled once and then traversed,. In our approach, rather than recursing over a data structure checking tags associated with data items, an operation such as `print` instead recurses over the *type* of the data structure. The actual language constructs are described in [24]; the point of this example is to demonstrate the usefulness of runtime type information for distributed programming.

2.2 User-Definable Marshalling

We think of dynamic as the basis for marshalling primitives for a programming language, analogous to the use of the Pickle module in Modula-3 (which in turn is used quite heavily in the implementation of Network Objects [10]). The Pickle module allows the programmer to register user-defined type-specific marshalling and unmarshalling operations (called *specials*). Specials are dispatched on the basis of run-time type tags associated with data structures. Some motivation for this is provided by Birrell et al [10].

An obvious question is, how can we provide a similar facility in a language with type polymorphism? Our language is designed to provide mechanisms for user-definable marshalling [23]. Marshalling and unmarshalling of data structures consists of functions that recurse over both the data structure and the *type* of the data structure. Instances of the marshalling and unmarshalling operations for a specific type may be registered by the programmer. Where the type is some collection type, these operations must be able to dispatch to hypothetical marshalling operations for the undetermined element type, using language mechanisms described in [23].patching is essentially a recursive call to the fixed point of a global marshalling operation, comprised of the combination of the clauses provided by the programmer.

Herlihy and Liskov [38] describe another approach to user-definable marshalling in polymorphic languages, in the CLU language. Their approach is based on *constrained genericity*: a marshalling operation for a bag ADT, for example, is parameterized by a marshalling operation for the element type of the bag. This is similar to type classes in Haskell [35], and as such this approach suffers from all of the problems associated with type classes [72].

The point of this description is to motivate the use of run-time type information in our implementation. To date, most ML implementations strip away type information at compile time [4], and a tagged run-time is used for garbage collection and marshalling/unmarshalling [70]. We briefly cite some other reasons for why run-time type information (RTTI) is desirable in a distributed ML implementation: (1) RTTI can be used as a basis for untagged garbage collection in polymorphic languages [83]. (2) RTTI can also be used as the basis for specializing data representations and code in polymorphic languages [36][81]. (3) Finally run-time type information can also be used to extend the Java idea of load-time bytecode verification to polymorphic languages. This cannot be done in Rouaix' MMM Web browser (based on Caml Light) because the bytecode is untyped; Rouaix instead relies on trusted compile servers for security [76].

All of this is motivation for RTTI, and in particular for run-time type parameters to polymorphic functions, in our implementation. ML has the particular property (similarly to Ada and C++) that polymorphic functions are second-class. Therefore it might be assumed that run-time types are not necessary: instead polymorphic functions can be specialized at their use-sites to their monomorphic instances. This is what is done in Ada and C++ implementations. However there are some objections to this approach: (1) Experience with Ada for example suggests that this leads to a significant growth in code size. The size of the code is a point that must be considered for programs that are meant to fit inside telephones. (2) If polymorphic library functions are down-loaded across the network, it is unreasonable to expect the loader to be able to specialize the code to the types at its use sites. The approach of monomorphic specialization is not compatible with separate compilation of polymorphic functions. (3) Finally, monomorphic specialization is not possible with first-class polymorphism. Our dialect combines first-class polymorphism and decidable type inference, through a novel form of primitive objects with polymorphic methods.

2.3 Objects

Our ML dialect includes built-in objects (rather than attempting to represent objects as closures, as is more commonly done). Our reason for doing this is to provide container objects with polymorphic methods [24]. For example, it is possible in our type system to define objects with interfaces of the form:

$$\alpha \text{ List} = \{ \text{car}: \alpha, \text{cdr}: \alpha \text{ List}, \text{map}: (\alpha \rightarrow \beta) \rightarrow \beta \text{ List} \}$$

Since our ML dialect already contains built-in objects, it is natural to use them as an abstraction for network connections, location transparency and third-party transfer, as is done in distributed object systems. As with Obliq, we allow all objects to be potentially network objects. A true object is transmitted as a network reference, that is unmarshalled at the receiving node of a communication as a proxy calling back to the true object.

Our object system shares some common foundations with Obliq objects, with their foundations in the primitive object calculus of Abadi and Cardelli [1]. Because of this relationship to Abadi and Cardelli objects, it is possible to add some of the features of Obliq to our language without difficulty. For example, object migration in Obliq is provided using object cloning with a network address, and method aliasing, and we believe these can be added straightforwardly to our system.

2.4 Type Equivalence

Datatypes play a particularly crucial rôle in transmitting values in a type-safe way between address spaces. Type tags are transmitted with values to facilitate unmarshalling and type-checking at the receiving site, as described in the next section. We assume that datatype equivalence is structural rather than name-based. Modula-3 also assumes structural equivalence of recursive types, but allows name equivalence with *branded types*. In distributed programming, branded types are required to be given the same user-defined brands in all address spaces. We obtain a simple form of user-defined brands by including the name of a datatype in the identity of that type. Type-checking hashes from a datatype definition into a *type table* to obtain an integer typecode for that datatype. These typecodes are the basis for RTTI in programs, with the type table saved by the compiler and reloaded at run-time, and indexed by the typecodes in the code.

3 Implementation

Our implementation of our extended ML dialect comprises a compiler, written in SML/NJ, and a bytecode interpreter. The compiler comprises a new frontend and type-checker (a frontend with subtype inference is in development), and a backend based on the Moscow ML compiler. The backend is modified to provide code for run-time types, among other things. The interpreter is based on the bytecode interpreter for the CAML Light language [52], modified again to support run-time types.

3.1 Compiler

The type-checker for the compiler incorporates several novel features. Currently dynamics and user-definable marshalling are not fully implemented. However the frontend does gather all of the information necessary to generate code for run-time types. In particular every let-defined variable records the type variables abstracted over in that polymorphic definition, and every use site for such a variable records the instantiations for those parameters. The compiler also supports objects with polymorphic methods and object cloning, as well as some other features (e.g. higher-order polymorphism).

Our treatment of datatypes deserves some mention, since they are at the core of our treatment of run-time types. As already noted, equivalence of datatypes is based on structural type equivalence. When a datatype is defined in the program, the compiler hashes into a table of type descriptions, the *type table*. Once an entry has been located for a datatype, the datatype is referenced by a *typecode*, essentially the index of that datatype declaration in the type table.

A similar approach is used for records and primitive types. Finally a type table entry may be the global signature for a method. First-class polymorphism is achieved in our language by requiring that method types be declared globally [24]. Each such declaration adds a corresponding entry for the method type to the type table. So an object type can be represented as a sequence of (method name, typecode) pairs. This is something of an oversimplification, since method types have much more structure than we have described here.

Datatype and record entries in the type table also contain a *fingerprint*, a 64-bit checksum that with high probability is unique for that datatype or record. Fingerprints are constructed as part of the process of hashing into the type table. Their real usefulness is at run-time, in transmitting typecodes between address spaces. A typecode is transmitted across the network by sending its fingerprint. On receipt of a fingerprint at the receiving site, this is used to hash into the type table for the corresponding typecode. This is essentially the approach taken to transmitting types in Modula-3 Network Objects [10]. The complication we must deal with is polymorphism: a type is transmitted by linearizing it as a sequence of fingerprints (and indexes for type variables). The tree for the type description is reconstructed at the receiving end by mapping from fingerprints to typecodes, using the type table.

A major change to the Moscow ML backend was to allow transmission of code segments as parts of closures. The CAML Light interpreter (for which Moscow ML is intended) keeps all bytecode together in a single memory block, which is clearly unacceptable for transmitting code. Our backend compiles each code segment into a separate string constant, saved as part of the global data in the output of a compilation. Whereas CAML Light addresses code using 16-bit offsets into the code area, in our implementation there are two 16-bit indexes: first, the global table address of the code string, and second the offset within that string.

3.2 Interpreter

For a given source file, the compiler produces a binary file containing structured literals (including code strings), references to C primitive functions, and the type table information useful at run-time. Each such binary file is loaded in by the interpreter at run-time, and dynamically linked in. Among other things, this means adjusting references to C primitive functions to point to the corresponding entries in a table of C function pointers. A header at the beginning of a bytecode file identifies the code string for the initialization code.

The CAML Light interpreter [52], on which this is based, maintains several data structures for execution. Arguments are passed to functions on an argument stack, and results are left in an accumulator register. Besides a table for global values, there is also an environment for the currently executing closure. A return stack is used to save return points on non-tail recursive function calls.

The main complications that we dealt with in adapting this interpreter, were extending it to support transmissible closures, and to support RTTI. For RTTI, the greatest complication comes with closures. In order to marshall the closure, the environment slots must be marshalled. But the type of a closure tells us nothing about the types of the environment slots. So closures must be “self-describing” [83]: every slot in a closure is a pair, of a value and its type. Every instruction which extends the environment needs to be modified to add this type information also. The type description for each formal parameter is constructed at compile-time, and saved in the bytecode file as a structured literal. These type descriptions are then installed in the global data table as part of loading the bytecode file.

It is of course possible that a type description for a formal parameter may contain free type variables. Type variables are treated at run-time just as ordinary variables, therefore the environment contains bindings both for ordinary variables and for type variables. Type variables in environment type entries are indexes into the corresponding slot in the same environment where the binding for that type variable is stored.

Marshalling a value is based on its type, rather than any run-time tags associated with values. If the type is a datatype, then the type table is used for marshalling. Assuming the value has type $(\tau_1,...,\tau_n)$ t, where t is a datatype defined with type parameters $\alpha_1,...,\alpha_n$, then the value is a tagged memory block, and is marshalled as follows. Since the value is a datatype value, the typecode for t indexes into the corresponding type table entry, and the memory block tag associated with the value indexes into a table of data constructor types. The types of the fields in the memory block are obtained by substituting τ_i for α_i in the corresponding data constructor type, for i=1,...,n. After this substitution, any remaining free type variables correspond to variables bound in the current environment; these variables are looked up in the environment as they are found.

Not surprisingly, marshalling of closures introduces the greatest complications. Essentially the marshalling algorithm must recurse over the closure, marshalling each value in the environment using the associated type information. However the complication is with global values that are referenced in the code. Clearly an index into a global table slot in one address space, carries no meaning if the code with that reference is transmitted to another address space. Therefore the marshaller scans the code string as it marshals a closure; for each reference to a global value that it finds in the code string, it adds a copy of that value to the environment of local values in the closure. Marshalling of the code string then requires that some instructions that access global values be replaced with local equivalents, where the 16-bit offset is now an index into the closure environment.

There are other reasons for modifying the code string during marshalling. Operations on references and arrays must be replaced by calls back to the originating site. Also calls to C primitives (which index a table of C function pointers) must be replaced by references to the names of the C primitives, for relocation during unmarshalling.

3.3 Implementing Distribution

The Obliq extension language uses Modula-3 Network Objects as its transport mechanism. We use a comparable approach, except that we use the ILU multi-language remote object system for transportation [40]. When a mutable reference cell or array is transmitted as an object, we create an ILU object with Get and Set methods, and the network address for this object is transmitted. The ILU objects we use are extremely uninteresting. All marshalling and unmarshalling, and dynamic interface checking, is (for now) done in our implementation.

We provide export and import operations for establishing network connections and performing third-party transfer, and these are implemented using the analogous ILU operations. When the export operation is performed for an ML true object, an ILU true object is generated. This latter has methods for obtaining the memory address and type descriptor of the ML true object. The ILU export ("publish") operation is used to register the ILU object with the ILU name server. Another program obtains the network reference by performing the import operation. This uses the ILU import ("lookup") operation to retrieve the network address of the ILU true object from the name server. The ML import operation then retrieves the type descriptor for the ML true object, and checks that the import operation is well-typed. If this run-time type check succeeds, then an ILU proxy object and an ML proxy object are generated. The ML proxy object uses the ILU proxy object to delegate any method calls to the true ML object at the sending site.

4 Conclusions

We have described the design and implementation of a dialect of ML intended for distributed application programming. A distinguishing characteristic of this implementation is the use of run-time type information, motivated by several constructs in the language. This is intended to motivate the use of run-time types in implementations of polymorphic languages intended for distributed programming.

Although our implementation supports run-time types, and these are used in various ways, we have not yet implemented all of the language constructs described in Section 2. The problem here is how to relate the semantics of type-safe user-definable marshalling, to a real implementation. For example, we do not seriously expect to transmit an unmarshaller with every pickle. More realistically we would expect to transmit the type pickle function, τ_{pickle} [23], as a vector of pairs of fingerprints, and ensure that the unmarshalling code at the receiving site uses the same representation types for pickles. Our current marshaller is similar to the algorithm used in the Modula-3 Pickle module, and for example preserves sharing and does not require a seekable output stream. A naive implementation of the semantics of user-definable marshalling would require an excessive amount of copying and buffering. A more realistic incorporation of this semantics into our marshaller is under development. Finally, the marshalling of closures appears expensive; to what extent can some of this work be done at compile time instead of at run-time, as is done in Knabe's implementation [42].