EXTENDING POLYMORPHISM TO MODULES

by

Jon Shultis

CU-CS-280-84 August, 1984

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

University of Colorado Technical Report No. CU-CS-280-84.

Extending Polymorphism to Modules

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

Abstract

The *capsule* facility extends the notion of type polymorphism to program modules. "Modules" are identified with data algebras, and capsules form a polymorphic calculus of algebras with polymorphic signatures. An innovation of capsules is the addition of a "type structure" to capsule variables, in contrast to the more familiar polymorphic type schemes in which type variables are unrestricted. The details of capsule abstraction and instantiation are discussed, along with an algorithm for inferring the structure of capsules.

1. Introduction

Capsules are an extension of the concept of type polymorphism to modules. A capsule is a program unit supplying a set of operations and possibly depending on other capsules. Capsules are polymorphic in the sense that the structures of parameter capsules are only minimally constrained by the capsule which uses them. For example, a "sorted list" capsule might require only that the element capsule have a binary boolean operation. Capsule structures can be inferred using a variant of the usual procedure for inferring principal type schemes. Capsules can be separately compiled, using a type-checking linker [7], without the need for header files.

A potential drawback of capsules is that they are sometimes less restrictive than one would like. This is because the signature of a capsule does not specify any properties of the operations other than their arities. For example, the "sorted list" capsule would accept an element capsule whose boolean comparison is not a linear order, though the programmer may not have intended it to be used in this way. This cuts two ways, however; a capsule may be useful in ways that the author did not foresee when it was written. In any event, checking that a program segment in any language satisfies an intent is properly the programmer's task, not the compiler's.

§2 defines capsules and their rules of composition. §3 discusses the kind of problems that capsules were invented to solve, and compares capsules to related language facilities. §4 describes the process of capsule instantiation, emphasizing the use of constrained substitutions to determine the compatibility of a particular capsule with a polymorphic specification. Constrained substitutions appear again in the constrained unification used in the capsule structure inference algorithm of §5. §6 summarizes our results and indicates directions for further research.

2. Definitions

Capsules form a calculus of algebras. A *ground* capsule is an algebra; it consists of a set of values and a set of (possibly polymorphic) operations. An *abstract* capsule is an operation taking a fixed number of parameter capsules and resulting in a new capsule, which may be either ground or abstract.

A capsule system is characterized by a set of ground capsules called *primitive capsules*, and a set of abstract capsules called *elementary constructors*. For example, *int* and *bool* are primitives, whereas *product* and *list* are elementary constructors.

A capsule system has one operation, viz. application of an abstract capsule to a list of arguments, also known as *instantiation*. Capsule terms (also known as *arities*) are composed from capsule constant and variable symbols via application in the usual way.

New capsules are formed in one of three ways: instantiation, abstraction, or encapsulation. Any capsule can be abstracted on any capsule variable, in the manner of a λ-calculus. (In fact, in the absence of encapsulation, to be described presently, capsules simplify to the second-order λ-calculus [12].)

An *encapsulation* consists of a capsule specification called the *representation part* and a list of function definitions called the *operation part*. Semantically, the representation capsule is first extended by the definitions, and then the native operations of the representation part are deleted.

The operation part may contain expressions which use operations of parameter capsules, resulting in restrictions on the structures of the actual arguments in an instantiation. A local environment for each capsule records these restrictions for each of its formal parameters. The restrictions take the form of a signature, indicating the operations that the actual argument is expected to supply, and, if the formal is treated as an abstract capsule, any necessary restrictions on *its* formal parameters.

Two restrictions apply to encapsulations. First, the representation part must not contain any free occurrences of capsule variables. More precisely, any capsule symbol appearing in the representation part must be either a parameter of the current capsule or an enclosing capsule, or it must be one of the primitive capsules or elementary constructors. As this implies, capsules may be nested, and they obey the usual scope rules of a λ-calculus. This restriction does *not* prohibit free occurrences of capsule variables in the arities of operators.

The second restriction is that every operation or constant symbol occurring free in the operation part must be statically bound to one of the parameters of the representation part. An occurrence of an operation or constant symbol is said to be *free* if it is not defined in the surrounding environment. (If a capsule contains free occurrences of operation or constant symbols and the representation part has no parameters, the encapsulation is ill-formed.) The capsule structure inference algorithm described in §5 makes a default association of an operation to the leftmost capsule appearing in that operation's arity. The programmer can explicitly override this association with a clause of the form (*op* of *cap*). If the leftmost capsule is a free capsule variable, an explicit association is required.

Any well-formed capsule can be abstracted on any capsule variable, and the stated restrictions will always be satisfied. Instantiation of a capsule is well-formed *iff* the arguments to the capsule are compatible with the structures indicated in the local environment. The details of instantiation will be elaborated in §4. Within these constraints, the calculus of capsules satisfies

α- and β-convertibility. We have not yet investigated the properties of this calculus in much detail, but we conjecture that every capsule system is strongly normalizable [3].

3. Motivation and History

In many respects, capsules resemble ML's abstract types [4], and the syntax used here for capsule specifications is based on ML. Semantically, however, capsules are closer to types in Hope [2]. One main difference is that capsule variables are "typed", i.e. they range over classes of algebras instead of indiscriminately ranging over the class of all value sets as in ML or the second-order λ-calculus. Consequently, encapsulations can use operations that are supplied by their argument capsules. The structure (or "type") of a capsule variable indicates the minimal capsule structure required by the capsule specification in which it is used. The structure of a capsule (variable) consists of a signature and a formal parameter environment which indicates the capsule structure of each of the capsule's formal parameters.

A second main difference between capsules and ML types is the uniform treatment of abstract and ground capsules. In ML, an "abstract type" is not treated as a type at all, but as a macro for generating a type; only fully instantiated abstract types are treated as types. This view of types is maintained in [10] by treating signatures as quasi-first-class objects. Due to the uniform semantic status of capsules the following declarations make perfectly good sense in a capsule system, provided that "list" and "tree" are defined with operations "new", "put", etc., with the proper arities.[†]

[†]A signature mapping mechanism is provided for making explicit associations between actual and formal operations. A more realistic declaration of capsule "bar" in the example might be: *let capsule bar <=> set int (tree using newtree for new, tree'insert for put, ...).*

```
let capsule set *elt *store <=> *store *elt
with    create = absset new
and     insert x,s = absset (put x s)
and     ...;;

let capsule foo <=> set int list;;
let capsule bar <=> set int tree;;
```

Note that this allows a uniform view of the abstract concept "set" which is, to a large extent, independent of the implementation of the "store".

Historically, capsules grew out of our need for a more powerful data abstraction facility to support the construction of multi-language software engineering tools. As an elementary example of the kind of problem we encountered, consider the following functional update function:

```
let update eq fun value point = \x.eq(x,point) => value | fun x;;
update = \ :(*a # *a -> bool) -> (*a -> *b) -> *b -> *a -> (*a -> *b)
```

The parameter "eq" is required in this ML declaration, because there is no way to restrict the domain of fun (*a) to range over varieties (types with equality). In a capsule system, the "eq" parameter can be dropped, resulting in a restriction to the domain of fun.

```
let update fun value point = \x. x = point => value | fun x;;
update = \ :(*a -> *b) -> *b -> *c -> (*a -> *b)
*a must have:
infix = :(*a # *c -> bool)
```

The programmer's likely intent that "point" be of capsule *a is not inferred, but this could always be specified by an explicit restriction of the form (point:*a). (In this example, we assume that the parser has been instructed, either explicitly or by default, to treat "=" as an infix operator symbol.)

4. Compatibility and Instantiation

A capsule specification *cspec* is said to be compatible with a second capsule specification *cspec'* if the signature of *cspec* is compatible with that of *cspec*', and the capsule environment of *cspec* is compatible with that of *cspec*'.

A signature *sig* is said to be compatible with a second signature *sig*' if, for each operator symbol in *sig*', there is a corresponding operator symbol in *sig*, and the arity of the operator in *sig* is compatible with the corresponding arity in *sig*'.

An arity *ar* is compatible with a second arity *ar*' if *ar* is a constrained substitution instance of *ar*'. The constraints arise from the properties specified for bound capsule variables. In ordinary ML, there are only type constants and free type variables. Here, however, an arity may include an instance of a bound capsule variable. In checking compatibility of arities, therefore, it is necessary to ensure that any term (arity) which is substituted for a bound capsule variable be consistent with the constraints on that capsule variable.

It is important to note in this connection that the arity of any operator is always found in the context of a defining capsule specification, so that the constraints (if any) on bound capsule variables are always known. If we did not require that every operation be associated unambiguously with a single capsule in any given context, this checking would be more complicated, but could be done using well-known methods for operator identification a la Ada,† [1] in which each operator is ascribed a set of possible arities as opposed to a single arity; an entire expression is acceptable if the sets of possibilities are eventually narrowed to singletons by mutual constraint. Allowing such random overloading might well turn out to be useful; in fact, we would like to include it in our future capsule-ml compiler, with the provision that operator

†Ada is a trademark of the DOD.

identification be complete for any compilation unit. Including it here would, however, needlessly complicate our discussion and algorithms.

An arity is a syntactic term denoting an instantiation. An important part of checking the compatibility of arities is the computation of the structure of the denoted instantiation. The semantics of instantiation is a slight variation on β-reduction in the simply-typed λ-calculus. First, the actual parameters are checked for compatibility with the corresponding formals. When this succeeds the actuals are substituted for the formals in the signature, and the environment is cleared, since the structures of the actuals are recorded in the surrounding environment. The resulting capsule specification is the specification for the term.

A capsule environment *cenv* in compatible with a second capsule environment *cenv*' if the number of capsule parameters recorded in the two formal parameter environments is the same, and they are pairwise compatible. Note that corresponding capsule parameters need not have the same name; capsule parameter matching is done positionally. This is consistent with the view of abstract capsules as capsule operators, the alpha-conversion rule of lambda-calculi, and the use of positional parameter association in the rest of ML. Of course, name association in the style of Ada would be a viable alternative.

It is heuristically useful to note that, throughout, x is compatible with x' if x is a specialization of x'. In some cases, this means that x may be more elaborate than x', but the converse never obtains.

5. Capsule Inference and Separate Compilation

The process of inferring the structure of an instantiation was described in the preceding section. Inferring the structure of an encapsulation from its declaration involves inferring both a signature for the declaration part and the capsule structures of each of its formal parameters, if any. Simple capsule abstractions are similar to encapsulations, except that the signature consists of the native operations of the body, which corresponds to the representation part of an encapsulation.

The algorithm for inferring the arity of an operation is essentially the same as that used for inferring principal type schemes [5, 11, 8]. There are only two real differences. First, an extra check is needed before a capsule variable can be unified with a term, to ensure compatibility. Two variables can be unified, provided that their structures do not conflict. That is, their formal parameter environments must be mutually compatible, and their signatures must have isomorphic arities for any operation that is common to both. Given two non-conflicting variables, the structure of their unification is the union of their structures. Any additional structure that is thus imposed on a variable is recorded in the appropriate formal parameter environment. This implies that a formal parameter may be required to supply operations that are never explicitly mentioned in the scope of its definition! It is an error for constraints to be imposed on free occurrences of capsule variables.

The second difference between the algorithms is that free operation identifiers are treated as implicit parameters. Once an arity has been computed for an operation, the free operation identifiers and their arities are included in the signatures of the appropriate capsule parameters. Although these are the only apparent differences, the details of arities and their manipulation are quite different, as described above.

Capsules can be compiled separately by storing the inferred capsule structure with the object module as a link table. The linker resolves references in the usual way, but should use the compatibility algorithm to check the interfaces between capsules.

Levy has recently described a system for separate compilation of Pascal modules which also infers the arities of free operation identifiers and uses a type-checking linker [9]. The major differences between capsule structure inference and Levy's system are due more to the differences between capsules and Pascal types than to the basic strategies for inference, compilation, and linking.

6. Summary

The capsule compatibility and inference algorithms have been coded in ML and are currently being tested and refined. Our work so far has focussed on the syntactic aspects of capsules and capsule inference as described in this paper. Our next concern is to explore issues in the implementation and use of capsule systems. In the area of implementation, we are particularly interested in measuring the amount of extra overhead incurred by capsule inference, and the production of efficient code for capsules. Our plan is to modify Johnsson's lazy ML compiler [6] to use capsules, and to compare his statistics with ours for similar programs, as well as to compare programs written using capsules with their corresponding non-polymorphic instances. In the area of usage, we have already found capsules to be more expressive than we had initially anticipated, and it is as yet difficult to distinguish between the limits of our inexperience as capsule programmers and the limitations of capsules themselves. For example, we found it easy enough to define a capsule *category*, and a capsule *functor* mapping categories to categories, but were unable to define an operation *mkfunctor* that would construct a *functor* directly from the usual mathematical data, viz. the object and morphism mappings. At first we suspected a weakness of capsules, but later realized that the problem is that there *is* no general way to construct the identities and composition of the target category directly from these data.

References

1. ANSI, *MIL-STD-1815A Ada Programming Language*, U. S. Government (Ada Joint Program Office) (Jan. 1983).

2. Burstall, R. M., D. B. MacQueen, and D. T. Sannella, "HOPE: An Experimental Applicative Language," *Proc. 1980 LISP Conference*, pp. 136-143 (August 1980).

3. Fortune, S., D. Leivant, and M. O'Donnell, "The Expressiveness of Simple and Second-Order Type Structures," *J. ACM* **30**(1) pp. 151-185 (Jan. 1983).

4. Gordon, M. J., A. J. Milner, and C. P. Wadsworth, "Edinburgh LCF," in *Lecture Notes in Computer Science, no. 78*, Springer-Verlag, Berlin (1979).

5. Hindley, R., "The Principal Type Scheme of an Object in Combinatory Logic," *Trans. Amer. Math. Society* **146** pp. 29-60 (1969).

6. Johnsson, T., "Efficient Compilation of Lazy Evaluation," *Proc. ACM Sigplan '84 Symp. on Compiler Construction*, pp. 58-69 (June 1984).

7. Kieburtz, R. B., W. Barabash, and C. Hill, "A Type-checking Program Linkage System for Pascal," *Proc. 3rd Intl. Conf. on Software Engineering*, pp. 23-28 (1978).

8. Leivant, D., "Polymorphic Type Inference," *Proc. 10th ACM Symp. on Principles of Programming Languages*, pp. 88-98 (Jan. 1983).

9. Levy, M. R., "Type Checking, Separate Compilation and Reusability," *Proc. ACM Sigplan '84 Symp. on Compiler Construction*, pp. 285-289 (June 1984).

10. MacQueen, D., "Modules for Standard ML," *Proc. 1984 ACM Symp. on LISP and Functional Programming*, p. ? (Aug. 1984).

11. Milner, R., "A Theory of Type Polymorphism in Programming," *J. Comp. & Sys. Sci.* **17** pp. 348-375 (1978).

12. Reynolds, J. C., "Towards a Theory of Type Structure," Systems an Information Science, Syracuse University (April, 1974).